
Probability *Explained*

by

Christopher R. Genovese

Contents

Part I	Finite Random Processes	1
1	Fixed Random Payoffs	7
1.1	FRPs	9
1.2	Kinds	14
1.3	What a Kind Tells Us	23
1.4	FRPs as Data, Kinds as Models	39
1.5	The Big 3+1 and the Likelihood Principle!	69
1.5.1	The Big 3+1: Fundamental Operations on FRPs and Kinds	69
1.5.2	Three Interpretations of the Big 3+1 Operations	71
1.5.3	Preview of the Big 3+1	74
1.5.4	Learning from Data: The Likelihood Principle	81
2	Transforming with Statistics	83
2.1	Statistics and Data Processing	84
2.1.1	Statistics in <code>frplib</code>	95
2.2	Transformed FRPs	115
2.3	Transformed Kinds	134
2.4	Projections and Marginals	153
2.5	Extended Examples	161
3	Equivalent Kinds and Canonical Forms	197
4	Building with Joins	211
4.1	The Evolution of Random Systems	212
4.2	Independent Joins	223
4.3	Conditional FRPs and Conditional Kinds	258
4.4	General Joins	271
5	Constraining with Observations	325

5.1	Accounting for Partial Information	326
5.2	Events	333
5.3	Observations	337
5.4	Bayes's Rule	361
5.5	Observations and Independence	369
5.6	Observations and Conditional Kinds	376
6	Predicting with Expectations	381
6.1	Fundamental Properties of Risk-Neutral Prices	390
6.2	Computing Expectations	414
6.3	Kinds and Expectations	425
6.4	Probabilities are the Expectations of Events	431
7	Three Dialogues: Computation, Systems, Simulation	435
7.1	A Dialogue on Computation	436
7.2	A Dialogue on Systems and State	445
7.3	A Dialogue on Solutions and Simulation	460
8	Patterns, Predictions, and Practice	477
8.1	Mind the Types	478
8.2	A Pattern Language for Random Systems	482
8.3	Simple Finite Random Processes	488
8.4	Strategies and Representations	505
8.5	Using Observations	514
8.6	Touching Infinity	521
Interlude F	Functions	535
9	Sets	541
9.1	Specifying Sets	543
9.2	Increments and Intervals	544
9.3	Making New Sets from Old Ones	545
9.4	To Infinity and Beyond	549
10	Function Foundations	551
10.1	Common Operations	580
10.2	Special Functions	585

10.3 Operators	585
11 Anonymous Functions	605
12 Indicator Functions	609
13 Composition	619
13.1 The Operation of Composition	621
13.2 Commutative Diagrams	627
13.3 Inverse Functions	636
13.4 Describing Structure	647
14 Function Properties	653
14.1 Input-Output Correspondence	654
14.2 Analytical Constraints	665
15 Tuple/Vector Functions	671
15.1 Functions on Tuples	672
15.2 Projections	673
15.3 Permutations	674
15.4 Joining Tuples	675
15.5 Combining Tuple Functions	676
16 Relations	679
17 Algebraic Structures	695
17.1 Associative Operators: Monoids, Semirings, and Beyond	702
17.2 Graphs and Matrices: An Enlightening Connection	731
17.3 Vector Spaces and Linearity	745
17.4 Orders	779
18 Application: Counting	793
18.1 Arithmetic Redux	794
18.2 How to Count	799
18.3 A Framework for Common Counting Problems	805
18.4 Unique Marbles in Labeled Pouches	811
18.5 Unique Marbles in Labeled Tubes	814
18.6 Identical Marbles, Labeled Batches	816
18.7 Identical Marbles, Unlabeled Batches	819

18.8 Unique Marbles in Unlabeled Pouches	820
18.9 Unique Marbles in Unlabeled Tubes	822
18.10 Permutations and Cycles	824
18.11 The Inclusion-Exclusion Principle	825
Part II Distributions	827
19 Kinds, Kernels, and Distributions	831
19.1 The Kernel	833
19.2 The Distribution Operator	837
19.3 Converting Among Kind, Kernel, and Distribution	842
19.4 Conditional Kinds and Conditional FRPs	845
19.5 Transforming with Statistics	851
19.6 Building with Joins	854
19.7 Constraining with Observations	860
19.8 Predicting with Expectations	863
20 Infinities, Large and Small	865
20.1 Infinite Kinds	866
20.2 Mass and Density	870
20.3 On Infinitesimals, Differentials, and Measure	878
20.4 Distributions and Kernels Revisited	886
20.5 FRPs, Random Processes, and Random Variables	891
21 Case Study: Symmetry and the Uniform Distributions	895
21.1 Symmetry for Finite Sets	902
21.2 Symmetry for Continuous Regions	910
22 Predicting with Expectations	919
22.1 The Expectation Rules	922
22.2 The Mass-Balancing Equation	928
23 Constraining with Observations	939
23.1 Review: Conditions, Events, Boolean Expressions	941
23.2 Technique: The Logic of Events	945
23.3 Constraining with Observations	953
23.4 The Expectation Updating Equation	957

23.5 Conditional Distributions, Conditional Expectations	959
24 Building with Joins	963
24.1 Independence	966
24.2 Conditional Independence	982
24.3 The Updating Equation	986
24.4 The Multiplication Rule	990
24.5 The Mighty Conditioning Identity	999
25 Transforming with Statistics	1013
25.1 Special Case: Projections	1015
25.2 Technique: Composition	1018
25.3 Technique: Determining Functions	1024
25.4 Technique: Change of Variables	1033
26 Variance and Correlation	1037
26.1 Properties of Variance	1041
26.2 Covariance	1043
26.3 The Variance of a Sum	1044
26.4 Higher Dimensional Preliminaries	1046
26.5 Variance in Higher Dimensions	1048
27 Conditional Independence and Graphical Models	1051
27.1 Directed Acyclic Graphs	1052
27.2 Kernel Decompositions	1052
27.3 Graphical Models	1056
Interlude C Canonical Distributions	1063
28 Canonical Distributions	1065
29 The Normal Distributions	1071
30 The Bernoulli Trials Process	1079
30.1 Counting the Number of Marks in a Fixed Number of Trials	1081
30.2 Waiting for Marks	1082
30.3 Counting Rare Marks	1086
30.4 Random Uniforms	1087

30.5 Random Walks	1088
30.6 Normals Everywhere	1088

Part III Approximations **1089**

31 The Joy of Approximation **1091**

31.1 Approximating Distributions	1093
31.2 Approximating Sums of Random Variables	1097
31.3 Asymptotic Order of the Remainder	1100
31.4 Review: Three Important Canonical Distributions	1101

32 The Constant Approximation **1107**

33 The Normal Approximation **1113**

34 The Poisson Approximation **1121**

35 Concentration Inequalities **1127**

35.1 Markov and Chebychev Inequalities	1132
35.2 Chernoff Bounds	1134
Aside: The Moment Generating Function	1137
35.3 Hoeffding's Inequality	1142
35.4 More Examples	1145

Interlude Z Generating Functions **1149**

36 The Power of Alternate Representations **1151**

36.1 Counting Configurations	1152
36.2 Power Series and Their Coefficients	1153
36.3 Operating on Generating Functions	1154
36.4 Distribution Example	1155

37 Probability Generating Functions **1157**

37.1 Basics	1158
37.2 Distributional Properties	1159
37.3 Operating on Generating Functions	1160

38 Moment Generating Functions **1161**

38.1 Basics	1162
38.2 Distributional Properties	1163
38.3 Operating on MGFs	1164
38.4 Cumulant Generating Functions	1165
39 Characteristic Generating Functions	1167
39.1 A Primer on Complex Numbers	1168
39.2 Basics: The Fourier Transform	1169
39.3 Distributional Properties	1170
39.4 Operating on CGFs	1171
 Part IV Stochastic Processes	 1173
 Part V Information and Entropy	 1175
 Interlude G Graphs	 1177
 Part VI Learning from Data	 1179
40 Probability versus Statistics	1181
40.1 Forward and Inverse Problems	1182
40.2 Statistical Models	1183
40.3 What does it mean to learn from data?	1184
40.4 The Frequentist Framework	1185
40.5 The Bayesian Framework	1186
41 Estimation and Inference	1187
41.1 Statistical Problems	1188
41.2 Maximum Likelihood Estimation	1189
41.3 Bayes and Other Estimation Methods	1190
41.4 Confidence Sets	1191
41.5 Posterior Distributions	1192
42 Regression	1193
42.1 The Regression Problem	1194
42.2 Smoothing	1195

42.3 Linear Regression	1196
42.4 A Geometric Perspective	1197
42.5 Neural Networks and More	1198
43 Classification	1199
43.1 The Classification Problem	1200
43.2 High-Dimensional Space	1201
43.3 Methods	1202
Index	1203

Part I

Finite Random Processes

Probability theory is one of the great intellectual achievements of the past few hundred years. It puts the ineffable idea of randomness on a firm mathematical footing and gives a calculus of the random that can be applied to practical problems. People have famously poor intuition about uncertainty and its effects, but probability theory offers a tool that helps us understand an uncertain world and make better decisions. It plays a fundamental role in many fields because what we measure and learn about the world is fraught with uncertainty.

Probability theory is at the heart of statistics, machine learning, finance, and many subfields in computer science. For students in those areas, facility with probabilistic reasoning and computation is a significant advantage. This book is designed for students in such areas to take you past the surface formulas to gain a deeper understanding of how we build and analyze random systems.

The basic setup is that we make observations and measurements of a random system as it evolves and use them to make predictions and decisions about what will happen. The fundamental model of probability theory is surprisingly simple: nature picks a ticket out of a box and that ticket determines the path that the random system's evolution takes, and thus determines all that we observe and measure from the system. The randomness is concentrated solely in the choice of ticket. The mathematical complexity arises because there are infinitely many tickets in the box – an *uncountably* infinite collection of tickets.

Still, the most common operation in probability is the *weighted average*, and the fundamental result is easy to state: if we are measuring a random quantity that can have values in some set $\mathcal{V}$ and if we process that measurement through a data-processing algorithm ψ , then our prediction of the output of our algorithm looks like

$$\sum_{v \in \mathcal{V}} k(v) \psi(v),$$

where k is some function that returns numbers between 0 and 1 such that

$$\sum_{v \in \mathcal{V}} k(v) = 1.$$

The numbers $k(v)$ is a measure of the propensity – called a probability – for us to observe value v for the quantity that we are measuring. So the predicted output of our algorithm is a weighted average of the possible outputs of the algorithm, where the weights reflect somehow the nature of the randomness in the system.

Most of what we do in probability is an elaboration on that idea, but it leave

questions. For what kind of systems does this apply? What is a random quantity? How do we describe and represent it? How do we find the function k ? What do probabilities really mean? How do we account for information that we have or that we observe along the way? What does this prediction mean and how do we use it?

In Part I of this book, we will answer these questions with an eye to developing tools and intuition about how to describe, build, and analyze random systems. Rather than start with an abstract model of probability and a mathematical approach, we will start with a *physical model* of probability and take a more *computational* approach. Rather than deal at the outset with infinities, which are the source of all technical complications in the theory, we will focus in the Part on *finite systems*.

Our physical model is based a *device* called a Fixed Random Payoff, or FRP for short. Each random quantity that we might measure or observe as a system evolves is represented by one FRP. We build more complicated random systems by wiring FRPs together. Happily, there are only three basic wiring diagrams that we need to use to build any finite system. Each of these wiring diagrams corresponding to an *operation* that acts FRPs. We use one additional operation to extract predicted answers to our questions about the system’s evolution and output. Together these are the **Big 3+1** operations.

The FRP is a physical device. We can build them and connect them. But rather than worry about wires and solder, we will use software to build and operate on our FRPs to our specifications. Part I is integrated with `frplib`, a library and interactive environment for probabilistic computation.

Each FRP has what we will call a *Kind*. The Kind captures everything we know about an FRP’s value and is what we use to analyze and make predictions about an FRP and the random systems it represents. The Big 3+1 operations on FRPs also act in parallel on Kinds, and we use those operations to analyze the systems that we build. The `frplib` software has tools to construct, view, and operate on Kinds. In probability theory, we typically do our analysis *before* observing the random system, so working with Kinds and the predictions derived them ends up being our primary focus.

Part I offers a novel take on probability theory. We will learn perhaps a dozen non-standard terms, but all of these directly generalize to standard ideas, as we will see in Part II. For instance, FRPs are special cases of “random variables,” and Kinds are special cases of “distributions.”

I use this novel approach for many reasons. First, the physical model helps us build more concrete intuition about what these objects mean. A “random variable”

is a vague and abstract thing, but an FRP better reveals its true nature. Second, traditional approaches to probability theory build towards complexity by introducing ideas and operations that seem different when they are in fact unified. We will exploit that unification from the start, with the only jump being from finite systems in Part 1 to arbitrary systems in Part 2. This reveals that there are only a few key operations to master, operations apply across different contexts but mean the same thing. Third, with this approach, we can tackle more interesting and complex problems right from the start. The operations we use to build systems all have a mathematical expression that will become familiar as we use them computationally. By the time we start Part 2, the fundamentals of the theory will be clear, with only the technicalities of dealing with infinities to grapple with. Finally, a computational approach lets you experiment with the ideas in a deeper and more satisfying way. Almost every significant example in Part 1 has a module in `frplib`. You can rerun the computations, tweak them, and build on them. And as we work through Part 1, you are strongly encouraged to follow along in the `frplib` playground. This makes the many examples in the book more impactful for your learning.

OVERVIEW OF FRPLIB. Talking about FRPs and Kinds is useful, but we get even more by building FRPs and Kinds, manipulating them, and using them to compute predictions for interesting random systems. For this purpose, we will use the `frplib` software.

This is a package built on the Python programming language that can be used both as an *interactive laboratory* for working with FRPs and Kinds or as a *library* within your own programs. This system encourages you to play with the examples, check your understanding against output, and engage more deeply with the ideas. You do not need to know Python to use this system; the book will show you how to use it.

Instructions for downloading and installing `frplib` are available at <https://github.com/genovese/frplib>.¹ When the package has been installed, two things happen: from within your Python programs, you will be able to import functions and data from `frplib`, and the application `frp` will be installed on your system. The `frp` application is run from your terminal command line and opens an interactive session for computing with FRPs, Kinds, and related quantities.

Within a Python program, you can load modules from the `frplib` library. For example:

```
from frplib.frps      import FRP, frp, conditional_frp
```

¹Mac OS, Windows, and Linux are all supported.

```
from frplib.kinds      import Kind, kind, constant, either, uniform
from frplib.statistics import statistic, __, Proj, Sum
```

The library documentation describes the available modules, functions, and data, along with shortcuts for importing some commonly used configurations. Modules within the sub-package `frplib.examples` include functions and data used in examples within this book. You will see such modules loaded or mentioned in the text and are encouraged to follow along in `frplib` as you read.

The terminal application `frp` is invoked with various *sub-commands* that open different interactive environments. The sub-commands two we will focus on are `playground` and `market`, especially the former. To invoke these sub-commands, enter `frp playground` or `frp market` at the terminal command line prompt. (There are several ways to start the application, as described in the instructions on the GitHub page.) The playground is an interactive REPL (Read-Eval-Print Loop) where you can do arbitrary computations. It has tools and interface elements to make it easy to use and includes a full interactive help system. The market gives direct access to the FRP Marketplace, where you can order FRPs of any configuration you like. Fortunately, the market functionality is also available through the playground.

When you enter `frp playground`, you will see a prompt `playground>` at which you can enter commands or other Python code. This code can span multiple lines; you end a code block by hitting return on a blank line. In the playground, you can move across the code, even multiple lines, and edit it before final submission. You can also access and edit your interaction history. The playground pre-loads all the most commonly used `frplib` functions and classes, so you can use them easily. In addition, you can import any `frplib` or other installed Python package from the playground prompt. To begin, start by entering `info()`. This will open the interactive help system that you can explore. The topic *Using the Playground* gives more detail.

When you enter `frp market`, you will see a prompt `market>` at which you can enter tasks for the market to perform. These can span multiple lines and must end in a period (`.`). After the first line of a multi-line task, the prompt will change to `...>` signaling that you can continue entering information. A `.` at the end of a line will complete the input. However, if the input is ill-formed in any way, the task will not be submitted; instead, an error message will identify the problem, allowing you to fix it. To end your session, enter “`exit.`” or “`done.`” at a fresh prompt. Enter “`help.`” for assistance.

The `frplib` library provides classes and methods relating to all the main concepts we cover in this chapter, including FRPs, Kinds, and Statistics. For each of these,

the library defines **factories**, which are functions for creating objects of the specified type with particular properties; **combinators** for combining several objects of a specified type into a new one; along with various **actions** and **utilities**. See the “Playground Overview” on page 195 and the `frplib` Cheatsheet for a summary, along with the many examples in this chapter.

Fixed Random Payoffs

1

Chapter

Contents

1.1	FRPs	9
1.2	Kinds	14
1.3	What a Kind Tells Us	23
1.4	FRPs as Data, Kinds as Models	39
1.5	The Big 3+1 and the Likelihood Principle!	69
1.5.1	The Big 3+1: Fundamental Operations on FRPs and Kinds	69
1.5.2	Three Interpretations of the Big 3+1 Operations	71
1.5.3	Preview of the Big 3+1	74
1.5.4	Learning from Data: The Likelihood Principle	81

You and a friend are minding your own business one day, when a stranger appears.

YOU: (*speaking to a friend*) ... So let's flip a coin to decide—

STRANGER: (*appearing suddenly from the bushes*) No, no. Don't do that. Flipping coins is so 2010s. There's a better way!

YOU: What the ... who are you??

STRANGER: (*handing you a box*) Use this instead. Push the big red button.

YOU: (*eyeing the box and the stranger suspiciously*) What is it?

STRANGER: The display is blank but look at it after you push the button.

YOU: (*reluctantly pushing button*) It shows a "1".

STRANGER: You got a heads.

YOU: Why doesn't it just say heads?

STRANGER: Well, the boxes use numbers. You can encode anything with them. Here, we're using "1" for heads and "0" for tails.

YOU: Hmm, OK. How do I know it's fair?

STRANGER: Oh, it's fair! Better than real coins, in fact, which are actually quite bad that way. See the diagram on the box's metallic display? That tells you it's fair.

YOU: (*looking*) I'm not sure exactly what that tells me. But in any case, I got a "1". It could have come from anywhere.

STRANGER: Sure, I get it. Gnomes in the box, making stuff up. But that's the question, eh? How do we know where anything comes from?

YOU: Just my luck, a philosopher in the bushes.

STRANGER: If I gave you a bunch of boxes of the same kind, you'd see it more clearly.

YOU: (*pushing the button again*) So let's do another flip: 1 again, 1 *again*, 1, 1, another 1. Something is certainly fishy here.

STRANGER: If you want to do another flip, you need another box. That one will only ever show the "1" you got when you first activated it.

YOU: That doesn't seem terribly convenient.

STRANGER: (*shrugs*) The box represents a specific *flip* not the coin. Can you unflip a coin once you flip it? No.

YOU: So if I wanted to flip ten coins?

STRANGER: You'd get ten boxes and connect them together. Push the button and you'll see a list of ten 0s and 1s telling you the results of the flips.

YOU: Isn't a coin easier?

STRANGER: Well, for some things. But you can use the boxes to represent almost any random system, not just coins. The same boxes, the same connections, the same operations and you've got it all. A coin is just a coin.

YOU: (*skeptical but intrigued*) How do I learn more?

STRANGER: (*Handing you a very large stack of paper*) Read this!

YOU: (*eyes wide*) Isn't paper so 2010s?

STRANGER: (*walking away*) Don't worry, there's a link in there. Sometimes it helps to have something physical to hold onto.

YOU: (*calling*) What do you call these things?

STRANGER: FRPs. Good luck!

1.1 FRPs

An FRP is a closed box whose top face has an input port, several output ports, a large red button with an internal LED, a color display, and a smaller metallic display. (Figure 1.1.) We can neither open the box nor see what is inside it, directly or indirectly.

An FRP does one thing, one time – it produces a value. Eventually. Every FRP proceeds through three states. Initially, each FRP is **fresh**, meaning that it has not yet produced a value. With a fresh FRP, the Color Display is blank, and the internal LED in the button is illuminated. At some point later, it is **activated**, causing it to produce a value internally. With an activated FRP, the button's LED turns off for good. Finally, the first time the FRP's button is pushed, it is first activated (if not already) and then becomes **observed**. With an observed FRP, the *value of that FRP* is shown on the Color Display and *remains visible thereafter*. (We will see later that there are ways to activate an FRP that do not make it observed.) Figure 1.2 depicts these three states.

When an FRP is fresh, its eventual value is uncertain. Once it is activated, that value is *fixed* for all time (the F in FRP), even if you do not know it. Once it is observed, you know what that value is. The value that an FRP produces is *random* (the R in FRP) in a sense that we will explore. You do not know where the value comes from or how it was produced. It could be the result of a complex physical process or a value that a group of gnomes living inside the box finds amusing. But it is not arbitrary. The Kind shown on the metallic display describes what we know

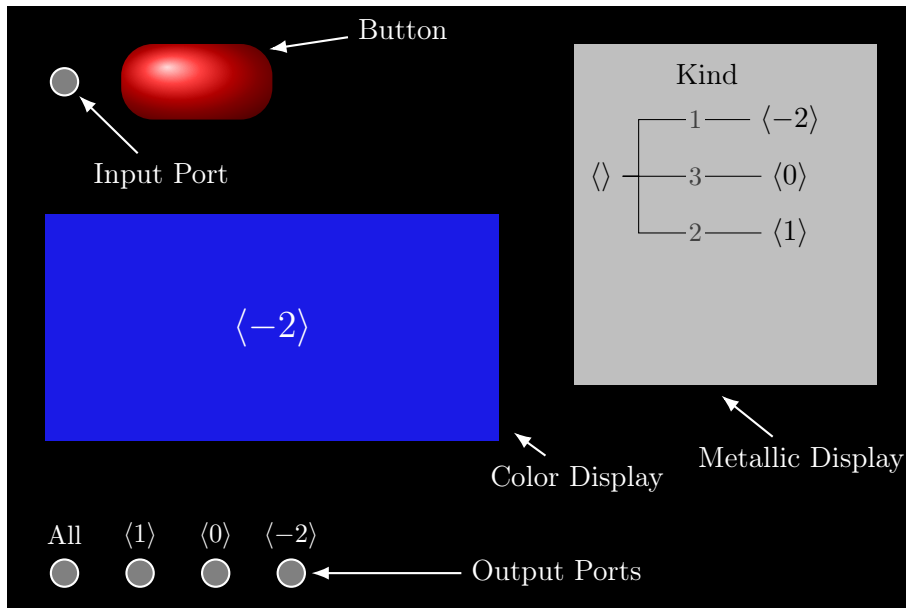


FIGURE 1.1. The top face of a typical fixed random payoff box, or FRP. When the FRP is *fresh*, the button’s internal LED is lit, and the color display is off. Once the button has been pushed, the button’s LED turns off, and the color display shows a value thereafter. The “Kind” shown in the Metallic Display is described in Section 1.2, and the uses of the input and output ports are described in Chapters 2 and 4.

about that value and will let us make predictions about it, as we will see in the next section. If you own an FRP, you hold the *promise* of receiving its value as a *payoff* (the P in FRP) through the “FRP Marketplace”. The price we would pay for an FRP is based on our best predictions of the payoff we will receive.

While the values produced by an FRP can, in principle, be objects of any type, we will almost exclusively use FRPs whose values are *tuples*¹ of numbers. A tuple is a list of finite length whose components are indexed by their position in the list, and in a numeric tuple, the components are all numbers. (We do allow a special numeric quantity, called **nothing**, in the components that is occasionally use for padding. Any arithmetic operation on **nothing** and another number produces **nothing**.) FRPs that produce only numeric tuples are called *numeric FRPs*, and unless otherwise indicated, we will use **value** to mean a *tuple of numbers*.

As discussed in Interlude F, we use angle brackets $\langle \rangle$ to delimit lists and tuples.² For example, $\langle \rangle$ is the empty tuple, $\langle 1 \rangle$ has a single component 1, $\langle 0, 1 \rangle$ has two components with first element 0, $\langle -1, 2, 32 \rangle$ has three components with first element -1, and so forth. The *dimension* of a tuple is the number of components it has. For example, $\langle \rangle$, $\langle 1 \rangle$, $\langle 0, 1 \rangle$, and $\langle -1, 2, 32 \rangle$ have respective dimensions 0, 1, 2, and 3. A

¹See page 547, Chapter 15, and Section 17.2 in Interlude F.

²It is common to use parentheses for lists, tuples, and vectors like $(3, 4, 5)$. This is fine, but parentheses are so frequently used and overloaded that it is helpful to have a more salient delimiter for this purpose.

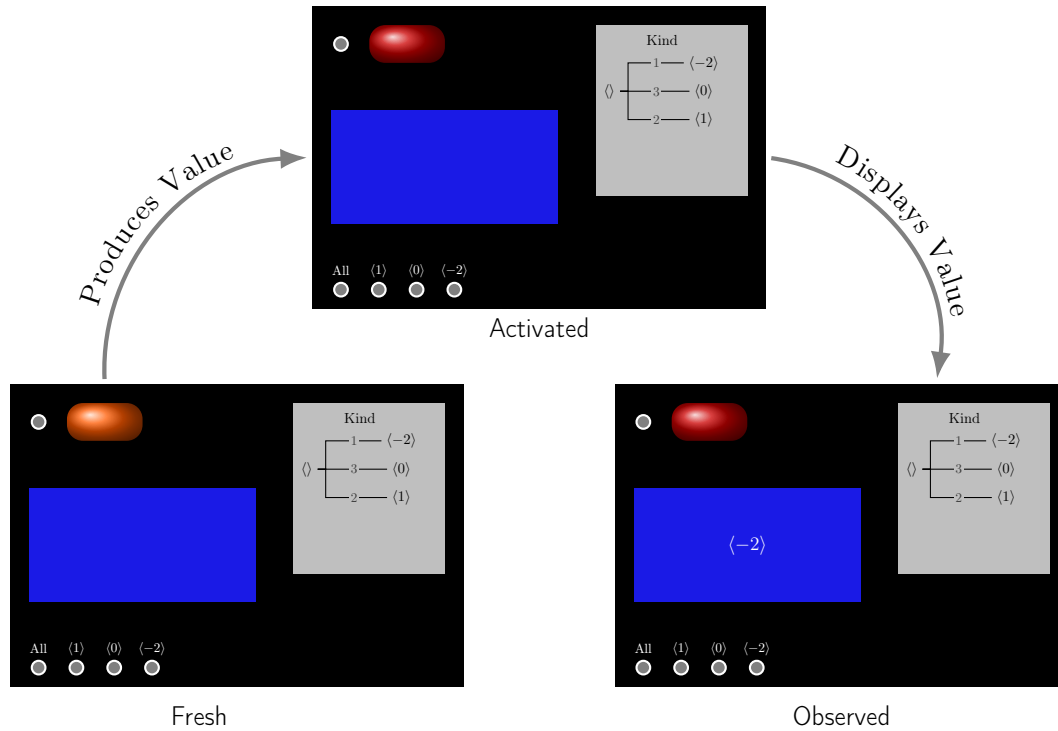


FIGURE 1.2. The states of an FRP. Each FRP starts out as *fresh*, depicted on the lower left, where it has not yet produced a value. The LED in the button is lit (note the lighter color in the picture), and the color display is blank. At some point, the FRP is *activated* and produces a value internally, depicted on the top center. The LED in the button is off, and the color display remains blank. The value has been fixed but is not known. As soon as the button is pressed for the first time, the FRP becomes *observed*. It is activated (if not already), and its value is shown on the color display, remaining visible thereafter.

tuple of dimension 1 is a special case because in practice, we make no distinction between a tuple with one component and the quantity it contains. So if I give you $\langle 42 \rangle$, for all practical purposes I have given you the number 42, and vice versa. A tuple of dimension 1 is called a **scalar**.

We add one requirement on the (numeric) FRPs that we use: that *all possible values* that the FRP might produce are tuples of the *same dimension*. If an FRP's values have common dimension d , we say that the FRP itself has **dimension d** . An FRP of dimension 1 is called a **scalar FRP**. This dimensional requirement is not essential, but it makes our work more convenient in several ways at little cost in generality.

An FRP's value is randomly produced and fixed for all time. We will explore the nature of that randomness in the coming sections, but first it will be helpful to get

some hands-on experience with FRPs. For this, we will use `frplib`, a software package built on the Python programming language that can be used both as an *interactive laboratory* for working with FRPs and as a *library* for probabilistic programming. The introduction to Part I on page 4 gives a brief overview of installing and using the package and of getting help. Install the package as instructed, open a terminal window, enter `frp playground`, and follow along. Text from a character `#` to the end of a line is a comment for your benefit; you need not enter such comments. The `pgd>` is a shortened form of the playground prompt that you will after entering each statement.

We will load some FRPs that are defined in an `frplib` module:

```
pgd> from frplib.examples.hello_world import X, T, V
```

This defines three fresh FRPs which we name X , T , and V . (Because the FRPs give a random value, the values you see might differ from what I show here.)

```
pgd> X.is_fresh          # This is a property of the FRP X.
True
pgd> T.is_fresh
True
pgd> V.is_fresh
True
pgd> X                  # By printing it, X becomes observed
An FRP with value <9>
pgd> X.is_fresh
False
pgd> X                  # Once activated, X's value is fixed.
An FRP with value <9>
pgd> X.value            # Accessing the value directly
<9>
pgd> T
An FRP with value <4.2, 3.1>
pgd> T.is_fresh
False
pgd> activate(V)        # V is activated but not observed
pgd> V.is_fresh
False
pgd> V                  # Now it's observed
An FRP with value <1, 7, 9, 3, -2, 6, 10>
```

First, note that we give each FRP a *name*. Our convention is that FRPs are named with uppercase Roman letters, with optional subscript labels. Notice that X has dimension 1, T has dimension 2, and V has dimension 7. There is a unique FRP of dimension 0:

```
pgd> FRP.empty
The empty FRP of dimension 0 with value <>
```

Second, when an FRP is created, it is fresh, and in `frplib`, we see this by the associated `.is_fresh` property. Examining the FRP’s value, either by printing it or through its `.value` property, corresponds to pushing its button, putting the FRP into an observed state. Once the button is pushed, an FRP’s value remains unchanged thereafter. We see that above with X .

Naming Convention: FRPs. We typically name FRPs with single, uppercase Roman letters, with optional labels attached as subscripts. Thus, X , T , and V here, or for example: A_2 , H_{red} , and $Z_{3,7}$. We usually try to make the base letter meaningful or mnemonic regarding the nature or meaning of the FRP’s value.

The *FRP Marketplace* is the primary supplier of FRPs, where you can obtain any number of FRPs of any Kinds you choose. For a price, you obtain a fresh FRP (of the chosen Kind) from the marketplace and a contract with the following stipulations: If it is a scalar FRP that shows value v after pushing its button, then (i) if $v > 0$, you may return the FRP and receive $\$v$, and (ii) if $v < 0$, you are obligated to return the FRP and *pay* $\$|v|$. (We will consider the case of FRPs with dimension > 1 later.) Thus, we can interpret an FRP as the *promise* of a *payoff*.

The price we are willing to pay to the FRP Marketplace for such a contract will therefore be based on our best *prediction* of an FRP’s value. A good prediction incorporates all the information we have about the value. Because that value is random, it is inherently uncertain, and the accuracy of our prediction will depend on the degree of uncertainty. Establishing how much an FRP is worth – that is, finding our best prediction of its value – is at the core of probability theory and will be a central goal for the rest of this Part. To do this, we will need more precise definitions of words like “prediction,” “randomness,” “accuracy,” and “uncertainty.”

Our first step will be to understand an FRP’s *Kind*, which is depicted on its metallic display, what it means and how to use it. We turn to that next.

Key Take Aways

A **Fixed Random Payoff** box, or **FRP** for short, is a device that does one thing, one time: it produces a value. Once the value is produced it is fixed for all time, but *random*. The value represents the promise of a payoff to the FRP's owner, and how much an FRP is worth is based on our prediction of its value.

The lifecycle of an FRP has three stages. It starts off as *fresh*, with no value produced. Then at some point it is *activated* in which a value has been produced but is not displayed. And finally, it is *observed* in which its value is permanently displayed and thus known.

We use FRPs to represent random quantities that we might observe or measure as we watch a random system evolve. The key value of an FRP is that we can reason about its value – and its relationship to other random quantities also represented by FRPs – *before we observe its value*. We name FRPs with capital Roman letters, optionally decorated with labels in subscripts or superscripts.

1.2 Kinds

An FRP's **Kind** is depicted on its metallic display. When the FRP is fresh, the Kind embodies all of our knowledge about the FRP's value. Two fresh FRPs *of the same Kind*, considered in isolation, are interchangeable to us. Although they may eventually produce different values, our knowledge and predictions about their values are the same before their buttons are pushed. Think of two FRPs with the same Kind as like two individuals of the same species, acting as individuals yet sharing some essential features. In this section, we define the structure of Kinds, what they are and how they are built. In the next section, we consider what they mean and see why Kinds are fundamental to pricing FRPs.

A Kind is represented by a *rooted tree*³ that is *complete*, meaning that every path from the root to a leaf has the same number of steps. Each edge in the tree has an associated *positive* number called its **weight**, and each node holds a *value* (i.e., a tuple of numbers). The root node holds the empty tuple $\langle \rangle$, and all the nodes within each level of the tree must hold *distinct* values of the *same* type and dimension. On every step on a path from root to leaf, the dimensions of the values in the nodes must increase. The leaf nodes collectively hold all the **possible values** that an FRP with

³A connected, acyclic, undirected graph with one node designated as the "root." See Interlude G.

that Kind might produce.

Figure 1.3 shows a few simple examples of Kinds. We display Kinds horizontally with the root at the *left* and the leaves at the *right* rather than the more common root-at-the-top display. This layout offers several advantages for us, including compactness, easier comparison of values and weights, and simpler output for the programs we use. (Figures 1.3 and 1.4 show the same Kinds with horizontal and vertical layout to help you get used to the horizontal layout.) FRPs of the left Kind in Figure 1.3 can only ever output the single value $\langle 2 \rangle$. Those of the middle Kind in Figure 1.3 can output either $\langle -1 \rangle$ or $\langle 1 \rangle$, and those of the right Kind can output $\langle -1 \rangle$, $\langle 0 \rangle$, or $\langle 9 \rangle$. Again, we do not distinguish in practice between tuples with one number and the number itself, so we can think of FRPs with these Kinds as outputting random numbers from the sets $\{1\}$, $\{-1, 1\}$, and $\{-1, 0, 9\}$. The weights are displayed along the edges. For example, in the right Kind, the weights are $1/4$, $1/2$, $1/4$ for the associated values $\langle -1 \rangle$, $\langle 0 \rangle$, and $\langle 9 \rangle$. We will learn how to interpret the weights on these Kinds in the next section.

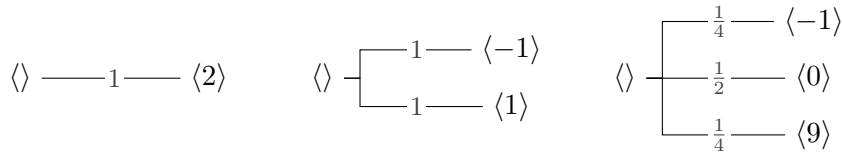


FIGURE 1.3. Several FRP Kinds, all with dimension 1 (scalar) and with sizes 1, 2, and 3.

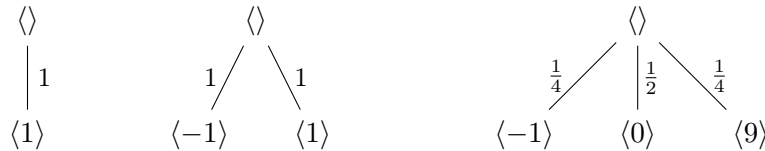


FIGURE 1.4. The same Kinds as in Figure 1.3 but displayed with the root at the top.

All the Kinds in Figure 1.3 have possible values of dimension 1 (i.e., scalars). Figure 1.5 shows an example with 2-dimensional values, and by examination, we can see that it follows the rules stated earlier. The weights on each edge are positive numbers. At each level of the tree, the values are distinct and have the same dimension: $\langle \rangle$ at the root; $\langle -1 \rangle$, $\langle 0 \rangle$, and $\langle 9 \rangle$ at the first level; and $\langle -1, -15 \rangle$, $\langle -1, -5 \rangle$, $\langle 0, 10 \rangle$, $\langle 9, 12 \rangle$, $\langle 9, 20 \rangle$, and $\langle 9, 32 \rangle$ at the leaves. The latter list gives all the possible values of an FRP with this Kind. Along each path from root to leaf, the values increase in dimension by 1, e.g., $\langle \rangle$, $\langle 9 \rangle$, $\langle 9, 4 \rangle$, and all such paths have the same length.

FRPs of this Kind can output the possible values $\langle -1, -15 \rangle$, $\langle -1, -5 \rangle$, $\langle 0, 10 \rangle$,

Property	FRP	Kind
dimension d	all values have dimension d	every leaf has dimension d
size s	s distinct, possible output values	s leaf nodes
type $0 \rightarrow d$	has dimension d	has dimension d
width w	–	w edges along any path from root to leaf $w = d$: fully unfolded $w = 1$: compact

TABLE 1.1. Commonly used properties of FRPs and Kinds.

$\langle 9, 12 \rangle$, $\langle 9, 20 \rangle$, or $\langle 9, 32 \rangle$. We think of the values on the leaves as being generated in stages. Starting from an empty tuple at the root, the FRP first generates one of -1, 0, or 9 and appends it to the tuple, producing one of the tuples shown at the first level. Then depending on whether -1, 0, or 9 was appended, generates another number (-5 or -15; 10; and 12 or 20 or 32 respectively) and appends it to the tuple. The tuple at each node contains the numbers generated in order along the path from the root. Remember: *at each level, the generated tuples must be distinct and the weights positive.*

Definition 1. Kinds. A **Kind** is a rooted tree where each node contains a *value* (i.e., a tuple), starting with the empty tuple $\langle \rangle$ at the root; each edge has an associated number, called its *weight*; and the tree satisfies four rules:

1. **Completeness Rule:** every path from root to leaf has the *same length*.
2. **Level Rule:** all the nodes within each level of the tree must hold *distinct* values of the *same dimension and type*.
3. **Prefix Rule:** on every path from root to leaf, each node is a *strict prefix* of every node later on the path.⁴
4. **Weight Rule:** all weights are *positive*.

FRPs and Kinds have several properties that we refer to frequently, as summarized in Table 1.1. The **dimension** is the common dimension of its possible values (FRP) or leaf nodes (Kind). The **size** is the number of possible values (FRP) or leaf nodes (Kind). The type of an FRP or Kind of dimension d is $0 \rightarrow d$; the reason for the 0 (and its generalization) will be discussed in Section 4.3. An FRP and its Kind always have the same size, dimension, and type. We call FRPs (Kinds) of dimension 1 *scalar* FRPs (Kinds).

Kinds have one additional property not shared by FRPs: the **width** of a Kind is the number of edges along any⁵ path from root to leaf. If a Kind has width w and

⁴This means that the later nodes on the path start with the components of the earlier node in order and have strictly greater dimension.

⁵Because a Kind is a complete tree, every path from root to leaf has the same number of edges.

dimension d , then w must lie in $[1 \dots d]$. If the width of a Kind equals its dimension, we say the tree is **fully unfolded**; if the width equals 1, we say the tree is **compact**.

The Kinds in Figure 1.3 have, respectively: size 1, 2, and 3. They all have width 1 and dimension 1, and thus are both fully unfolded and compact. The Kind in Figure 1.5 has dimension 2, size 6, width 2, and is thus fully unfolded. The trivial FRP, called **empty**, has dimension 0 and size 1; its Kind, denoted by $\langle \rangle$ is just a root node with an empty tuple, with dimension 0, size 1, and width 0.

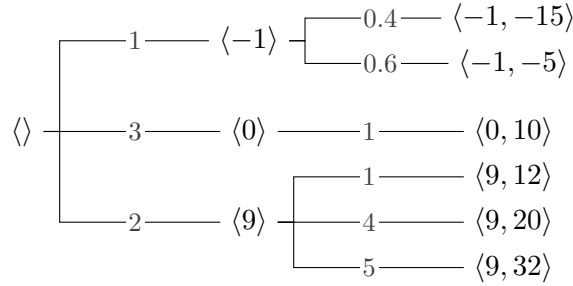


FIGURE 1.5. A Kind with more than two levels and possible values of dimension 2.

The structure of the fully unfolded Kind in Figure 1.5 gives us a picture of a random process at work “in stages”. We can think of the FRP as generating its value sequentially. Starting from the empty tuple at the root, we generate a value from the branches to the first level of the tree. Then, from the node for that value, we generate the next value in the output tuple from the branches of the subtree at that node. And so on for the entire width, which is just 2 in this case. Each stage appends a new random number to the output that is contingent on the numbers that had been generated previously. The tuple at each node shows the numbers generated so far on the path to that node, with the last element having been generated and appended at that level, eventually yielding a final value at a leaf node. Each node in the tree acts as an *address* and specifies a unique path from the root to that node. For instance, the value $\langle 9, 20 \rangle$ indicates that from the root node $\langle \rangle$, we choose the $\langle 9 \rangle$ branch; then from the $\langle 9 \rangle$ node, we choose the $\langle 9, 20 \rangle$ branch.

The Kind in Figure 1.6, has the same set of possible values as that in Figure 1.5, but it is compact. From this perspective, we can think of an FRP with this Kind as generating its value “all at once” when activated.

It turns out that these two perspectives on Kinds – describing values produced “in stages” or “all at once” – are consistent. We can view the Kind of a multi-dimensional FRP in either way. Chapter 3 will show that the Kinds in Figures 1.5 and 1.6 are in fact *equivalent*: they give the same predictions, and we can convert seamlessly

between them. Moreover, every Kind is equivalent⁶ to a Kind, called its *canonical form*, that has width 1 and has weights normalized to sum to 1. ⁶See Chapter 3

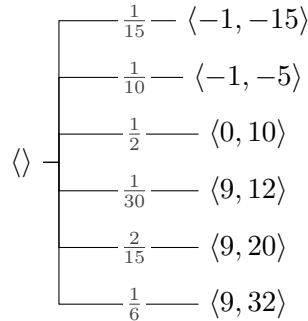


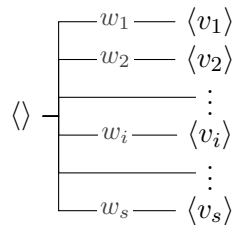
FIGURE 1.6. A Kind with dimension 2 and width 1.

Puzzle 1. Draw the Kind of an FRP of dimension 3 (and size bigger than 3) in which the values generated are all tuples whose elements sum to zero.

Puzzle 2. Draw the Kind of an FRP of dimension 2 in which knowing the second component of the value gives you no information about the first component.

Puzzle 3. Draw the a Kind of dimension 3 and width 3 (fully unfolded). Pick any subtree at a node with value v . From every node of that subtree, including the subtree, remove the components of v from that node. Is the tree that you have created a Kind? Why or why not?

A Kind gives us an association between the values on the leaf nodes and the weights along the paths from the root to each leaf. Consider first a compact (width 1) Kind of arbitrary dimension d and size s :



The information in this tree can be represented in two equivalent ways: as a table and as a function.

The table representation of a Kind, shown in Table 1.2, has a column for the value and a column for the weights. Because the values are distinct, no rows in the table are repeated. In `frplib`, we access this table for a Kind `k` with the `.weights` attribute; `k.weights` is a dictionary giving a weight for each value. For Kinds with width > 1 , each table row has a column for every value and weight along a path from root to leaf. Fortunately, Chapter 3 shows that we can express any Kind in compact form.

Value	Weight
v_1	w_1
v_2	w_2
$\vdots$	$\vdots$
v_i	w_i
$\vdots$	$\vdots$
v_s	w_s

TABLE 1.2. The table representation of a Kind in compact form.

The function representation of a Kind maps each value in $\{v_1, v_2, \dots, v_s\}$ to its corresponding weight (which are necessarily positive) and any other value to 0. In `frplib`, it looks like

```
def kind_as_function(v):
    index = index_of(v, possible_values, not_found=None)
    if index is None:
        return 0
    return weights[index]
```

where `possible_values` is the list of values $v_1, v_2, \dots, v_s$ and `weights` is the list of corresponding weights $w_1, w_2, \dots, w_s$. This function is called the **kernel** of the Kind. In mathematical work, if a Kind is already named with a symbol, we use the same symbol to denote its kernel. So for Kind k , we write writing $k(v)$ for the function evaluated at value v . If we have an FRP X (whose Kind is not named), we write k_X for the kernel of `kind(X)`, using a common base letter `k` and the name of the FRP as a subscript to *label* which kernel it is.

The kernel of a Kind can be expressed in terms of indicator functions,⁷ as shown in equation (1.1) below. In `frplib`, if `k` is a Kind, then `k.kernel` is its kernel function.

⁷See Chapter 12 in Interlude F. The braces in equation (1.1) are Iverson braces (see page 614) that give the value 1 when expression in braces is true and 0 when false.

Definition 2. The Kernel of a Kind. For any Kind, its **kernel** is the function that takes values as input and returns either (i) the canonical weight on that value, if the value is possible for the Kind, or (ii) 0, if the value is not possible.

Suppose that k is a Kind with possible values $v_1, v_2, \dots, v_s$ and corresponding canonical weights (i.e., that sum to 1) $w_1, w_2, \dots, w_s$. Then, the kernel is the function given by

$$k(v) = \sum_{i=1}^s w_i \{v_i = v\}, \quad (1.1)$$

This means that $k(v_i) = w_i$ for $i \in [1..s]$ and $k(v) = 0$ for any other value v .

For a named Kind, we use the same symbol for Kind and kernel. For an FRP, we use the generic **k** as base letter (note the typeface) with the name of the FRP as a label in the subscript, e.g., k_X is the kernel of $\text{kind}(X)$.

As an example, we will construct a Kind in **frplib** and look at these representations. One common way we build Kinds in **frplib** is with **Kind factories** – a function that builds a Kind according to a particular specification. A commonly used Kind factory is **uniform** that takes some values and builds a Kind with *equal weights* on all those values.⁸ For example:

```
pgd> k = uniform((1, 2, 3), (2, 2, 2), (3, 2, 1), (3, 3, 3))
,---- 1/4 ---- <1, 2, 3>
|---- 1/4 ---- <2, 2, 2>
<> -|
|---- 1/4 ---- <3, 2, 1>
`---- 1/4 ---- <3, 3, 3>
```

⁸Try `uniform(1, 2, ..., 6)`, the ... is explicit. Also enter `info('kind-factories')` and `info(uniform)`.

Internally, **frplib** uses a special type of tuples that have some extra features we need, but we can give regular tuples to the factory and other functions, as we do here. The associated table is

```
pgd> k.weights
{(1, 2, 3): Decimal('0.25'), (2, 2, 2): Decimal('0.25'),
 (3, 2, 1): Decimal('0.25'), (3, 3, 3): Decimal('0.25')}
```

The **Decimal** wrappers on the numbers here are due to the internal representation as high-precision decimals. The kernel function for **k** looks like

```
pgd> k.kernel((1, 2, 3))
0.25
```

```

pgd> k.kernel(1, 2, 3)      # Note: Works same without extra ()s
0.25
pgd> k.kernel(3, 3, 3)
0.25
pgd> k.kernel(9, 9, 9)
0
pgd> k.kernel((-1, 0, 1))
0

```

Notice that we can give the tuple as one argument or give the components as multiple separate arguments, interchangeably.

Given these alternative ways to capture the information in a Kind, one might wonder why we represent Kinds as trees. The reason is that trees offer us several practical advantages. First, trees are as easy to use as tables, more concrete than functions, and more definite than other representations of the same information. Second, trees are easy to visualize and keep the values (nodes) and weights (edges) visually separate. Third, trees help us reason about the structure of the Kinds. For instance, if we think about the unfolded Kind in Figure 1.7 from the “generating values in stages” perspective, then if we already know that the first component is 1, say, the subtree at $\langle 1 \rangle$ looks just like a Kind. In fact, if we remove the 1 prefix from the tuples in that subtree, it *is* the Kind representing the remaining components. Finally, and most importantly, from four simple and easy to visualize manipulations on trees (called the Big 3+1), we can derive all the calculations we do in probability theory. We will see more concrete examples of these benefits through the remainder of this chapter. But next, we turn to a fundamental question: *what does a Kind mean?*

The Kind of We often speak of Kinds associated with FRPs, saying for instance “The Kind of the FRP that represents *yada yada*.” This phrasing can get somewhat cumbersome, so we sometimes shorten it to “The Kind of *yada yada*” with the understanding that the Kind describes FRPs representing particular quantities.

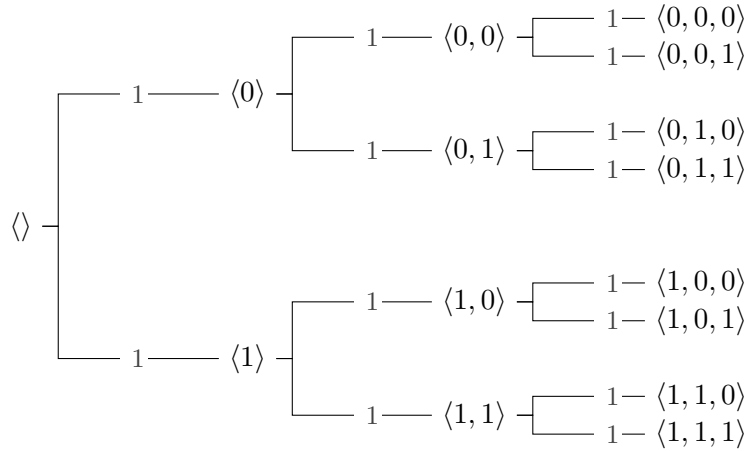


FIGURE 1.7. A fully unfolded 3-dimensional Kind with size 8.

Key Take Aways

An FRP's Kind captures all our knowledge about an unobserved FRP's value. The Kind effectively classifies an FRP into groups that “behave similarly.” All else being equal, two fresh FRPs with the same Kind are interchangeable.

A Kind is represented as a complete, rooted tree. Each node is a distinct tuple of numbers, and each edge has an associated positive number called its *weight*. We require that all nodes at the same level of a tree have the same dimension, with the dimension increasing from 0 at the root to the Kind's dimension at the leaves. Along any path from root to leaf, each pair of successive nodes share a common prefix of components; that is, we only add numbers to the tuple as we go deeper.

A Kind has several properties that we can query, including its *dimension* (the dimension of its leaf nodes), its *size* (the number of leaf nodes), its *type*, and its *width* (number of edges from root to leaf).

The *kernel* of a Kind is a function that maps values to weights, returning 0 if a value is not a leaf node of the Kind. We can construct the kernel from the Kind and vice versa.

1.3 What a Kind Tells Us

We now have some idea of what FRPs are and how Kinds are defined and represented, but we have not yet seen what Kinds *mean* and what they tell us about the value and price of an FRP of that Kind. In this Section, we will create large collections of FRPs with a common Kind to reveal the information that a Kind holds.

As a reader of this book, you have free trial⁹ access to the FRP Marketplace and thus to a seemingly inexhaustible supply of FRPs. The Marketplace manager does not like other people poking around, so you tell the manager the *Kind* and number of FRPs you want, and the manager fabricates them for you. All these FRPs are *fresh*.

It is rather tiresome to go to the Marketplace and haul back tons of boxes whenever you order some FRPs, not to mention pushing all the buttons and recording the values. Fortunately, the Marketplace is a highly automated operation, so you can manage the entire transaction with software that the Marketplace makes available. The `frp market` command opens a direct interface to the FRP Marketplace's system, allowing you to request FRPs, have their buttons pushed, and receive a record of the values from each (as a list or summary). Fast and painless for you, though a lot of work for the Marketplace staff.¹⁰ Fortunately, we can also access the Marketplace through the `frp playground` application, which is what we will use throughout Part I. See the FRP Cookbook entry “How to use `frp market`” for an `frp market` version.

To begin, we need to explore ways to construct both Kinds and FRPs. Part I builds a “pattern language” for describing and analyzing random systems. Here, we will use one basic ingredient of this language – simple Kinds constructed using [Kind factories](#). We earlier saw the `uniform` factory that takes a collection of values¹¹ and returns a Kind with equal weight on each given value.

The `constant()` Kind factory produces the simplest non-empty Kind; it takes a value and produces a Kind with only one branch for that value.

```
pgd> constant(1)
<> ----- 1 ---- 1
pgd> constant(100, -10)
<> ----- 1 ---- <100,-10>
pgd> constant(3, 4, 5)
<> ----- 1 ---- <3, 4, 5>
```

and similarly for any specified value. Notice that we can supply a value either with a single explicit tuple, like `constant( (3, 4, 5) )` or with the components of that

⁹As this is only a trial, you receive no actual payoff in exchange for the FRPs whose buttons you push. Later, when the trial expires, money will change hands, so the stakes will be higher.

¹⁰Many Bothans worked hard to bring you this information.

¹¹The values can be passed collected in a single argument, like a list, or as separate values. See `info('uniform')` for details and examples.

tuple as separate arguments. For any Kind `k` in `frplib`, we can use `frp(k)` to construct a fresh FRP of that Kind. For example,

```
pgd> frp(constant(2, 4, 8, 16, 32))
An FRP with value <2, 4, 8, 16, 32>
```

An FRP of Kind `constant(v)` is called a **constant FRP** with value v , because it always outputs the value v . For any weight w on the single branch of the Kind, a constant FRP with value v can be completely identified with the constant v itself. Whether I gave you the value v or an FRP that produces that value, you would be indifferent. For all practical purposes, they are the same. For reasons that we will see, we default to a weight of 1 on the Kind `constant(v)`.

But rather than look at one FRP at a time, we will study the values of many FRPs of the same Kind in aggregate. In the FRP Marketplace, a “demo” is a request for a large collection of FRPs of the same Kind. In the `frplib` playground, we use the `Market.demo` command¹² to make request a demo from the Marketplace. (Keep in mind that as you follow along, you can expect to get somewhat different numbers because of the randomness inherent in the FRPs we build.)

¹²The `FRP.sample` function is identical, except it neither prints the Kind nor makes an official demo request to the Marketplace.

Let’s try it with a `constant` Kind:

```
pgd> Market.demo(1000, constant(1))
Activated 1000 FRPs with kind
<> ----- 1 ---- 1
```

Summary of Output Values

+	-----	+	-----	+	-----	+
	Values		Count		Proportion	
+	=====	+	=====	+	=====	+
	1		1000		100%	
+	-----	+	-----	+	-----	+

```
pgd> Market.demo(100_000, constant(3, 4, 5))
Activated 100000 FRPs with kind
```

```
<> ----- 1 ---- <3, 4, 5>
```

Summary of Output Values

+	-----	+	-----	+	-----	+

Values	Count	Proportion
<3, 4, 5>	100000	100%

This causes the specified number of fresh FRPs of the specified Kind to be observed. It shows the Kind and summarizes the values observed, their counts, and their proportions. An FRP with a **constant** (size 1) Kind can produce only one value, and that is all we see in the tables. Notice that one can use `_` as a digit separator to make numbers more readable, e.g., `100_000` for 100,000. In the remainder of this section, the Kind and table title will be omitted for space when they are clear from context.

Puzzle 4. In the Kinds of size 1 above, the weights on the single branch is 1. How would the results differ with a different positive weight? Remember, the demo builds a collection of fresh FRPs with that Kind, so what values will we see in the table?

After the empty Kind and the **constant** Kinds, the next most complicated Kind has two branches. We can make such Kinds with the **choice** Kind factory: `choice(a, b, r)` creates a Kind with values *a* and *b* where *r* > 0 is the ratio of weights *b*-to-*a*.

```
pgd> choice(-1, 1, 1/3)
```

```

,---- 3/4 ---- -1
<> -|
`---- 1/4 ---- 1
```

```
pgd> k = choice(-1, 1, 1/3)
```

```
pgd> Market.demo(100, k)
```

```
Activated 100 FRPs with kind
```

```

,---- 3/4 ---- -1
<> -|
`---- 1/4 ---- 1
```

```
Summary of Output Values
```

```
+-----+-----+-----+
```

Values	Count	Proportion
-1	71	71%
1	29	29%

Notice that we can bind Kinds to variables (here `k`) like any other object. The demo of 100 FRPs with Kind `k` gives values in proportion suspiciously close to the weights from the Kind. And if we do a demo of the same size again, we see some variation in the results.

```
pgd> Market.demo(100, k)
```

Values	Count	Proportion
-1	70	70%
1	30	30%

```
pgd> Market.demo(100, k)
```

Values	Count	Proportion
-1	78	78%
1	22	22%

Try this with an even smaller sample of FRPs, and we see even more variation.

```
pgd> Market.demo(3, k)
```

Values	Count	Proportion
-1	3	100%

```
pgd> Market.demo(3, k)
```

Values	Count	Proportion
-1	1	33.33%

```

| 1      | 2 | 66.67% |
+-----+-----+-----+
pgd> Market.demo(3, k)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 2    | 66.67%    |
| 1      | 1    | 33.33%    |
+-----+-----+-----+

```

Each FRP in the batch produces its value at random, so in a batch of size 3, we might see all -1s or all 1s or anything in between. We call this **sampling variation**, and we can see that the variation is noticeably smaller with a larger demo size of 100. Let's increase the demo size further

```

pgd> Market.demo(1000, k)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 744   | 74.4%      |
| 1      | 256   | 25.6%      |
+-----+-----+-----+
pgd> Market.demo(1000, k)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 748   | 74.8%      |
| 1      | 252   | 25.2%      |
+-----+-----+-----+
pgd> Market.demo(10_000, k)      # Can use _ for "," to make #s readable
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 7539  | 75.39%     |
| 1      | 2461  | 24.61%     |
+-----+-----+-----+
pgd> Market.demo(10_000, k)
+-----+-----+-----+

```

```

| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 7453  | 74.53%    |
| 1      | 2547  | 25.47%    |
+-----+-----+-----+
pgd> Market.demo(1_000_000, k)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 750603 | 75.06%    |
| 1      | 249397 | 24.94%    |
+-----+-----+-----+
pgd> Market.demo(1_000_000, k)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 750949 | 75.09%    |
| 1      | 249051 | 24.91%    |
+-----+-----+-----+

```

From these results, we glean two ideas. First, as the number of FRPs in a demo grows, the sampling variation in the results gets smaller, though somewhat slowly. You can try repeating these commands to see this even more clearly. Second, as the number of FRPs in a demo grows, the proportions of each value get closer to the weights on the corresponding branches of the Kind, in this case $3/4$ and $1/4$.

Puzzle 5. Do repeated demos for a Kind of size 2 for each batch size in 1, 10, 10,000, and 1,000,000 (and if you are ambitious 100,000,000). Characterize roughly how the sampling variation in your results depends on the batch size. Use a measure of sampling variation that makes sense for you. (I recommend the “intra-ocular” definition – what hits you between the eyes?)

What happens if we change the weights? To answer this question, we will use the `weighted_as` Kind factory. It takes values (like `uniform` does) and an extra argument `weights` that gives the weights for the corresponding values.¹³ For example:

```

pgd> weighted_as(1, 2, ..., 6, weights=[6/21, 5/21, ..., 1/21])
,---- 6/21 ---- 1

```

¹³If the `weights` argument is missing, the weights are all equal, like `uniform`.

```

|---- 5/21 ---- 2
|---- 4/21 ---- 3
<> -|
|---- 3/21 ---- 4
|---- 2/21 ---- 5
`---- 1/21 ---- 6

```

Here, the values are specified as individual arguments, the “...” tells the factory to follow the pattern determined by the gap between the preceding two values, up until the next specified value. The weights can be given as numbers, symbols (which we will see later), or strings that specify numbers for higher precision (e.g., '1/6'). The factory `weighted_pairs` is similar, except values and weights are specified in pairs, e.g., `weighted_pairs((0, 3), (1, 27))` instead of `weighted_as(0, 1, weights=[3, 27])`. The factory `weighted_by` is also similar, except weights are determined by a given function of the values.

Puzzle 6. Indicate what sequences of values are produced by each of the following specifications:

- 1/99, 2/99, ..., 1
- 10, 20, ..., 100, 200, ..., 1000
- 7, 9, 13, ..., 43
- 3, 11, ..., 99
- 3, 11, ..., 100
- 5, 10, ..., 25, 50, ..., 100, 101, ..., 111

Try to reason out what you expect. The trickiest part is what happens at the end when it does not fit exactly into the pattern.

To test these in `frplib`, do

```
from frplib.examples.hello_world import patterned_sequence
```

and pass these sequences (including ...s) to that function.

Now, consider demos for Kinds with the same two values but different weights:

```
pgd> Market.demo(10_000, weighted_as(-1, 1, weights=[1, 1]))
```

Activated 10000 FRPs with kind

```
,---- 1 ---- -1
<> -|
    `---- 1 ---- 1
```

Summary of Output Values

Values	Count	Proportion
-1	4976	49.76%
1	5024	50.24%

```
pgd> Market.demo(10_000, weighted_as(-1, 1, weights=[3, 1]))
```

Activated 10000 FRPs with kind

```
,---- 3 ---- -1
<> -|
    `---- 1 ---- 1
```

Summary of Output Values

Values	Count	Proportion
-1	7501	75.01%
1	2499	24.99%

```
pgd> Market.demo(10_000, weighted_as(-1, 1, weights=[49, 1]))
```

Activated 10000 FRPs with kind

```
,---- 49 --- -1
<> -|
    `---- 1 ---- 1
```

Summary of Output Values

Values	Count	Proportion

```

+=====+=====+=====+
| -1      | 9790 |      97.9% |
| 1       | 210  |      2.1% |
+-----+-----+-----+
pgd> Market.demo(10_000, weighted_as(-1, 1, weights=[999, 1]))
Activated 10000 FRPs with kind

      ,---- 999 -- -1
    <> -|
      `---- 1 ---- 1

Summary of Output Values
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 9990  |      99.9% |
| 1      | 10    |      0.1% |
+-----+-----+-----+

```

What have you learned from these results? Try the following puzzle.

Puzzle 7. Create a series of similar demos for a wide variety of different weights and study the results. Use these and the earlier demos to formulate a hypothesis about the relationship between the weights on a Kind of size 2 and the (approximate) proportions of the values you will see in a large demo of that Kind.

Notice that in each case, the values appear in proportions very similar to the relative sizes of the weights. When the weights are 1 and 1, the proportions are close to 50-50. When the weights are 3 and 1, the proportions are close to 75-25. And so on. The proportions are not exact, just *close*, and we might suspect that the reason for this is the *sampling variation* that we saw earlier. Try two experiments to confirm this: (i) pick one or more of the demos above, and repeat it multiple times; (ii) pick one or more of the demos above, and repeat it with a larger and a smaller batch size. In (i), sampling variation should show proportions that scatter randomly around the proportions we expect. In (ii), the magnitude of sampling variation – the dispersion around those expected proportions – should get smaller overall as the batch size increases. Do your findings confirm these points?

We can take from this a hypothesis, that for a Kind

$$\langle \rangle \begin{cases} w_a & \langle a \rangle \\ w_b & \langle b \rangle \end{cases}$$

the proportions of values a and b in a large demo will be approximately $\frac{w_a}{w_a+w_b}$ and $\frac{w_b}{w_a+w_b}$, respectively. Thus, *the relative sizes of the weights correspond approximately to the relative propensities of the corresponding values in a demo, and the approximation gets better and better as our batch size increases.* As we increase the batch size of the demo, the sampling variability gets smaller, and we approach the idealized proportions of an infinite batch. We can thus view the weights as determining the idealized proportions of values in a demo of infinite batch size.

There is a geometric view of this conclusion that may be helpful, illustrated in Figure 1.8.¹⁴

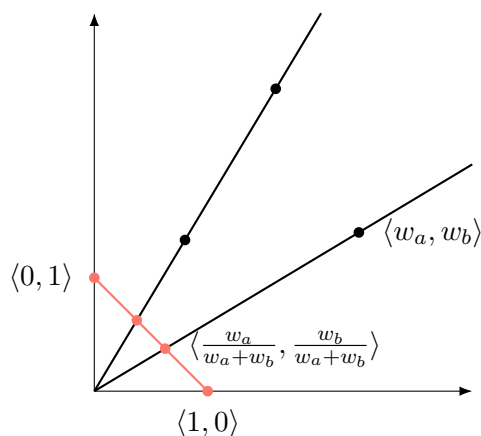
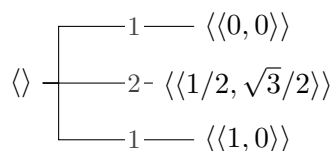


FIGURE 1.8. Each point in the positive-positive quadrant of the plane represents a choice of weights for a Kind of size 2. Every choice of weights on the same *line from the origin* yields the same proportion in a large demo of FRPs. For example, the two marked points on the upper line give the same proportions. The proportions are the coordinates of each line's intersection with the (red) diagonal line and thus also represent a choice of weights, normalized to sum to 1. We call these the *canonical weights* for the Kind.

Now, let us look at Kinds with more structure, restricting our attention for the moment to Kinds of width 1. For a variety of Kinds of size 3 and width 1, run a range of demos, and using our experience with Kinds of size 2, try to formulate a hypothesis about the relationship between the Kind's weights and the proportions of each value in a very large demo. Is there a geometric interpretation of your hypothesis akin to Figure 1.8? Write down your hypothesis explicitly in a sentence or two.

As one example, using the Kind

¹⁴Example 13.9 and the discussion around equation (13.15) in Interlude F are relevant here.



```
pgd> Market.demo(100_000, weighted_as((0, 0), (1/2, sqrt(3)/2), (1, 0),
...>                                     weights=[1, 2, 1])
```

```
Activated 100000 FRPs with kind
```

```
,---- 1/4 ---- <0/2, 0>
<> -+---- 2/4 ---- <1/2, 0.86603>
`---- 1/4 ---- <2/2, 0>
```

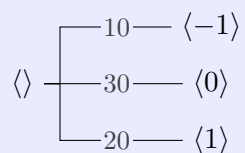
Summary of Output Values

Values	Count	Proportion
<0, 0>	24958	24.96%
<1/2, 0.866>	49899	49.9%
<1, 0>	25143	25.14%

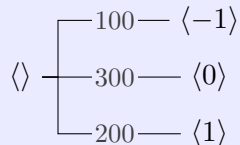
Repeating this demo multiple times will give similar proportions. Notice that the dimension of the Kind is irrelevant to the question of how the weights relate to the observed proportions.

The hypothesis you formulated in the previous puzzle generalizes. Again, for the moment, we consider only Kinds of width 1; we will see more about the width > 1 case later.

Puzzle 8. If I give you a choice between the actual payoffs of two FRPs of respective Kinds



and



do you have a preference between them? For which? Why? You can use the `frplib` playground to support your choice.

If we run a demo of n FRPs of size s and width 1 whose Kind has weights $w_1, w_2, \dots, w_s$ and values $v_1, v_2, \dots, v_s$, we will get a table of associated proportions $p_1, p_2, \dots, p_s$ that sum to 1. Across repeated runs of this `demo`, these proportions will vary slightly because of sampling variability (i.e., different FRPs are being observed), even though the FRPs have the same Kind. As we make n larger, this sampling variability gets ever smaller, and the proportions p_i get closer to “ideal” proportions $\bar{p}_i$ that are determined by the Kind’s weights. In fact,

$$\bar{p}_i = \frac{w_i}{w_1 + w_2 + \dots + w_s}, \quad (1.2)$$

or put another way, $\bar{p}_i/\bar{p}_j = w_i/w_j$. The idealized proportions we see in the table are thus determined by *the weights normalized to sum to 1*, and the relative proportions of any two values are determined by the *relative magnitudes of the corresponding weights*. We call these the *canonical weights* of the Kind. A Kind’s weights thus tells us the **relative propensity to appear of each of the FRP’s possible values**

We can leverage this insight to understand how to assign a price to the payoff represented by an FRP. The FRP Marketplace’s `buy` command allows one to purchase FRPs of specified Kinds and numbers at specified prices. It is like `demo` except it also computes the *net payoff* – the difference between the total payoff from all the purchased FRPs and the total payment we made for those FRPs – and the net payoff per FRP purchased for the entire demo. For example:

```
pgd> Market.buy(1_000_000, [-1, -0.5, 0.5, 1, 2, 4],
...>           weighted_as(-5, 0, 4, weights=[2, 3, 5]))
Buying 1,000,000 FRPs with kind
      ,---- 0.2 ---- -5
      <> -+---- 0.3 ---- 0
      `---- 0.5 ---- 4
at each price

+-----+-----+-----+-----+
```

Price/Unit (\$)	Net Payoff (\$)	Net Payoff/Unit (\$)
-1	2001045	2.0010
-0.5	1503463	1.5035
0.5	501775	0.50178
1	4	0.000004
2	-1004968	-1.0050
4	-2996398	-2.9964

We specify the batch size, a list of prices, and the Kind of the FRPs to buy.¹⁵ Unsurprisingly, as the price we pay per unit increases, our net payoff decreases. As each batch for which these payoffs are measured is an arbitrary sample of FRPs, the payoffs we record in this demo are subject to sampling variability. If we re-run the buy command, there will be random variation in the payoffs, and the magnitude of the sampling variability will be smaller the bigger the batch size.

¹⁵ A negative price means we receive that amount in payment.

The net payoffs range from around 2 to around -3, so there is a price per unit where the net payoff per unit will equal zero, excluding sampling variability. The table strongly suggests that that price is 1, or very close to it. And indeed, notice that the net payoff per unit is approximately $1 - p$, where p is the price per unit paid.

Suppose we run `buy` with price c for each of n FRPs of size s whose Kind of width 1 has weights $w_1, w_2, \dots, w_s$ and values $v_1, v_2, \dots, v_s$. The underlying demo has proportions $p_1, p_2, \dots, p_s$ for those values. Then our net payoff per unit is

$$\frac{np_1v_1 + np_2v_2 + \dots + np_sv_s - nc}{n} = p_1v_1 + p_2v_2 + \dots + p_sv_s - c.$$

For large n , the p_i 's will be close to the “ideal” proportions

$$\bar{p}_i = \frac{w_i}{w_1 + w_2 + \dots + w_s}$$

derived from the weights. When we choose $c = \sum_{i=1}^s \bar{p}_i v_i$ as the price per FRP, the net payoff will be close to 0 for a large enough n . This special price c is called the *risk-neutral price* of FRPs with that Kind because it is the price at which we are indifferent between having the batch of FRPs or the money, provided we are inured to the risk of borrowing enough money to buy a sufficiently large batch. We will discuss risk-neutral prices in detail in Chapter 6 and see their relation to a property of a Kind (or an FRP) called its **expectation**, which is our best prediction of an FRP's value.

So far in this Section, we have restricted our attention to Kinds of width and dimension 1 because these are simpler. Changing the dimension does not really affect our conclusions as the proportions are associated with values, whatever they are.

```
pgd> Market.demo(1_000_000, choice((1, 2, 3), (9, 11, 16)))
```

Activated 1000000 FRPs with kind

```
,----- 1 ---- <1, 2, 3>
<> -|
      ^----- 1 ---- <9, 11, 16>
```

Summary of Output Values

-----+-----+-----
Values Count Proportion
-----+-----+-----
<1, 2, 3> 499893 49.99%
<9, 11, 16> 500107 50.01%
-----+-----+-----

And similarly with the Kind from Figure 1.6,

```
pgd> Market.demo(
...>   1_000_000,
...>   weighted_as( (-1, -15), (-1, -5), (0, 10),
...>                  (9, 12), (9, 20), (9, 32),
...>                  weights=['1/15', '1/10', '1/2',
...>                  '1/30', '2/15', '1/6'])
...> )
```

Activated 1000000 FRPs with kind

```
,----- 2/30 ----- <-1, -15>
|----- 3/30 ----- <-1, -5>
|----- 15/30 ----- <0, 10>
<> -|
      |----- 1/30 ----- <9, 12>
      |----- 4/30 ----- <9, 20>
      ^----- 5/30 ----- <9, 32>
```

Summary of Output Values

-----+-----+-----
Values Count Proportion
-----+-----+-----
<-1, -15> 66903 6.69%

<-1, -5>	99555	9.96%	
<0, 10>	500256	50.03%	
<9, 12>	33545	3.55%	
<9, 20>	133086	13.31%	
<9, 32>	166655	16.67%	
-----+-----+-----			

The market tries to show the weighs with a common denominator to make comparison easier, when the denominators are not too large.

For a Kind of width > 1 , the relationship between the weights and the proportions in the demo tables requires some clarification. Using the Kind in Figure 1.5,

Activated 1000000 FRPs with kind

			,----- 2/5 ---- <-1, -15>
	,----- 1 ---- <-1> -		
		`----- 3/5 ---- <-1, -5>	
<> -+----- 3 ---- <0> -+----- 1 ----- <0, 10>			
		,----- 1 ----- <9, 12>	
`----- 2 ---- <9> -+----- 4 ----- <9, 20>			
		`----- 5 ----- <9, 32>	

Summary of Output Values

-----+-----+-----			
Values	Count	Proportion	
-----+-----+-----			
<-1, -15>	67182	6.72%	
<-1, -5>	100190	10.02%	
<0, 10>	499985	50.00%	
<9, 12>	33438	3.34%	
<9, 20>	133213	13.32%	
<9, 32>	165992	16.60%	
-----+-----+-----			

we see that these proportions are essentially the same as what we got above for the Kind in Figure 1.6, consistent with our claimed equivalence of the two Kinds. We will explore this connection further in Chapter 3.

Our work in this Section provides support for the following claim.¹⁶

¹⁶The *increment* $[1..s]$ is the set of integers from 1 to s , as described in Section 9.2.

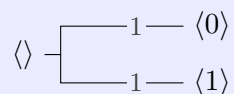
Let k be a Kind of width 1 with weights $w_1, w_2, \dots, w_s$ and corresponding values $v_1, v_2, \dots, v_s$, and let k' be the Kind (of width 1) with the same values and corresponding weights $w'_1, w'_2, \dots, w'_s$ where

$$w'_i = cw_i, \quad (1.3)$$

for all $i \in [1 \dots s]$ and some $c > 0$. Then, we cannot distinguish a demo of n FRPs with Kind k from a demo of n FRPs with Kind k' .

This claim means that two Kinds whose weights differ by a constant multiplicative factor are in practice interchangeable. Chapter 3 will define a formal notion of Kind equivalence along these lines.

Puzzle 9. We have seen empirically that a demo of n FRPs with Kind



will produce roughly even proportion of 0's and 1's. Let $p_1, p_2, \dots, p_k$ be the proportion of 1s in k repeated demos of n FRPs with this Kind. These are all *approximately* 50%, but they vary around it by a bit from the randomness in the FRPs. How does the variation around 50% depend on n ?

To investigate this, start with several demos of 100 FRPs of this Kind. About how close are is the frequency of 1's to 50%? Look at this as you increase n several times. How big does n need to be so that the frequencies are within about one decimal point of 50%? Within two decimal points? Within three? What can you conclude?

Key Take Aways

A *demo* of FRPs is a sample of fresh FRPs with the same Kind. The FRP Marketplace pushes the buttons on all these FRPs and provides us with a summary “demo table” of the values we observed and their frequencies.

By studying large demos of various Kinds, we see that the relative proportions of values in the demo tables are close to the relative sizes of the weights in the Kind. The deviation of the table from the Kind is due to *sampling variability*, the random quirks of the particular sample of FRPs we obtained in the demo.

As the size of the demo grows larger, sampling variability reduces.

This gives us one interpretation of a Kind: the relative size of weights in the Kind tell us the relative propensity for an FRP with that Kind to produce that value. We also see that multiple Kinds can be *equivalent* in that they produce interchangeable demo tables. We will explore this further in Chapter 3.

1.4 FRPs as Data, Kinds as Models

Having earlier seen what FRPs are and gotten some sense of what their Kinds mean, this section describes how we represent random systems using FRPs and how we model and analyze random systems using the Kinds of those FRPs. We build a framework that applies consistently across systems of widely varying complexity and scope, and we will develop several simple examples to illustrate how it is used.

A **random system** is any process or scenario where we can (eventually) observe information about the system's output and state that may be uncertain. The uncertain information may be observable all at once, or available across some range of space and time. We might watch the system evolve – or **run** – measuring with notebook in hand, or we might merely consider a hypothetical run of the system. A flipped coin and the returns over time on a particular stock are both random systems. But so is the amount of money in a box that we *know* contains exactly \$1; a **deterministic** system is a special, trivial case of a random system with no uncertainty. We call the information that we observe (in reality or hypothetically) from a random system **data**. In practice, we collect data from a random system to **answer questions**, questions about how to act, about how things work, and about particular features derived from the data.

Probability theory is a tool for making predictions about that data obtained from a random system. A fundamental conceit of the theory is that we make predictions about a random quantity *before* we observe that quantity. After all, once the quantity has been observed, it is fully known, and there is no further need to predict its value. Calculations in probability theory are thus expressed in terms of the data from a random system and of features derived from those data *without the values of these quantities being known*.

This is the role of FRPs in our framework. FRPs represent random quantities – data – that can be observed/measured during the evolution of a random system. A fresh FRP represents a meaningful random quantity whose value *has not yet*

This paragraph is worth returning to multiple times as you digest the material in Part I

been determined. An activated FRP represents a random quantity whose value has been determined but *that has not yet been observed.* An observed FRP represents a random quantity whose *value we know.* Pushing the button on an FRP corresponds to observing the corresponding quantity. To specify a random system, we build a special FRP whose value represents all the relevant data that will be obtained from that system, which we call the “Data FRP”. This gives us the basic data that pertains to our goals and that we can use to answer questions that drive our analysis of the system. The answers to our questions are expressed in terms of features that are derived from the data, and each such answer is also represented as an FRP, which we call “Feature FRPs”. Predicting the values of these “Feature FRPs” gives us predicted answers to our questions. A schematic of these relationships is illustrated in Figure 1.9. We will elaborate this framework throughout Part I as we learn how to build the Data FRP and feature FRPs through various operations on simpler FRPs and how to handle partial observations of the data.

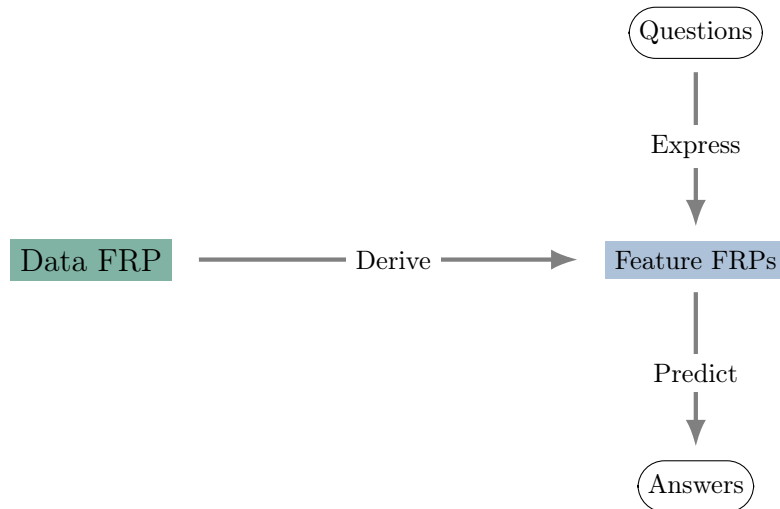


FIGURE 1.9. A simplified framework for how FRPs are used. The “Questions” node represents the various questions or goals that motivate our analysis and data collection. The value of the “Data FRP” represents all the data we measure or observe from a random system. By processing the data, we *derive* various features of the data that express answers to our questions. As random/uncertain quantities, these are also FRPs, and thus called “Feature FRPs”. Predicting the values of these FRPs gives us predicted “Answers” to our questions, which we can use for making decisions or drawing conclusions from the data.

Describing a random system with FRPs will reveal the structure of the system and relationships among its parts. One advantage of the physical model for systems that FRPs gives us is that the operations for building systems have a tangible, physical

interpretation.

FRPs are a useful tool for thinking about the data and the questions we are asking. But because our analysis is done *before* observing the full data – before all these FRPs are activated – it is the *Kinds* of the FRPs that are most directly useful for our analysis. Recall that the Kind of an FRP embodies everything we know about its value. So the Kind of the Data FRP incorporates our knowledge and assumptions about the random system itself. We thus *model* the system by positing a specific form of this Kind. An important aspect of the operations we use to build an FRP for the data from simpler pieces is that all these operations also apply to Kinds in a consistent way. So our random system can be described in a language of operations on FRPs and our *analysis* of the system can be described in a parallel language of those same operations on the corresponding Kinds.

But this is all a bit abstract, so let's make it concrete with a few simple examples.

We start with one of the applications that historically inspired people to think about probability – dice games. Suppose we have three balanced, six-sided dice   , one red, one green, and one blue. We shake the dice together thoroughly and roll. The data we observe from this process are the numbers on the top face of the three dice, where we distinguish the individual dice. The FRP representing these data has size 216, with one possible value for each possible pattern of three dice. Because the dice are balanced, shaken thoroughly, and rolled without favor, the Kind of this FRP looks like that shown in Figure 1.10.

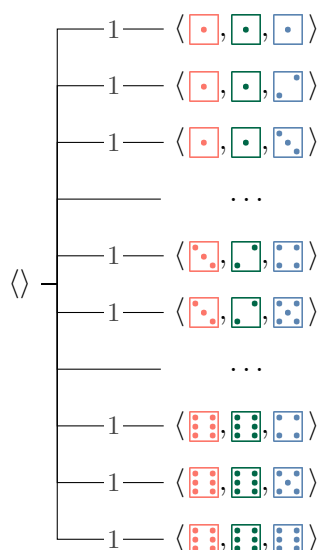


FIGURE 1.10. The Kind of the FRP representing a roll of three, distinct, balanced dice.

This paragraph is worth returning to multiple times as you digest the material in Part I

We can see directly that the value of the FRP is composed from the values of three FRPs, one for each of the three dice. Each Element FRPs has Kind as shown in Figure 1.11.

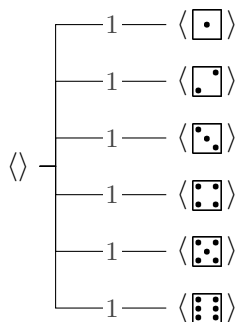


FIGURE 1.11. The Kind of the FRP representing single roll of a balanced six-sided die.

We encode the values of these FRPs with numbers rather than pictures. Here, the mapping to numbers is direct – the number showing up on the dice. The data then are three-dimensional tuples like $\langle 1, 1, 1 \rangle$ for $\langle \text{red}, \text{green}, \text{blue} \rangle$ and $\langle 3, 2, 4 \rangle$ for $\langle \text{red}, \text{green}, \text{blue} \rangle$. We list the dice in specific order (red, green, blue), so there is no need to further encode the colors into the values.

We will analyze this example in the `frplib` playground. Enter the playground by invoking `frp playground` at the terminal prompt, or one of the alternative commands described in the documentation. Follow along with the commands listed here, examining the output.¹⁷

We start by constructing and viewing the Kind of a single roll, the elements from which the data are built. As we did earlier, we will use the *Kind factory*¹⁸ `uniform` to create a Kind with equal weights over specified values. Here, the function `uniform` is called a **Kind factory**, which takes as input a specification of a Kind and returns the Kind itself. The `uniform` factory always returns Kinds with equal weights on all its values, so we only need to specify the sequence of values.

```
pgd> single_roll_kind = uniform(1, 2, ..., 6)
pgd> single_roll_kind    # Will print this Kind, output omitted
pgd> unnormalized(single_roll_kind)
```

This will show the Kind in Figure 1.11, both normalized and unnormalized. The *pattern* `1, 2, ..., 6` in the call to `uniform`, with an explicit `...`, expands to the values 1, 2, 3, 4, 5, 6.

Next, we construct the FRPs that represent the three individual dice rolls. Since the three FRPs have the same Kind, we can use that Kind to build them.

¹⁷When showing playground input, text from # to the end of a line is a comment for your benefit. You should not type or enter that. Playground output is sometimes omitted after commands to indicate that you should try the command yourself.

¹⁸Many other Kind factories are available. The `frplib` Cheatsheet is a helpful resource as is the built-in documentation.

```
pgd> R = frp(single_roll_kind)
pgd> G = frp(single_roll_kind)
pgd> B = frp(single_roll_kind)
```

The `frp` function takes a specification, including a Kind, and creates a *fresh* FRP that meets that specification. The FRP will be observed once you examine its value. Even though R, G, and B have the same Kind, they are distinct FRPs with their own random values.

Finally, we combine R, G, and B into a new FRP `Roll` that represents our data. To do this, we use an operation that we will call the **independent join**¹⁹ of the three FRPs. In `frplib`, this operation is denoted with the `*` operator, so we have

```
pgd> Roll = R * G * B
```

Do not yet look at the value of `Roll`, we will keep it fresh for the moment. Examine the Kind of this FRP with

```
pgd> kind(Roll)
```

and you will see that it is just the Kind in Figure 1.10, with the dice rolls encoded as numbers. Note that the independent join operator `*` is *not* multiplication. It is a special operation on FRPs and Kinds, that acts like product in many ways.

Suppose we want to predict answers to the questions: (i) what is the sum of the numbers on the three dice, and (ii) if we see a sum bigger than 10, what is the minimum of the numbers on the three dice? Notice that the second question is *conditional* in that it asks about a feature of the data only when the particular condition is true that the sum of dice is bigger than 10. We derive the Feature FRPs representing answers to these questions from the Data FRP *before we see the value of the data*, so our predictions of these values can guide our actions and decisions.

We construct these Feature FRPs by transforming and constraining the FRP `Roll`. For transforming FRPs we use *statistics*, which as mentioned earlier are functions that take values (tuples of numbers) as input and return to values (tuples of numbers) as output.²⁰ `frplib` pre-defines a variety of useful statistics, which typically have names that start with a capital letter. In particular, `Sum` and `Min` return the scalars (i.e., one-dimensional tuples) that compute the sum and minimum of their input tuples:

```
pgd> Sum(1, 2, 3)
<6>
pgd> Min(4, 7, 2)
<2>
```

¹⁹ Joins are discussed in detail in Chapter 4.

²⁰ Transformation will be discussed in detail in Chapter 2.

Transforming the FRP `Roll` with the statistic `Sum` gives a new *Feature* FRP whose value is the sum of the components in `Roll`'s value. This is what we need for prediction target (i). In the playground, we write this as

```
pgd> DiceSum = Roll ~ Sum
```

where we think of the transformation operator `~` as an arrow connoting that the value of `Roll` is passed through the statistic `Sum` to produce the new FRP `DiceSum`.

Our predictions of the sum of the dice are determined by `kind(DiceSum)`, and we compute our prediction of its value – its “risk-neutral price” or **expectation**²¹ – with the `E` operator:

²¹Both terms are discussed in detail in Chapter 6.

```
pgd> kind(DiceSum)
,---- 0.0046296 ---- 3
|---- 0.013889 ----- 4
|---- 0.027778 ----- 5
|---- 0.046296 ----- 6
|---- 0.069444 ----- 7
|---- 0.097222 ----- 8
|---- 0.11574 ----- 9
|---- 0.12500 ----- 10
<> -|
|---- 0.12500 ----- 11
|---- 0.11574 ----- 12
|---- 0.097222 ----- 13
|---- 0.069444 ----- 14
|---- 0.046296 ----- 15
|---- 0.027778 ----- 16
|---- 0.013889 ----- 17
~---- 0.0046296 ---- 18

pgd> E(DiceSum)
21/2
```

We can see from the Kind that the weights are *symmetric* around the midpoint of the possible values, and the expectation is equal to that midpoint: 10.5.

For prediction target (ii), we want to compute the minimum of `Roll`'s components but only on the condition that `DiceSum` is bigger than 10. For this, we use an **observation**, a constraint on the system based on a real or hypothetical observation of some feature of the data before the full data are seen..²² In `frplib`, we specify an

²²Observations are discussed in detail in Chapter 5.

observation with the `|` operator, which is always read as “given.” We define a new FRP derived from `Roll` by applying a constraint:

```
pgd> Roll | (Sum > 10)
```

To the right of the `|`, `(Sum > 10)` is the observation; it means that the statistic `Sum` applied to the value of the FRP to the left of the `|` operator is bigger than 10. Note that the outer parentheses are needed here because of Python’s rules for operator precedence. The observation constrains the sum of `Roll`’s components to be greater than 10.

The new FRP defined by an observation is only meaningful when the constraint is true. What you see when you enter this will depend on whether your copy of `Roll` has value with sum bigger than 10. If it does, you will see the same value as that of `Roll`; if not, you will see the empty FRP. Try repeating `clone(Roll) | (Sum(__) > 10)` until you see both possibilities.²³ Now, for `Roll` – or a clone with sum bigger than 10 – enter

```
pgd> kind(Roll | (Sum(__) > 10))
```

You should see a Kind with 108 branches, all weights equal, and all its values satisfying the observation’s constraint that their components sum to more than 10. Our feature FRP for question (ii) is a transform of this constrained FRP using the statistic `Min`.

```
pgd> M10 = (Roll | (Sum(__) > 10)) ^ Min
pgd> kind(M10)          # You should look at this
pgd> E(M10)
2.722
```

`M10` is obtained by first constraining `Roll` with the observation and then transforming the constrained FRP with a statistic. (Again, for the purpose of this example, it is more interesting to use a clone of `Roll` for which the observation is true, so repeat the definitions of `R`, `G`, `B`, `Roll`, and `DiceSum` until `kind(M10)` is not the empty Kind.) Look at `kind(M10)`. Try comparing it to `(Roll | (Sum(__) > 16)) ^ Min` and `(Roll | (Sum(__) > 5)) ^ Min`.

Now we can look at the values of the FRPs themselves. Because `DiceSum` is derived from `Roll` and `Roll` is derived from `R`, `G`, and `B`, looking at the value of `DiceSum` requires the values of the other FRPs. So once you examine `DiceSum`’s value, `Roll`, `R`, `G`, and `B` are no longer fresh. Look at their values and confirm that their sum equals the value of `DiceSum`.

²³The `clone` function creates a fresh FRP with the same Kind as the FRP given to it.

```

pgd> DiceSum
An FRP with value <16>
pgd> Roll
An FRP with value <4, 6, 6>
pgd> R
An FRP with value <4>
pgd> G
An FRP with value <6>
pgd> B
An FRP with value <6>

```

These are the values for the FRP that I drew from the Marketplace; your values will likely be different.

If we want to roll the dice again, we need a new FRP with the same Kind. For this, we can use the `clone` function to request a new FRP from the Marketplace.

```

pgd> clone(Roll)
An FRP with value <6, 3, 2>
pgd> clone(DiceSum)
An FRP with value <14>
pgd> Roll2 = clone(Roll)
pgd> DiceSum2 = Roll2 ^ Sum

```

Notice that the first two clones are independent of each other; there is no relation, for instance, between the values of `Roll`'s clone and `DiceSum`'s clone. However, `Roll2` and `DiceSum2` *are* related because we derived `DiceSum2` from `Roll2` by the appropriate transformation.

We can draw from the Marketplace a large batch of new FRPs of the same Kind as any FRP using either the `Market.demo` or `FRP.sample` functions.

```

pgd> Frp.demo(10000, DiceSum)
|-----+-----+-----|
| Values | Count | Proportion |
|=====+=====+=====|
| 3      | 41    | 0.41%      |
| 4      | 135   | 1.35%      |
| 5      | 285   | 2.85%      |
| 6      | 431   | 4.31%      |
| 7      | 661   | 6.61%      |

```

8	938	9.38%	
9	1198	11.98%	
10	1265	12.65%	
11	1267	12.67%	
12	1145	11.45%	
13	1027	10.27%	
14	712	7.12%	
15	452	4.52%	
16	279	2.79%	
17	116	1.16%	
18	48	0.48%	
-----+	-----+	-----	

These functions take the size of the batch and either an FRP or a Kind. Compare these proportions with the Kind of `DiceSum` above, and try several (perhaps larger) demos. (You may not get the same values in your output because you are obtaining a different FRP with the same Kind.)

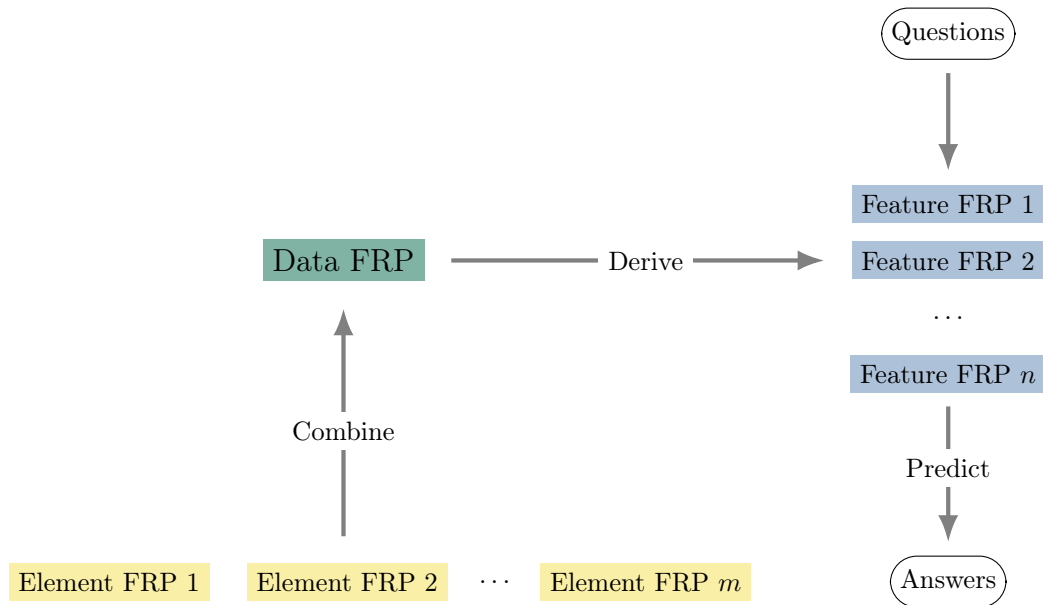


FIGURE 1.12. Schematic of how FRPs are used. The value of the Data FRP represents all the data we measure or observe from a random system. This is typically built by *combining* simpler Element FRPs whose Kinds and interaction are determined by our knowledge of and assumptions about the system. From the data, we *derive* feature FRPs whose values represent features of the data that answer the questions that motivate our analysis and are the target of our predictions.

With this example in hand, we are in a position to refine the schema given in Figure 1.9. We saw here how we built the Data FRP by operating on the simple elements of the system, in this case the FRPs for the individual die rolls. The operation that combines these elements, called a *join*, is one of our **Big 3+1 operations** and is what we use to inject new randomness into a system. We also saw that we can have multiple questions and thus multiple Feature FRPs that we derive from the Data FRP, like the sum and the minimum of the dice rolls.

Figure 1.12 shows the refined picture. Here, we build the *Data FRP* by combining random quantities that are easier to reason about and understand, represented by the *Element* FRPs, which are the building blocks of the system. In the dice example, the Elements FRPs represent the rolls of the three individual dice. The Kinds of the Element FRPs are based on our knowledge of and assumptions about the system. These Kinds determine the Kind of the Data FRP, which in turn determines the Kinds of the Feature FRPs, which in turn determine our predictions. In that way, the Kinds and predictions that we use to answer our questions are based on a **model**, a collection of assumptions about the Kinds of the Element FRPs and the relationships through which they combine and interact.

For our next example, we go to the probability theorist’s happy place: flips of a coin. The specific random system we will consider consists of two coins, one gold and one silver. We flip the gold coin. If it comes up heads, we flip the gold coin a second time; if the first flip comes up tails, we flip the silver coin instead for the second flip. While there are many different aspects of this system that we might observe and record – the specific paths of the coin in space, the point of contact between the thumb and the coin, and so forth – we will focus our description and analysis on the more direct outcome of which sides of the coins end facing up on both flips. Thus, the data that we measure from this system will be a 2-tuple listing the results of the two flips in order. Here, the Element FRPs from Figure 1.12 are the individual coin flips, two FRPs for flips of the gold coin and one for the flip of the silver coin. We build an FRP from these elements to represent these data and call it *C* for “coin flips”. This is the *Data FRP* in the framework of Figure 1.12. We will denote the values of the Element FRPs – and the components of *C*’s value – by either  /  (tails) or  /  (heads).

We also have two questions about this system whose answers we would like to predict: (i) how many times will heads come up in these two flips? and (ii) how many times will the gold coin be flipped? The answer to the first question will be represented by an FRP *H*, for “heads”.²⁴ *H* is derived from *C* in the sense that *C*’s

²⁴Notice our use of the Naming Convention for FRPs described on page 13. We use capital Roman letters, possibly labeled with subscripts, that are chosen to be meaningful or mnemonic if possible.

value determines H 's value. We analyze this system when both C and H are fresh. Notice, however, that while C 's value determines H 's value exactly, observing H 's value only gives us *partial* information about C . For instance, there are two values of C that are consistent with H having the value 1. Predicting the value of H gives us a predicted answer to our first question. The answer to the second question will be represented by an FRP G , for “gold”. Again, G is derived from C . For each possible value of C , we can find the value of G with a simple algorithm. Both of these FRPs H and G represent different features of C , random quantities whose values are computable once C 's value is available. These are *Feature FRPs* in the schema of Figure 1.12.

We can play with this system in `frplib` to get a feel for it before we look more closely at how to build and analyze this system. First, we load an object `coin_flips` from the `hello_world` example module.²⁵ Properties of this object generate fresh copies of C and create H from a copy of C . For example:

```
pgd> from frplib.examples.hello_world import coin_flips
pgd> C = coin_flips.symbolic()
pgd> C
An FRP with value <H, T>
pgd> H = coin_flips.count_heads(C)
pgd> H
An FRP with value <1>
pgd> G = coin_flips.count_gold(C)
pgd> G
An FRP with value <2>
```

The function `coin_flips.symbolic` returns a fresh FRP whose values are symbolic as above. The functions `coin_flips.count_heads` and `coin_flips.count_gold` transform the flip FRP C to FRPs that counts the heads in the two flips and the number of gold flips. Notice that we need not distinguish the gold and silver coins in the value of C because the first flip is always a gold coin and the color of the second flip's coin is determined by result of the first flip. We can also look at the first flip and second flip as isolated FRPs:

```
pgd> C[1]      # The FRP representing the first flip
An FRP with value <H>
pgd> C[2]      # The FRP representing the second flip
An FRP with value <T>
```

²⁵If you import `frplib.examples.hello_world` you can type `help(hello_world)` to learn more about the example and what the module contains. This works for any `frplib.examples` module.

Again, all text after # is a comment; you need not enter that.

Puzzle 10. We do not label the color of the coin in the value of C as represented above, but we could include this information if we wanted to. Describe a direct way to represent the values of an FRP like C that would include both the coin colors and the flip outcomes. What would its dimension be?

For the moment, assume that both coins are balanced, with symmetry between heads and tails. We can generate many copies of C and tabulate the results, using either `Market.demo` or `FRP.sample`.

```
pgd> FRP.sample(10000, C)
      Summary of Output Values
+-----+-----+-----+
| Values | Count | Proportion |
+-----+-----+-----+
| <H, H> |  2485 |    24.85% |
| <H, T> |  2518 |    25.18% |
| <T, H> |  2418 |    24.18% |
| <T, T> |  2579 |    25.79% |
+-----+-----+-----+
```

As usual, this table exhibits sample variation. If we repeat the demo or increase the sample size, we can convince ourselves that the proportions here hover around 25% for all four values. This fits with our intuition about the symmetry between heads and tails that we assumed. This suggests that we can express the Kind of C as in the left panel of Figure 1.14.

However, we want to understand C 's Kind in a way that reflects what we know about individual flips, both to clarify the structure of the system and to let us to vary the assumptions in our model. So, think about C as generating its value “in stages,” the first flip and then the second. We can decompose the above demo table in such stages, giving the proportions of each outcome for the first flip and the proportions of each outcome for the second flip *for each outcome of the first flip*.

For the first flip, count the numbers of samples in the demo table where the first component of the value are, respectively, H and T. We get 50.03% H (from the first two rows of the table) and 49.97% T (from the last two rows of the table).

Consider next the outcomes of the second flip in cases where we *know* the first flip is H and thus are flipping the gold coin a second time. This focuses our attention

on the first two rows of the table and how around 49.7% are H and 50.3% are T.

Finally, consider the outcomes of the second flip in cases where we know the first flip is T and thus are flipping the silver coin for the second flip. This focuses our attention on the last two rows of the table, and of these around 48.4% are H and 51.6% are T.

If we repeat the demo multiple times, we will see that all these proportions fluctuate closely around 50% due to sampling variation. This tells us the Kinds at each stage of the system's evolution, which are illustrated in Figure 1.13. The weights on these Kinds are the idealized proportions without sampling variation and hence all equal in these cases. The Kind on the left corresponds to the first flip. The upper Kind on the right is the Kind of the second flip after we have already seen a T on the first flip. The lower Kind on the right is the Kind of the second flip after we have already seen a H on the first flip. All three reflect the assumed balance between H and T for both coins.

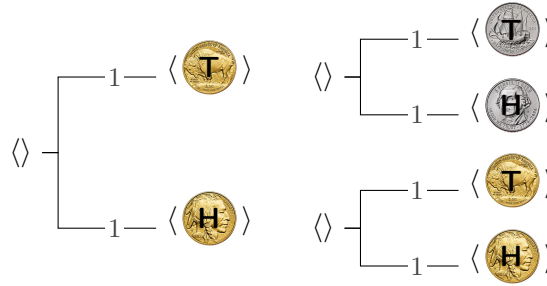


FIGURE 1.13. The Kinds of the individual flips comprising the FRP C , arranged in stages. These are the three Kinds of the Element FRPs for this system. We attach the roots at right to the leaves at left and propagate the value of the first flip into the leaves to satisfy the Prefix Rule (Definition 1), creating the Kind in Figure 1.14.

Figure 1.13 gives us the ingredients to construct the Kind of C in a way that clarifies what we know at each stage of the system's evolution. To produce the Kind, we attach the roots of the two trees on the right to the corresponding leaves of the left tree, and we propagate the value of the first flip into the leaves to ensure that the resulting tree follows the rules of Kind construction laid out in Section 1.2. This gives us the Kind of C shown in the right panel of Figure 1.14, with dimension 2, width 2, and size 4.

The first stage (i.e., the first flip) can produce either $\langle \text{T} \rangle$ or $\langle \text{H} \rangle$, with equal weights on the two possibilities. At the second stage, we know the value of the first flip, and we are either in the upper subtree at $\langle \text{T} \rangle$ or the lower subtree at $\langle \text{H} \rangle$. From that point, we *know* the first component of the value because result of the first

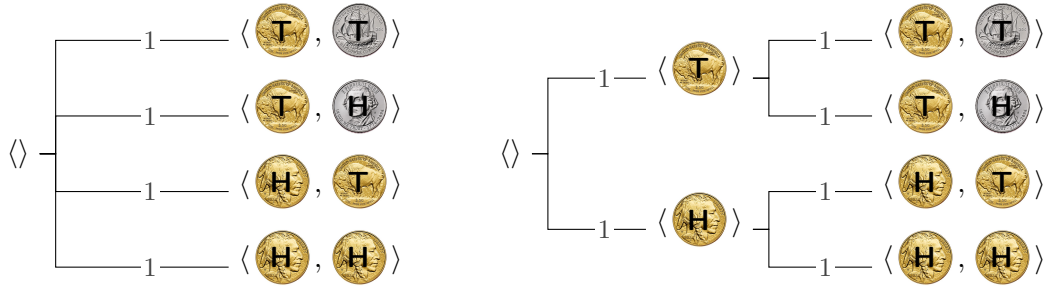
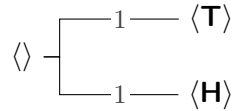


FIGURE 1.14. The Kind of the FRP C assuming that both coins are balanced, in two equivalent forms. The left Kind has width 1 and reflects what we see in a large demo of C . The right Kind shows the structure of the flips in stages, revealing how the second flip is impacted by the outcome of the first flip.

flip is just propagated into the component. Looking forward from the  subtree, the second component is either  or  with equal weight. Looking forward from the  subtree, the second component is either  or  with equal weight.

As we move down the tree in the right panel of Figure 1.14, everything we know about the value of C is reflected in the tree that we see looking from where we are toward the leaves. At the first stage, before the first flip, we are at the root of the tree, with all four values possible. After the first flip, we are either at the root of the $\langle \text{T} \rangle$ subtree or at the root of the $\langle \text{H} \rangle$ subtree. In each case, the two possibilities remaining have the known value of the first flip in the first component.

We can see then that $\text{kind}(C)$ is built from copies of the Kind



which is the Kind of an FRP representing a single balanced coin flip (for either coin). These pieces are shown in Figure 1.13. One copy of this Kind governing the first flip with the gold coin; this becomes the top of the tree. If the first flip is tails, we see another copy governing the second flip with the silver coin. If the first flip is heads, we see another copy governing the second flip with the gold coin. The operation that builds C from the FRPs for individual flips and that builds the Kind in Figure 1.14 from the Kinds in Figure 1.13. is called a **join**.²⁶

So far in this example, we have used FRPs whose values are *symbols* like H and T. While it is fine to have FRPs with arbitrary values (e.g., symbols, images, strings), in practice we avoid this in favor of *numeric* values. This means that we will encode all

²⁶See Section 1.5 and Chapter 4.

components of FRPs' values as numbers. When the data is a numeric measurement, the measurement itself gives us the number. When the values are not intrinsically numeric, we assign numeric tuples to each possible value, as meaningfully as possible. In this case, "heads" and "tails" are merely labels for the two sides of the coin, so we encode this numerically by arbitrarily assigning a distinct scalar to each label. Here, we arbitrarily assign *the value* $\langle 0 \rangle$ for tails,  or , and *the value* $\langle 1 \rangle$ to heads,  or .²⁷

²⁷Recall that for scalars, we do not distinguish between the component and the 1-dimensional tuple, e.g., 0 vs. $\langle 0 \rangle$.

Puzzle 11. Give an assignment of numbers for the components of C 's values that indicates both heads and tails and which color coin is being flipped.

We can play with the numeric encoding in `frplib` by using `coin_flips.numeric`:

```
pgd> from frplib.examples.hello_world import coin_flips
pgd> C = coin_flips.numeric()
pgd> C
An FRP with value <1, 0>
pgd> H = coin_flips.count_heads(C)
pgd> H
An FRP with value <1>
pgd> G = coin_flips.count_gold(C)
pgd> G
An FRP with value <2>
pgd> FRP.sample(10_000, C)
```

Summary of Output Values

Values	Count	Proportion
<0, 0>	2560	25.60%
<0, 1>	2464	24.64%
<1, 0>	2487	24.87%
<1, 1>	2489	24.89%

The assumption that the coins are balanced, while pleasing in its symmetry, is limiting, and we want to allow for a broader range of models. For example, we might assume that the gold and silver coins each have a weighting that governs all flips

with those coins. We can define positive numbers g and s where the Kinds for flips of the gold and silver coins are respectively



Joining these element Kinds as we did when moving from Figure 1.13 to Figure 1.14, we get the Kind of C in Figure 1.15. The values g and s are called **parameters** of our model. A choice of these parameters represents an assumption about the flips. For instance, setting $g = 1 = s$ assumes that both coins are balanced, as above.

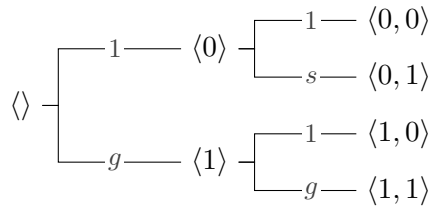


FIGURE 1.15. The Kind of C when its values are encoded numerically and where we allow different and possibly unbalanced weightings for the two coins, denoted by g for the gold coin and s for the silver coin. Compare with the Kind on the right in Figure 1.14.

This Kind exposes the two stages of the system's evolution: the first coin is flipped, and then the second coin is flipped contingent on the result of the first flip. Note that the same weights are used for both flips of the gold coin, reflecting an assumption that all flips of each coin are interchangeable. Our *model* for this random system consists of the collection of Kinds indexed by the parameters $g, s > 0$. Each choice of the parameters g, s specifies the structure of the random system, and the range of possible parameters allows for varied structure.²⁸

You can play with this system in `frplib` by providing the g and s parameters to the `coin_flips.numeric` function.

```
pgd> from frplib.examples.hello_world import coin_flips
pgd> C = coin_flips.numeric(g=2, s=1/3)
pgd> C
An FRP with value <1, 0>
pgd> H = coin_flips.count_heads(C)
pgd> H
An FRP with value <1>
pgd> G = coin_flips.count_gold(C)
```

²⁸We can visualize this space of possible parameters as the positive-positive quadrant of the plane, with coordinates $\langle g, s \rangle$.

```

pgd> G
An FRP with value <2>
pgd> FRP.sample(10_000, C)
  Summary of Output Values
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| <0, 0> | 2544 | 25.44% |
| <0, 1> | 851 | 8.51% |
| <1, 0> | 2202 | 22.02% |
| <1, 1> | 4403 | 44.03% |
+-----+-----+-----+

pgd> C = coin_flips.numeric(g=3, s=3)
pgd> C
An FRP with value <1, 1>
pgd> H = coin_flips.count_heads(C)
pgd> H
An FRP with value <2>
pgd> FRP.sample(10_000, H)
  Summary of Output Values
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| 0      | 675 | 6.75% |
| 1      | 3774 | 37.74% |
| 2      | 5551 | 55.51% |
+-----+-----+-----+

pgd> E(H) # Our prediction of H, it's risk-neutral price
3/2

pgd> G = coin_flips.count_gold(C)
pgd> FRP.sample(10_000, G)
  Summary of Output Values
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+

```



```

| 1      | 2531 | 25.31% |
| 2      | 7469 | 74.69% |
+-----+-----+-----+
pgd> E(G)
7/4

```

If we repeat the demos for H and G , we can guess their Kinds. This is a worthwhile exercise – try it – but we do not have to guess. Using $g = 3$ and $s = 3$ as above,

```

pgd> kind(H)
,---- 1/16 ---- 0
<> -+---- 6/16 ---- 1
    `---- 9/16 ---- 2

pgd> kind(G)
,---- 1/4 ---- 1
<> -|
    `---- 3/4 ---- 2

```

The Kinds of H and G are determined from the Kind of C using the specific transformation that defines each feature FRP.

For instance, the value of H is obtained by applying a simple function $\langle c_1, c_2 \rangle \mapsto c_1 + c_2$ to whatever value C produces. (Try it on all four possible values of C .) A function like this that maps values to values is called a **statistic**,²⁹ and we call H a *transform* of C by this statistic. When C is activated, so is H . H represents the “feature” of C that sums the components of its value.

²⁹Statistics are discussed in detail in Chapter 2.

Just as there is a relationship between the FRPs C and H using this statistic, there is a parallel relationship between their Kinds mediated by the same statistic. What we know about H ’s value is determined by what we know about C ’s value and the statistic that derives H ’s value from C ’s value. For example, in a simpler version of our model, where g and s are equal, with $w = g = s$, we can compute the Kind of H for each value of w :

$$\begin{array}{c}
 \langle \rangle \left\{ \begin{array}{l}
 \text{---} 1 \text{---} \langle 0 \rangle \\
 \text{---} 2w \text{---} \langle 1 \rangle \\
 \text{---} w^2 \text{---} \langle 2 \rangle
 \end{array} \right.
 \end{array}$$

We learn how to derive this and to compute it in **frplib** in the next chapter.

From the Kind of H , we can make predictions about H 's value. Building on what we found in the previous section, Chapter 6 will show how to construct good predictions of an FRP's value from its Kind with its “risk-neutral” price. We can apply this to find the risk-neutral price of H , denoted by $\mathbb{E}(H)$. For a specific value of w in our simpler model, we have

$$\mathbb{E}(H) = \frac{0 \cdot 1 + 1 \cdot 2w + 2 \cdot w^2}{1 + 2w + w^2} = \frac{2w}{1 + w}.$$

This ranges from 0 to 2 as w ranges from 0 to ∞ . For instance, if w is very, very small, then both coins produce tails almost all of the time, and a large demo of FRPs of Kind $\text{kind}(H)$ will show almost all 0s. Thus, we would not pay much for the payoff from H . In contrast, if w is very, very large, FRPs like H will tend to produce 2, and so we would pay just under \$2 for that payoff. The point is that for any choice of w , we can make reasonable predictions about H 's value.

We can do a similar analysis with G , the FRP representing the number of flips of the gold coin. G 's value is obtained from C 's value by the simple statistic $\langle c_1, c_2 \rangle \mapsto 1 + c_1$, as we always flip the gold coin at least once and do it a second time if the first flip is heads (1). For each specific value of w , this analysis will give us $\text{kind}(G)$:

$$\langle \rangle \begin{cases} \text{---} 1 + w \text{---} & \langle 1 \rangle \\ \text{---} w(1 + w) \text{---} & \langle 2 \rangle \end{cases}$$

with a risk-neutral price

$$\frac{1 + 3w + 2w^2}{1 + 2w + w^2}.$$

Aside. We can do calculations in `frplib` with symbolic quantities. For example, to compute the Kinds of H and G as above, we create a new symbol for the weight w and then compute the Kinds as before:

```
pgd> w = symbol('w')    # Creates a symbolic variable w
pgd> C = coin_flips.numeric(g=w, s=w)
pgd> H = coin_flips.count_heads(C)
pgd> G = coin_flips.count_gold(C)
pgd> unnormalized(kind(H))
pgd> unnormalized(kind(G))
```

This will show the Kinds as above. Without the call to `unnormalized`, the displayed Kinds would have canonical (normalized) weights.

As we will see, we can also obtain these Kinds by transforming the Kind of C :

```
pgd> kind(C) ~ Sum           # Yields the Kind of H
pgd> kind(C) ~ (1 + Proj[1]) # Yields the Kind of G
```

The `~` operator is used to *transform* FRPs and Kinds by statistics. Don't worry about the details for now; this is just to give you a sense of what is possible.

The Kinds for H and G and our predictions for their values, are given above for each value of w . Each value of w uniquely specifies the system. If we know w – if we know the specification of the system – we can make predictions about how the system will evolve. That is one of the key uses of probability theory – *probabilistic inference*: specifying a random system precisely and predicting the outcomes of running that system.

But there is another common situation we face: when we do not know which specific value of w describes the system among a range of possibilities but we see how the system evolves. In this situation, we can use the data to *learn* about the value of w . This is the other key use of probability theory – *statistical inference*. We learn about the system based on our model of the system and the data we observe when that system evolves.

Our framework in Figure 1.12 works for both uses. For probabilistic inference, we specify the parameters of the model and make predictions about the answers to our questions. For statistical inference, we have a family of possible Kinds that can describe the Data FRP, and we observe the value of that FRP, or some Feature FRPs. We use a basic tool, called the **likelihood principle**, to convert the observation of an FRP from the system into information about which Kind in the family describes the Data FRP.

Here, we will give a simple illustration using a modified form of the coin flip random system above. Rather than just a single pair of flips, we will repeat the runs of the system three times: on pair of flips, then another, then another. We observe the *total* number of heads in all flips. This is the value of a 3-dimensional feature FRP, which we will call N .

```
pgd> C_3 = coin_flips.repeated(3, g=_, s=_) # Data FRP, parameter values obscured
pgd> N = coin_flips.count_heads(C_3)         # Observed Feature FRP
An FRP with value <1>
```

Here, we use the simplified model in which $g = s = w$ for some w , but I have obscured the value of the parameters for the Data FRP with a $_$. (The value of w I used will be revealed later.)

The Kind of N , in terms of the unknown parameter w , is given in canonical form³⁰ by

$$\begin{array}{l} \langle \rangle \left\{ \begin{array}{l} \text{---} 1/p(w) \text{---} \langle 0 \rangle \\ \text{---} 6w/p(w) \text{---} \langle 1 \rangle \\ \text{---} 15w^2/p(w) \text{---} \langle 2 \rangle \\ \text{---} 20w^3/p(w) \text{---} \langle 3 \rangle \\ \text{---} 15w^4/p(w) \text{---} \langle 4 \rangle \\ \text{---} 6w^5/p(w) \text{---} \langle 5 \rangle \\ \text{---} w^6/p(w) \text{---} \langle 6 \rangle \end{array} \right. \end{array}$$

³⁰This implies that all the weights are normalized to sum to 1.

where $p(w) = 1 + 6w + 15w^2 + 20w^3 + 15w^4 + 6w^5 + w^6$.

For probabilistic inference, where we know the quantity w , **we have one Kind (for the specific choice of w) and many possible values (yet to be observed)**. The weights tell us the relative propensity of the FRP producing each value. For instance, the seven possible values would appear in a large demo in proportions roughly equal to the canonical weights. The canonical weight on 1, for instance, is $6w/p(w)$, and this is the approximate proportion of 1s that will occur in a large demo. In this context, we will call the canonical weight on 1, the **probability** of seeing 1 from this system, and similarly for all other possible values.

For statistical inference, however, we have the dual situation: **we have many possible Kinds (corresponding to different possible w 's) and only one value (that which we observed)**. The canonical weight for the observed value 1 is still $6w/p(w)$, the same number, but our interpretation of this number is different than in probabilistic inference. We use that weight for every possible Kind in our family as a score measuring how well that Kind fits the observed data, with larger values implying a better “fit”. In this context, we will call the canonical weight on 1 for a Kind in the family, the **likelihood** of that Kind.

So, having observed the value $\langle 1 \rangle$ for N , we use the likelihood of N 's Kind as a function of w to estimate w . Here, we will use a simple approach of picking the w that maximizes the likelihood. Figure 1.16 shows the likelihood as a function of w . The plot is truncated arbitrarily at $w = 2$ for clarity, because the likelihood continues

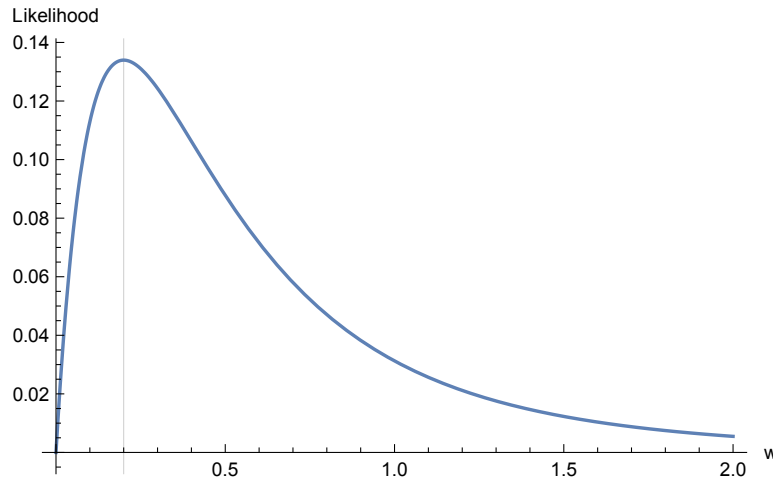


FIGURE 1.16. The likelihood for $\text{kind}(N)$ as a function of w . The range of the plot is limited to $[0\_2]$, but the likelihood continues to decrease for $w > 2$. The vertical line at $w = 1/5$ shows the point at which the likelihood is maximized.

to decrease for $w > 2$. The vertical line shows that the likelihood hits its maximum at $w = 1/5$. This is the Kind in our family of possible Kinds that makes the value that we observed most “likely”. So one approach to statistical inference here is to *estimate* that w is 0.2. This estimate is uncertain because there is sampling variability in the small sample of data we observed from the process. The actual value of w used to generate the data above was 0.22. Was our estimate a good one? On the face of it, it seems quite close, but we will learn how to assess and generalize this procedure as we proceed.

Perhaps this all seems like a lot of work to handle a couple coin flips, which seem simple. Worry not, we will streamline these steps significantly. But the key point is that the *same framework and steps* apply to any system we model. This careful breakdown is meant to identify the key pieces and steps we use in modeling general random systems in Figure 1.12.

Our last example is the (in)famous Monty Hall game. Monty is the host of our game, and you are the contestant. The game proceeds as follows:

1. You are presented with three doors: left, middle, right.
2. Monty has secretly selected a door at random and placed a prize behind it; the other two doors have nothing behind them.
3. You choose a door.
4. Monty opens one of the other doors that does not hide the prize.
5. He offers you a chance to switch your initial choice to the other unopened door.

6. You indicate whether you will switch your choice of doors.
7. Monty opens your final choice of door. If the prize is behind it, you win; otherwise, you lose.

Should you accept Monty's offer to switch?

To begin, consider the *strategies* available to you in this game. Each strategy specifies: (i) how you pick your initial door, and (ii) whether you accept Monty's offer to switch from your initial door choice. For (i), we allow random choices of door, so we specify this with a Kind whose values are the three doors. For example, one strategy is $\langle \text{constant}(\text{Left}), \text{Do not switch} \rangle$, where you always choose the left door and stick to that choice; another is $\langle \text{uniform}(\text{Left}, \text{Middle}, \text{Right}), \text{Switch} \rangle$, where you choose among the three doors with equal weight and accept Monty's offer to change your choice. Each choice of strategy is thus a pair $\langle k, s \rangle$ where k is the Kind of an FRP whose possible values are the doors and s is a Boolean binary indicating whether you switch (0 for don't switch, 1 for switch). We will analyze each distinct strategy separately; specifically, we will **use one FRP per strategy** to represent the game's outcome.

Once your strategy has been specified, the outcome of the game is the sequence of decisions made at each stage:

1. Monty hides the prize behind the left, middle, or right door.
2. You select either the left, middle, or right door according to your strategy.

These are illustrated in Figure 1.17.

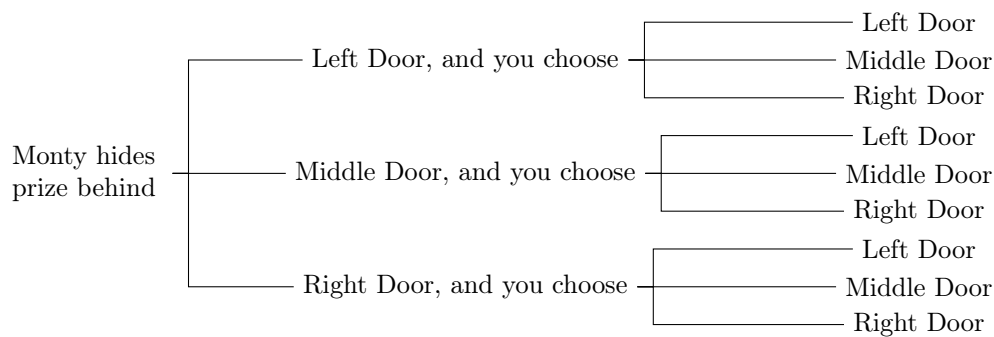
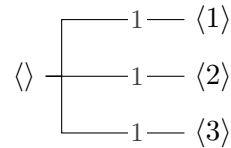


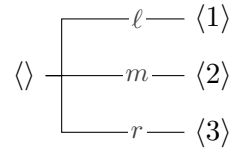
FIGURE 1.17. The decision tree leading to your initial door choice in the Monty Hall game. For any given strategy, these are the choices that determine the outcome of the game.

We describe this system with an FRP that represents the outcome of the game. This is the Data FRP in our framework. Keeping with our convention of using

numeric FRPs, we assign numeric quantities to the choices at each stage, using 1 for the Left door, 2 for the Middle door, and 3 for the Right door. (This is an arbitrary choice, one can use any distinct numbers.) There are two elements making up the Data FRP: Monty's choice of door and your choice of door. First, we will assume for simplicity that Monty places the prize behind a door picked at random with equal weight on each door. The Kind of this Element FRP is thus $\text{uniform}(1, 2, 3)$, i.e.,



Second, if your strategy is $\langle k, s \rangle$, then we will write the Kind k as



with positive weights ℓ, m, r on the three doors. The Data FRP combines the output values of Monty's choice and your choice, with neither choice influencing the other. This is the type of *join* we called an **independent join** earlier. We construct the Kind of the Data FRP by attaching a copy of k to each leaf node of Monty's uniform Kind, and propagating the values to the leaves to respect the rules of Kinds. The resulting Kind is shown in Figure 1.18. The “in stages” interpretation of the Data FRP is that Monty makes a choice and then you make a choice, and the outcome of the game is collects these choices in order.

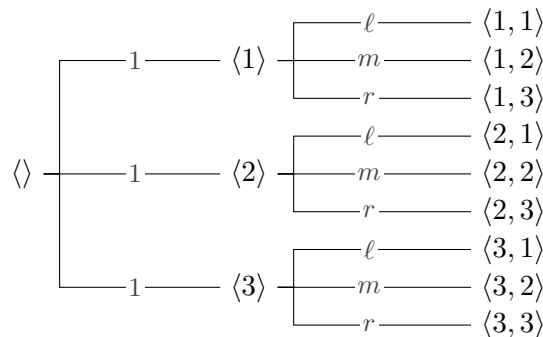


FIGURE 1.18. The Kind for the FRPs modeling the Monty Hall game. In each value list, the first element is Monty's door choice and the second element is your door choice. The weights ℓ , m , and r are discussed in the text.

To analyze this system, we want to predict the answer to a key question: do you win the game? For each strategy choice $\langle k, s \rangle$, we get a predicted answer to this question, and we can compare these predictions across strategy choices. This is a common theme in probabilistic inference; we have a range of choices in how to configure the system and look at our predictions as a function of these configuration parameters.

In this case, it turns out that for each value of s , every choice of weights in your strategy (i.e., in k) gives the same predicted answer. So, it is sufficient for our purposes here to focus on our prediction of whether you win as a function of whether you switch. We will thus compare the “Don’t Switch” and “Switch” strategies, with arbitrary weights in k .

Puzzle 12. What does it say about your strategy if ℓ , m , and r are all equal? What does it say about your strategy if $\ell = 1 = r$ but m is very, very, very large?

I claimed earlier that the value of the Data FRP describes the *outcome* of the game, yet those values merely specified the door choices not whether you win. The Data FRP’s values describe all the fundamental ways the system can evolve, restricting to relevant and observable measurements. The answers to our questions can be *derived* from those data. So here, once we’ve specified your strategy $\langle k, s \rangle$, whether you win is a function of the door choices. If you know the value of the Data FRP, you can answer the question of whether you win. (Again, remember that we have one Data FRP for each choice of strategy.) When you do *not* switch ($s = 0$), your final choice of door is the same as your initial choice of door, and when you do switch ($s = 1$), your final choice of door is uniquely determined by your initial choice of door and where Monty put the prize (because he will only open a door without a prize that you did not select). Try the following puzzle to see how this works.

Puzzle 13. (Important!)

For each leaf node in Figure 1.18, fill in the table below indicating whether you Win or Lose under the DON’T SWITCH and the SWITCH strategies.

Value	Don't Switch	Switch
$\langle 1, 1 \rangle$		
$\langle 1, 2 \rangle$		
$\langle 1, 3 \rangle$		
$\langle 2, 1 \rangle$		
$\langle 2, 2 \rangle$		
$\langle 2, 3 \rangle$		
$\langle 3, 1 \rangle$		
$\langle 3, 2 \rangle$		
$\langle 3, 3 \rangle$		

Your answer to this puzzle shows that whether you win the game is a *function* of the outcome, that is, a function of the Data FRP's value. As described earlier, we call a function that takes a value (tuple of numbers) and returns a value (tuple of numbers) a **statistic**. So the answer to our question, whether we win, is a feature FRP obtained by transforming the Data FRP with a statistic.

Specifically, if the Data FRP's value is denoted by $\langle d_{\text{Monty}}, d_{\text{You}} \rangle$, then

- If you do not switch ($s = 0$), you win if and only if $d_{\text{Monty}} = d_{\text{You}}$.
- If you do switch ($s = 1$), you win if and only if $d_{\text{Monty}} \neq d_{\text{You}}$.

We thus use one statistic when $s = 0$ and another statistic when $s = 1$. For such true-false statistics, we will return a “Boolean binary,” with $\langle 0 \rangle$ for false (i.e., you lose) and $\langle 1 \rangle$ for true (i.e., you win).³¹

Puzzle 14. Describe these two statistics explicitly, in code or mathematics or even in words.

We can now transform the output of each Data FRP using the corresponding statistic – one for SWITCH and one for DON'T SWITCH.³² This gives us new “feature FRPs” with Kinds shown in Figure 1.19 for the no switch case (top) and the switch (bottom).

We can play with this system in the `frplib` playground.³³ First, load the module `frplib.examples.monty_hall` from `frplib`.

```
pgd> from frplib.examples.monty_hall import (
...>     door_with_prize, chosen_door,
...>     got_prize_door_initially,
```

³¹Remember that we elide the distinction between a scalar and a 1-dimensional tuple in practice.

³²We will see how to do this formally in Chapter 2.

³³Here and later, text from # to the end of a line is a comment for your benefit. You need not type the comments.


```
pgd> Game = frp(outcome_by_strategy(left=1, middle=1, right=1))
pgd> Game
An FRP with value <2, 3>
```

Here, we specify numeric rather than symbolic values for ℓ, m, r so that we can observe the FRP.

The statistic `got_prize_door_initially` takes a pair $\langle d_{\text{Monty}}, d_{\text{You}} \rangle$, where d_{Monty} is the door with the prize and d_{You} is the door you chose initially, and returns 1 (for true) if $d_{\text{Monty}} = d_{\text{You}}$ or 0 (for false) otherwise. Transforming the Data FRP for the game with this statistic gives a feature FRP whose value represents whether you win under the DON'T SWITCH strategy (or lose under the SWITCH strategy).

```
pgd> Win_no_switch = Game ^ got_prize_door_initially
pgd> Win_no_switch
An FRP with value <0>
```

which has Kind

```
pgd> kind(Win_no_switch)
,---- 2/3 ---- 0
<> -+
`---- 1/3 ---- 1
```

which is consistent with the DON'T SWITCH Kind in Figure 1.19 in that losing has twice the weight of winning. Again, the transform operator $\wedge$ here takes an FRP or Kind on the left and a statistic on the right; think of the arrow to mean that the value of the FRP is passed *through* the statistic.

From `got_prize_door_initially`, we can define the complementary statistic `didnt_get_prize_door_initially`

```
pgd> didnt_get_door_prize_initially = Not(got_prize_door_initially)
```

This statistic returns 1 when `got_prize_door_initially` returns 0 and vice versa.³⁵ Transforming `Game` (and its kind) with this statistic gives

```
pgd> Win_switch = Game ^ didnt_get_prize_door_initially
pgd> Win_switch
An FRP with value <1>
pgd> kind(Win_switch)
,---- 1/3 ---- 0
<> -+
`---- 2/3 ---- 1
```

³⁵Not here is an `frplib` *statistic combinator*, it is *not* the Python built-in `not`.

which is consistent with the SWITCH Kind in Figure 1.19. These Kinds make clear that switching is the better choice.

A recurring theme in the coming chapters is that the operations we do on FRPs have directly analogous operations on Kinds, and vice versa. For example, we can transform the Kind `game_outcome` with the same statistics:

```
pgd> dont_switch_win = game_outcome ^ got_prize_door_initially
pgd> dont_switch_win
,---- 2/3 ---- 0
<> -+
`---- 1/3 ---- 1

pgd> switch_win = game_output ^ didnt_get_prize_door_initially
pgd> switch_win
,---- 1/3 ---- 0
<> -+
`---- 2/3 ---- 1
```

This yields the same Kinds. It also shows that your choice of weights ℓ, m, r on the doors do not change our prediction, as `game_outcome` has arbitrary weights but the result is independent of those weights.

Similarly, the independent join operation applies to FRPs as well as Kinds.

```
pgd> Montys_choice = frp(door_with_prize)
pgd> Your_choice = frp(substitution(chosen_door, l=1, m=1, r=1))
pgd> Game2 = Montys_choice * Your_choice
An FRP with value <2, 1>
```

We use `substitution` to assign numeric values to the symbolic weights in `chosen_door` so that we can construct an FRP from that Kind. `Game2` is a clone of `Game`; they have the same Kind as you can check.

Finally, we can also observe FRPs of each Kind to simulate the outcome of many games.

```
pgd> FRP.sample( 12_000, dont_switch_win )
Summary of output values:
0          7929 (66.1%)
1          4071 (33.9%)
```

```
pgd> FRP.sample( 12_000, switch_win )
```

Summary of output values:

0	3954 (32.9%)
1	8046 (67.1%)

Here, we pushed the buttons on 12,000 FRPs of each Kind `dont_switch` and `switch`. Again, the results are clear cut: switching is the best choice.

Key Take Aways

FRPs represent random quantities – data – that can be observed/measured during the evolution of a random system. A fresh FRP represents a meaningful random quantity whose value *has not yet been determined*. An activated FRP represents a random quantity whose value has been determined but *that has not yet been observed*. And observed FRP represents a random quantity whose *value we know*. Pushing the button on an FRP corresponds to observing the corresponding quantity.

But because our analysis is done *before* observing the full data – before all these FRPs are activated – it is the *Kinds* of the FRPs that are most directly useful for our analysis. Recall that the Kind of an FRP embodies everything we know about its value. So the Kind of the Data FRP incorporates our knowledge and assumptions about the random system itself. We thus *model* the system by assuming the form of this Kind.

A *random system* is any process or scenario where we can (eventually) observe information about the system’s output and state that may be uncertain. To specify a random system, we build a special FRP whose value represents all the relevant data that will be obtained from that system, which we call the “Data FRP”. This gives us the basic data that pertains to our goals and that we can use for to answer which questions drive our analysis of the system. The answers to our questions are expressed in terms of features that are derived from the data, and each such answer is also represented as an FRP, which we call “feature FRPs”. Predicting the values of these “feature FRPs” gives us predicted answers to our questions.

1.5 The Big 3+1 and the Likelihood Principle!

Almost everything we will do with probability theory – simulation, modeling, inference, decision making, prediction, and learning – is built on four basic operations, which we call the **Big 3+1**, and one key principle, the **likelihood principle**.

1.5.1 The Big 3+1: Fundamental Operations on FRPs and Kinds

The “3” in the Big 3+1 operate on FRPs to produce new FRPs by connecting output ports to input ports in several ways: (1) transforming the output of an FRP by some algorithm (statistics), (2) generating random outcomes contingent on earlier outcomes (joins), and (3) accounting for partially observed information (observations). All of these operations can also act on Kinds to produce new Kinds, and the operations on FRPs and Kinds are consistent with each other. The fourth operation (called expectation) takes an FRP (or Kind) and yields a value that represents our prediction of the FRP’s value. We use these operations to describe systems, building the Data FRP and Feature FRPs from the Element FRPs. We use the same operations to analyze the systems, constructing the Kinds of the Data and Feature FRPs and computing predicted answers to our questions.³⁶

³⁶cf. Figure 1.12

- **Transforming with Statistics** (Chapter 2)

A *statistic* is just a function that maps values (i.e., tuples of numbers) to values (tuples of numbers). Whenever we want to transform, summarize, or extract a feature from our data, we define a statistic that does the job. Whenever we apply an algorithm to process or analyze our data, we are using a statistic.

We can use statistics to *transform* FRPs and Kinds. We transform an FRP by *applying the statistic to the FRPs value*, producing a new but related FRP. We transform a Kind by *applying the statistic to each possible value of the kind* (i.e., the leaf nodes) and then combining branches that map to the same value in the transformed tree, adding their weights.

We use statistics to **express and answer questions**. The “Derive” step in Figure 1.12 uses statistics (and observations) to build feature FRPs whose values answer our questions. The statistic describes the steps we will take to extract the desired information from the data, *before those data are available*. Each feature FRP is derived by transforming the Data FRP with a statistic that represents one of our questions. The values of feature these FRPs will

answer those questions, and our goal is to predict their values (or compute their Kinds) as accurately as possible to guide our decisions and actions.

- **Building with Joins** (Chapter 4)

We use joins **to build a model by combining simpler parts**. When a system can most easily be described as a combination of smaller parts, think about using a join. The “Combine” step in Figure 1.12 uses joins (and statistics) to build the FRP for the data we measure from FRPs that represent the basic parts of the process or system.

In the dice and Monty Hall example, we have seen one type of join, where the different stages of the process do not interact (e.g., Monty chooses a door, you choose a door). This is called an independent join. More generally, though, the different elements will interact in that what happens at one stage influences the outcomes of later stages. This was the case in the coin flip example, where the second coin flipped depended on the outcome of the first flip. A general join reflects this by passing the values produced at one stage into the next and collecting that entire history.

We join FRPs by connecting the output port of the FRP representing one stage to a device that picks which FRP to trigger at the next stage based on the value it sees. We join Kinds by attaching the Kinds of the FRPs from the next stage to the corresponding leaf nodes of the Kind for the current stage, propagating values down the tree to be consistent with the Level and Prefix rules for Kinds.

- **Constraining with Observations** (Chapter 5)

We use observations to **update our knowledge and predictions with new information**. An observation is a constraint that some specific observable condition is *known to be true*, either because we actually observed that condition or because we are considering the hypothetical in which we observe it.

When we constrain a Kind with an observation, we simply *erase all the branches that are inconsistent with the condition*. This gives us a new Kind, which in canonical form simply re-normalizes the weights of the remaining branches by the total weight of branches that are consistent with the condition.

If you want to update your knowledge or predictions for some known information, use an observation.

- **Predicting with Expectations** (Chapter 6)

We use the risk-neutral price **to compute our best prediction of an FRP's value**, which we call its **expectation**. The expectation of an FRP reflects a “typical” value that is “close” in some sense to what the FRP will produce.

From their definition as risk-neutral prices, expectations inherit many useful properties, and from these properties, we can deduce how to compute the expectation of an FRP from its Kind. Computing expectations is often the target of our analysis because predictions can guide our behavior and decisions in the context of the system we are studying.

By understanding how the Big 3+1 operate – transforming values, erasing branches, combining stages, taking weighted averages – we can recognize and exploit these operations in complicated calculations and contexts.

1.5.2 Three Interpretations of the Big 3+1 Operations

While Chapters 2, 4, 5, and 6 discuss each of the Big 3+1 operations in detail, it will be useful to get an high-level look in advance at what these operations are and how we use them.

Let's step back for a moment and consider what it means to *operate* on FRPs. For the 3 of the Big 3+1, an operation takes one or more FRPs and creates a *new*, related FRP from them. We will see three interpretations of these operations:

1. *Physical*. We connect the physical FRPs that are the targets of the operation by wires and other hardware to create the new FRP that results from the operation. The values of the target FRPs are transmitted as signals over the wires and processed by the hardware to produce the value of the new FRP. Each operation has a *wiring diagram* that shows the connections and hardware used and defines the relationship between the new FRP that results and the FRPs used to define it.
2. *Algebraic*. We express the new FRPs by an algebraic expression in which we have defined *operators* that correspond to each of the Big 3 operations. The structure of this expression, with parentheses and operator precedence, defines the relationship between the new FRP that results and the FRPs operands.
3. *Procedural*. We write down a *procedure*, essentially a recipe for finding the value of the new FRP from the FRPs being operated on. The procedure shows the relationship between the new FRP that results by the computational steps taken.

We can translate freely among these interpretations. If I give you a wiring diagram,

you will be able to convert it to an algebraic expression in terms of operators or express it as a procedure, and similarly if I start with an algebraic expression or procedure. The three interpretations highlight different aspects of the operations and so collectively they all help our understanding of each. We will frequently use all three interpretations of these operations in practice and will learn to move among them easily.

We begin by illustrating these interpretations with a simple example. Let X be an FRP, and create a new FRP Y whose value is always the value of X with every component scaled by 2. This is actually an instance of the first of the Big 3, Transforming with Statistics, where we name the statistic $(\langle x \rangle \mapsto 2x)$ **Double**.

1. *Physical*. A wiring diagram defining Y is shown in Figure 1.20, showing a connection from X 's All output port through an adapter, a piece of hardware that computes the statistic **Double** from X 's value and sends it on the wire to Y 's input port, activating Y with this value when it is available. When X 's value is x , Y 's value is $2x$.

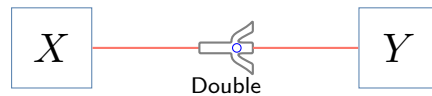


FIGURE 1.20. Wiring diagram for a simple transformation by the statistic **Double**. The FRPs are depicted in simple form as named boxes, with no buttons, displays, or ports shown. The statistic is represented by an adapter that computes the statistic on the value passed to it. The wire connects X 's All output port to Y 's input port, through the adapter.

2. *Algebraic*. For each of the Big 3+1, we define *operators* that enact that operation. We have two options for transforming with statistics: (i) the *transform operator* $\hat{}$ that takes an object on the left and a statistic on the right, $X \hat{} \text{Double}$; and (ii) the *overloaded evaluation operator* $()$ that takes a statistic to the left of the parentheses and an object inside the parentheses, $\text{Double}(X)$. The two are interchangeable, but sometimes one form is more convenient than the other. As we will see, we use both operators extensively in **frplib**, but mathematically, we mostly use the overloaded evaluation operator.
3. *Procedural*. An FRP *procedure* looks like:

FRP Y :

```
 $x \leftarrow X$ 
return  $2x$ 
```

This defines an FRP Y whose value is defined by the steps: (i) get X 's value when it is available, (ii) multiply that value by 2, (iii) set that as Y 's value. The $\leftarrow$ is the **yield** operation; it takes a name on the right and either an FRP (for FRP procedures) or Kind (for Kind procedures) on the right. It acts on FRPs and Kinds in related but distinct ways. For an FRP, yield binds the variable on the left-hand side of the arrow (in this case x) to the value of the FRP *when it is activated*. The **return** statement specifies the value of the resulting FRP when it is available.

Every operation in the Big 3+1 applies to both FRPs and to Kinds. The key point is that these operations are consistent. We can first operate on FRPs and then take Kind of the resulting FRP or we can first take the Kinds of the relevant FRPs and operate on those. **Both orders produce the same Kind.**

The three interpretations apply to Kinds as well in analogous fashion:

1. *Physical*. The new FRP Y has a metallic display that shows its Kind. We can read the Kind directly off the FRP.
2. *Algebraic*. We use the same operators for FRPs and Kinds. For instance, we again have two ways to transform $\text{kind}(X)$ with the statistic **Double**: (i) the *transform operator* $\hat{}$ that takes an object on the left and a statistic on the right, $\text{kind}(X) \hat{} \text{Double}$; and (ii) the *overloaded evaluation operator* $()$ that takes a statistic to the left of the parentheses and an object inside the parentheses, $\text{Double}(\text{kind}(X))$. Both of these are equal to $\text{kind}(Y)$.
3. *Procedural*. A Kind procedure has the same form as an FRP procedure

Kind k :

$$\begin{aligned} x &\leftarrow \text{kind}(X) \\ \text{return } 2x \end{aligned}$$

This defines a Kind k whose values are the values of $\text{kind}(X)$ scaled by 2. In a Kind procedure, the yield operation binds the variable on the left-hand side of the arrow (here also x) to every possible value of the Kind on the right *in parallel*. The weights associated with each value are tracked and managed automatically behind the scenes. The **return** statement gives the value corresponding to each combination of bound variables (in this case x). Again, we have $k = \text{kind}(Y)$.

Unlike the first 3 operations in the Big 3+1, the “+1” operation does not produce a new FRP or Kind. Rather, this associates to each FRP and Kind a value called

its **expectation**. We only use an algebraic approach to this operation, denoting the expectation by the operator $\mathbb{E}$ mathematically and by `E` in `frplib`. This operation is consistent as well in that for every FRP X , we have:

$$\mathbb{E}(X) = \mathbb{E}(\text{kind}(X)). \quad (1.4)$$

1.5.3 Preview of the Big 3+1

Now, we will look at each of the Big 3+1 operations through the lens of our three interpretations. All of these will be discussed in complete detail in the corresponding chapters, but this is intended to orient you for what is to come.³⁷

³⁷Also see the Big 3+1 Cheatsheet.

In `frplib`, both FRP and Kind procedures are specified much as above, with two provisos: (i) We use the function definition syntax with a *decorator* before the `def` statement to define the procedure. The decorator is either `@frp` for an FRP procedure or `@kind` for a Kind procedure. The function definition can take new parameters, and defines the symbol of the given name *not* as a function but as an FRP or Kind, respectively. (ii) Instead of $\leftarrow$, we use the `yield` keyword and assign its value. For example, here are the procedures for Y and its Kind $k = \text{kind}(Y)$ shown above and defined in `frplib`, assuming the existence of an FRP X .

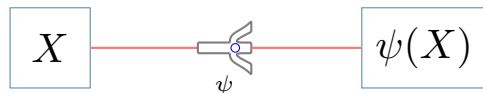
<pre><code>@frp def Y(): x = yield X return 2 * x</code></pre>	<pre><code>@kind def k(): x = yield kind(X) return 2 * x</code></pre>

Again, we exploit the function definition syntax here, but after these, Y is an FRP and k a Kind. While there is a procedural form for each of the Big 3 operations, the procedural syntax is particularly nice for *combinations* of operations.

TRANSFORMING WITH STATISTICS. We transform FRPs with statistics to answer questions, change the form of data, and to extract useful summaries of our measurements. The transform of an FRP X by a statistic ψ creates a new FRP whose value will be $\psi(v)$ when X 's value is v . Because of this relationship, we can reason about the transformed FRP even without knowing its value, and we can make predictions about it with the Kind of X .

Physically, we connect X to an empty FRP through an adapter that computes the statistic ψ . X 's value is sent as a signal over a wire from the All output port through the adapter and into the input port of the new FRP. When this connection is made, the originally empty FRP is automatically reconfigured as the transformed FRP and

its Kind is displayed. When X produces a value v , this activates the transformed FRP which then has value $\psi(v)$, though it is only observed when its own button is pushed. The wiring diagram for transforming X by ψ is



Algebraically, we transform an FRP X by a statistic ψ using either the *transform operator* $\hat{}$ or the *overloaded*³⁸ *evaluation operator* $()$ as we prefer. We write either $X \hat{\psi}$ or $\psi(X)$. We can analogously transform a Kind k by ψ , producing a new Kind denoted by $k \hat{\psi}$ or $\psi(k)$. Note that neither $\psi(X)$ nor $\psi(k)$ is actually evaluating the statistic at X or k ; rather, we are overloading the syntax for evaluation as an *operator*. In `frplib`, the transform operator $\hat{}$ takes a transformable object (e.g., an FRP, a Kind, or a statistic) on the left and a statistic on the right and produces the transform by the specified statistic. If `X` is an FRP, `k` is a Kind, and `psi` is a statistic, then (i) `X  $\hat{\psi}$` and `psi(X)` both give the transformed FRP, (ii) `k  $\hat{\psi}$` and `psi(k)` both give the transformed Kind. In mathematical work, we usually use the $\psi(X)$ notation, but in `frplib`, we use both forms `X  $\hat{\psi}$` and `psi(X)` frequently.

³⁸To “overload” an operator is to use the same operator in two different settings that are distinguishable from context.

Procedurally, the transform of an FRP X and a Kind k by ψ can be written:

FRP Y :

```
x ← X
return ψ(x)
```

Kind ℓ :

```
x ← k
return ψ(x)
```

The `frplib` versions of these procedures are almost the same, except we replace $\leftarrow$ with `yield` and an assignment and use the syntax of decorated function definition to define the objects.

```
@frp
def Y():
    x = yield X
    return psi(x)
```

```
@kind
def ell():
    x = yield k
    return psi(x)
```

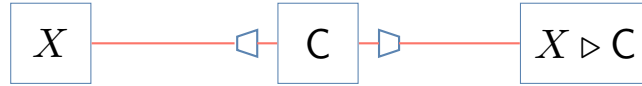
This defines FRP `Y` as the transform of FRP `X` by `psi` and Kind `ell` as the transform of Kind `k` by `psi`. Compare these to the earlier displays to see the common structure. The definition of `ell` may seem a little mysterious. Think of this as saying: “For any *arbitrary* value `x` from the Kind `k`, the corresponding value in our new Kind is `psi(x)`, and the weights will be tallied automatically for all the values that result.”

BUILDING WITH JOINS. We use *joins* to build more complicated systems from simpler pieces. Joins build in stages, selecting a random quantity for the next stage based on the value of all the earlier stages. The essential ingredient is a new device



that takes an input on the left and contains a switch that selects an FRP to activate based on that input. The output on the right “joins” the input value and the value of the selected FRP into one large tuple. This is called a *conditional FRP*.

To create a join, start with an FRP X and a conditional FRP C that accepts the possible values of X . We connect X ’s All output to the switch on the conditional FRP, and we wire the multiplexed output of the conditional FRP to a new, empty FRP that is automatically reconfigured. The output of the new FRP, which we call the *join* of X and C , is the joined value of both stages. The wiring diagram is:



Algebraically, we define joins with the *join operator* $\triangleright$, expressing the join of X and C as $X \triangleright C$. In `frplib`, we use the related $\gg$ operator, writing $X \gg C$, for appropriate objects X and C . Mathematically, we can view C as a function that maps input values to FRPs. We can also join Kinds. Here, we use a mapping c from input values to Kinds, called a *conditional Kind*. Starting with a Kind k and a conditional Kind c that accepts k ’s values as input, we can create the join $k \triangleright c$. In `frplib`, we write this as $k \gg c$. The join operations for FRPs and Kinds are consistent. For a conditional FRP C , we write $\text{kind}(C)$ for the conditional Kind that gives $\text{kind}(C(x))$ for each valid input x . Then:

$$\text{kind}(X) \triangleright \text{kind}(C) = \text{kind}(X \triangleright C). \quad (1.5)$$

Procedurally, $X \triangleright C$ and $k \triangleright c$ become

FRP Y :

```
x ← X
y ← C(x)
return x :: y
```

Kind ℓ :

```
x ← k
y ← c(x)
return x :: y
```

where $::$ is the join operation on tuples that concatenates and flattens them into one large tuple. (See Section 15.4 in Interlude F.) This defines FRP $Y = X \triangleright C$ and Kind

$\ell = k \triangleright c$. These procedures first get the value of X or every value of k and use that value to get an input-selected FRP (from C) or Kind (from c). The final value joins both tuples. Here, $C(x)$ is the FRP produced by C when X produces value x and $c(x)$ is the Kind produced by c for input value x . In `frplib`, these procedures look almost the same:

```
@frp                                @kind
def Y():                            def ell():
    x = yield X                     x = yield k
    y = yield C(x)                  y = yield c(x)
    return join(x, y)               return join(x, y)
```

Here, `join` is the `frplib` builtin function that implements the `::` operation on tuples.

Puzzle 15. Enter the following in the `frplib` playground, and examine the `Y` and `k` that result.

First, define three FRPs that we will use below.

```
X1 = frp(choice(0, 1))
X2 = clone(X1)
X3 = frp(choice(0, 1, 2))
```

The `clone` function returns a fresh FRP with the same Kind as the one it is given.

Then, define an FRP `Y` and a Kind `k` with following procedures:

```
@frp
def Y():
    x = yield X1
    y = yield X2 if x == 1 else X3
    return join(x, y)

@kind
def k():
    x = yield choice(0, 1)
    y = yield choice(0, 1) if x == 1 else choice(0, 1, 2)
    return join(x, y)
```

Notice that for `Y`, we are choosing which FRP to use for the second stage based on the value of `X1`. For `k`, we are choosing which Kind to use for *each* value of `k`.

Explain why these are joins and for what values of s and g these describe the FRP C and its Kind from the coin flip example in Section 1.4 and Figure 1.15.

CONSTRAINING WITH OBSERVATIONS. An *observation* means taking the value of an FRP as known, through observing it in reality or hypothetically. An observation gives us partial information about the data we measure from the random system. Before the observation, we could make predictions about the answers to our questions, and the new information in the observation allows us to *update* our predictions. If the observation is informative, our uncertainty will decrease and our predictions will improve.

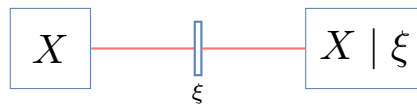
We use observations to account for new information that we obtain as a random system evolves. They are also incredibly useful for considering hypothetical information, questions like: “How would my predictions change if I were to observe ___?” Our methods for constraining with observations work the same for both real and hypothetical observations. Observations are defined in terms of *conditions*: Boolean statistics that return $\langle 1 \rangle$ for true and $\langle 0 \rangle$ for false. An *observation*, then, is the assertion that a specific condition is true. For example, in the coin flip example of Section 1.4, an observation is that the second flip is tails. The corresponding condition is the question “Is the second flip tails?” and the observation is the assertion that the answer is yes.

Physically, constraining with an observation creates a new FRP, related to the original, for which the observed information is a *fact*. We do this with a new piece of hardware, the *constraint processor*:



Each constraint processor is associated with a specific condition and only produces values for which the condition is true.

So, constraining an FRP X with the observation that a condition ξ is true has a wiring diagram that looks like



The constraint processor’s output is connected to the input port of a new, empty FRP which is automatically reconfigured to be the constrained FRP when the connection is made. If the value of X is consistent with the observation – as it will if it is based on an actual observation derived from X – the constraint processor passes

this value through unchanged. If not, the constraint processor generates a new value based on X 's Kind that is consistent with the observation. We only require that the observation be *possible* for X ; if not, the constrained FRP is the empty FRP.

Algebraically, we constrain with observations using the *given operator* $|$, which is read as “given.” On the left side of this operator, we have an FRP or a Kind to be constrained. To the right of the operator, we can give either a Boolean expression that specifies the observation or simply a condition (like ξ). So, we can write $X | (\xi(X) = 1)$ or $X | \xi$. This is flexible syntax. For instance, if X has dimension > 2 , we can write $X | (\text{proj}_1(X) < 0 \wedge \text{proj}_2(X) > 0)$. The observation in this case is that the first component of X is negative and the second component is positive.³⁹ This corresponds to the condition $\zeta(x) = \text{proj}_1(x) < 0 \wedge \text{proj}_2(x) > 0$, in which case we could write the previous FRP as $X | \zeta$. Remember that a constrained FRP like $X | \xi$ is a *new* FRP; its Kind will differ in general from the Kind of X . When the observational constraint is consistent with X 's value, both X and the constrained X will have the same value, but this need not be true for hypothetical observations.

³⁹The $\wedge$ operator is logical *and*; the $\vee$ is logical *or*.

We can also constrain Kinds with observations using the same operator and syntax. If k is a Kind, then $k | \xi$ is the Kind constrained by the observation that ξ is true. This is the same as $\text{kind}(U) | (\xi(U) = 1)$ where U is an FRP with Kind k . With Kinds, we tend to use conditions on the right side of the given operator, but we can easily use the short-hand form of anonymous functions with nice effect.⁴⁰ For example, $k | (\text{proj}_1(\blacksquare) < 0 \wedge \text{proj}_2(\blacksquare) > 0)$ is the same as $\text{kind}(U) | (\zeta(U) = 1)$ for the condition ζ defined in the previous paragraph. The $\blacksquare$ in the anonymous function (read “hole”) is filled with the single argument to the function, the tuple at each leaf node of k .

⁴⁰See page 606 in Chapter 11 of Interlude F.

In `frplib`, we use the same notation. The given operator $|$ takes a condition on the right and an FRP or Kind on the left. We can write conditions as expressions like the anonymous function above, using `__` as “hole” in lieu of $\blacksquare$. For instance, given an FRP `X`, a Kind `k`, and a condition `xi`, we can write

- `X | xi`
- `k | xi`
- `X | And(Proj[1] > 0, Proj[2] < 0)`
- `k | And(Proj[1](__) > 0, Proj[2](__) < 0)`
- `X | (Sum(2 * __ + 3) < 10)`
- `k | (Sum(2 * __ + 3) < 10)`

In fourth line, we use a statistic combinator to define the statistic ζ without naming it. This is equivalent to, and more concise than, the version with “hole” in the fifth

line. In the last two cases, we use `--` “hole” in an expression. The `|` operator has low precedence in Python, so any expression needs to be enclosed in parentheses.

Note again we have consistency between the FRP and Kind versions of this operation. For any valid observation `obs`, we have

$$\text{kind}(X \mid \text{obs}) = \text{kind}(X) \mid \text{obs}, \quad (1.6)$$

assuming that X is fresh.

Procedurally, we impose the constraint with a `given` line in the procedure that states the constraint. For example, if the observation is defined by condition ξ , we can define a constrained FRP and Kind with the following procedures:

FRP Y : $x \leftarrow X$ <code>given $\xi(x)$</code> <code>return x</code>	Kind ℓ : $x \leftarrow k$ <code>given $\xi(x)$</code> <code>return x</code>

This defines $\text{FRP } Y = X \mid \xi$ and $\text{Kind } \ell = k \mid \xi$. If the Boolean value⁴¹ in the `given` line is true, the procedure continues; otherwise, it stops with an empty result. In `frplib`, observational constraint procedures have the same form, but we use `yield given(b)` instead of `given b`. The analogous procedures in `frplib` are:

⁴¹In the extended sense: true, false, $\langle 1 \rangle$, or $\langle 0 \rangle$

<pre style="color: #008000;">@frp def Y(): x = yield X yield given(xi(x)) return x</pre>	<pre style="color: #008000;">@kind def ell(): x = yield k yield given(xi(x)) return x</pre>

where Y is $X \mid xi$ and ell is $k \mid xi$.

PREDICTING WITH EXPECTATIONS. The *expectation* of an FRP or Kind is our prediction of the FRPs value. The expectation depends only on the Kind, not on the particular value of any specific FRP with that Kind. It is how we extract our “best guess” (in a particular sense) of the value from the Kind. As described above, we denote the expectation by the operator $\mathbb{E}$ mathematically and by `E` in `frplib`, e.g., $\mathbb{E}(X)$ and `E(X)` or `E(k)`.

1.5.4 Learning from Data: The Likelihood Principle

In the coin flips example, we encountered a common situation, where our model for the system has *multiple* Kinds that might describe the Data FRP. These Kinds are often collected into a family that is indexed by one or more **parameters**. Each choice of parameters uniquely determines the Kind of the Data FRP, and given a specific choice of parameters we can make predictions about the answers to our questions. But in some situations, we observe data *without knowing which Kind in the family* describes the Data FRP. Our goal then is to learn as much as possible about the true parameters from the data. For the coin flips, the coins' weighting was the unknown parameter, and we can learn that parameter by observing the flips.

There are various ways to learn the parameters from the data, and the **likelihood principle** says that all the information *from the data* about the parameters is contained in the *likelihood*. The **likelihood** of a Kind in our family is the canonical weight for that Kind of the branch associated with the *observed value of the data*. This gives us a tool for learning from data when our description of a system is incompletely specified. It is the basis for statistical inference.

Key Take Aways

Almost everything we will do with probability theory is built on four basic operations, (the **Big 3+1**) and one key principle (the **likelihood principle**).

The “3” in the Big 3+1 operate on FRPs to produce new FRPs by connecting output ports to input ports in several ways: (1) transforming the output of an FRP by some algorithm (statistics), (2) generating random outcomes contingent on earlier outcomes (joins), and (3) accounting for partially observed information (observations). All of these operations can also act on Kinds to produce new Kinds, and the operations on FRPs and Kinds are consistent with each other. We saw three interpretations of these operations: Physical, Algebraic, and Procedural.

The “+1” in the Big 3+1 is operation (called expectation) takes an FRP (or Kind) and yields a value that represents our prediction of the FRP's value.

The Big 3+1 are

- Transforming with Statistics
- Building with Joins
- Constraining with Observations

- Predicting with Expectations.

In addition, many methods for learning about our model from observed data are based on the *likelihood principle*. This says that all the information *from the data* about the parameters is contained in the *likelihood*. The *likelihood* of a Kind in our family is the canonical weight for that Kind of the branch associated with the *observed value of the data*.

Chapter Checkpoints

After reading this Chapter you should be able to:

- Describe what an FRP is and what each of the words fixed, random, and payoff in the name refers to.
- Explain the structure of a valid FRP *Kind*.
- Use the `frplib` playground to examine the values of FRPs of a given Kind and run demos with large numbers of such FRPs.
- Guess to within a reasonable margin of error the proportions of each value observed in a large demo of FRPs of a specified Kind.
- Explain in rough terms what the weights on an FRP's Kind indicate.
- Explain in broad strokes our framework for using FRPs and Kinds to describe and analyze random systems.
- Explain at a high level both probabilistic and statistical inference.
- List the Big 3+1 operations on FRPs and Kinds.
- Describe the three interpretations of the Big 3 operations.
- Explain how the Big 3+1 are consistent in their actions on FRPs and their Kinds.

Transforming with Statistics

2

Chapter

Contents

2.1	Statistics and Data Processing	84
2.1.1	Statistics in <code>frplib</code>	95
2.2	Transformed FRPs	115
2.3	Transformed Kinds	134
2.4	Projections and Marginals	153
2.5	Extended Examples	161

2.1 Statistics and Data Processing

We use observations of random systems' evolution for learning, discovery, prediction, and effective decision-making. For realistic problems, we typically collect a vast amount of data from the system – as a *high-dimensional tuple* – and *process* those data with targeted algorithms to extract the information and features that are relevant to our particular goals. The data-processing algorithms that do this are collectively called statistics. We use statistics for a variety of purposes: to transform the data into a more useful form, to compute summaries to help us understand it, or to extract features that answer our questions.

Consider a simple system in which a coin is flipped five times in succession. If we encode flips with 0 for tails and 1 for heads, this system is represented by an FRP whose possible values – the data we collect from the system – are 5-tuples whose components are 0s and 1s, like $\langle 1, 0, 0, 1, 1 \rangle$ or $\langle 0, 1, 0, 0, 1 \rangle$. There are various ways we might be interested in processing these data.

One important type of processing is to extract a feature of the data that answers a question. For instance, to answer the question “how many heads appear in those five flips,” we would use the statistic **Sum** that takes a tuple as input and returns the sum of its components, e.g., $\text{Sum}(\langle 1, 0, 0, 1, 1 \rangle) = 3$ and $\text{Sum}(\langle 0, 1, 0, 0, 1 \rangle) = 2$. **Sum** is a function that accepts tuples of numbers and returns a scalar (i.e., a 1-dimensional tuple of numbers). Mathematically, we can express it as

$$\text{Sum}(\langle x_1, x_2, \dots, x_n \rangle) = \sum_{i=1}^n x_i,$$

for any $n \in [0..)$ and any n -tuple $\langle x_1, x_2, \dots, x_n \rangle$. (In **frplib**, this statistic is called **Sum**.) If we restrict the inputs to tuples of 0s and 1s, then we can see that the **Sum** statistic simply *counts the components in the tuple that are 1*. Thus for these data, the **Sum** statistic *embodies* the question “how many heads appear” because given any tuple of flips, it returns the answer to the question for that tuple.

Puzzle 16. What are two more questions we might want to ask about this sequence of five coin flips? Describe statistics that answer these questions. They should accept a 5-tuple as input and return as output a tuple that answers the corresponding question for that input value.

Another common type of processing is to transform data to another useful form.

An example with the coin flips might be a statistic that changes our encoding of the flips so that tails are labeled with 1 and heads with 0. For example, this statistic converts a data tuple $\langle 1, 0, 0, 1, 0 \rangle$ to the tuple $\langle 0, 1, 1, 0, 1 \rangle$. Mathematically, we can write this statistic, which we name t , for any $n \in [0..)$ as

$$t(\langle x_1, x_2, \dots, x_n \rangle) = \langle 1 - x_1, 1 - x_2, \dots, 1 - x_n \rangle.$$

In `frplib`, we can write this statistic even more simply by using statistic combinator expressions to define new statistics:

```
pgd> t = 1 - __
pgd> t(1, 0, 0, 1, 0)
<0, 1, 1, 0, 1>
pgd> t(0, 1, 1, 0, 1)
<1, 0, 0, 1, 0>
```

To help understand the expression `1 - __`, note that we can express t mathematically as the combination of two other statistics. Define

$$\begin{aligned} c(\langle x_1, x_2, \dots, x_n \rangle) &= \langle 1, 1, \dots, 1 \rangle \\ h(\langle x_1, x_2, \dots, x_n \rangle) &= \langle x_1, x_2, \dots, x_n \rangle \end{aligned}$$

then

$$t = c - h,$$

which means t 's output subtracts h from c component by component. In an `frplib` statistic expression, a bare constant like 1 corresponds to a statistic that always returns that constant for any input.¹ The statistic `__` (two adjacent underscores) is called `hole` and just returns as output any input it is given. While simple, we use it often in expressions to capture the original input to the new statistic. The `-` operator acts on two statistics as a statistic combinator. If ψ and φ are statistics, then $\psi - \varphi$ is the statistic that passes the same input to both ψ and φ and then subtracts their output tuples component-wise. The two statistics must either have the same dimension, in which case

$$\begin{aligned} \psi(\langle x_1, x_2, \dots, x_n \rangle) &= \langle y_1, y_2, \dots, y_d \rangle \\ \varphi(\langle x_1, x_2, \dots, x_n \rangle) &= \langle z_1, z_2, \dots, z_d \rangle \end{aligned}$$

¹See the documentation on the `Constantly` statistic factory.

implies

$$(\psi - \varphi)(\langle x_1, x_2, \dots, x_n \rangle) = \langle y_1 - z_1, y_2 - z_2, \dots, y_d - z_d \rangle,$$

or one of ψ and φ must be a scalar statistic, in which case its scalar output is used with every component of the other statistic's output: e.g.,

$$\begin{aligned}\psi(\langle x_1, x_2, \dots, x_n \rangle) &= \langle y \rangle \\ \varphi(\langle x_1, x_2, \dots, x_n \rangle) &= \langle z_1, z_2, \dots, z_d \rangle\end{aligned}$$

implies

$$(\psi - \varphi)(\langle x_1, x_2, \dots, x_n \rangle) = \langle y - z_1, y - z_2, \dots, y - z_d \rangle.$$

Thus, `1 - __` in `frplib` is the same as $c - h$ in mathematics, and both equal t .

A related example of a statistic for the coin flips is to extract *both* the number of heads and number of tails. This transformation gives us a summary of the data that keeps some information (the counts) and loses other information (like the order of the flips). This statistic gives a 2-dimensional output, and we can express it in several ways. We have seen that the statistic `Sum` counts the number of heads, so necessarily the statistic `Dim - Sum` counts the number of tails, where `Dim` is the statistic that gives the dimension of its input tuple. The statistic s that gives both counts maps a tuple x to a 2-tuple that concatenates the tuples `Sum(x)` and `Dim(x) - Sum(x)`. We can write this in terms of the *fork* operator defined in Section 15.5:

$$s = \text{Sum} \curlyvee (\text{Dim} - \text{Sum})$$

which means

$$s(x) = \text{Sum}(x) :: (\text{Dim}(x) - \text{Sum}(x))$$

concatenating the statistics' output tuples, or equivalently

$$s(\langle x_1, x_2, \dots, x_n \rangle) = \left\langle \sum_{i=1}^n x_i, n - \sum_{i=1}^n x_i \right\rangle.$$

In `frplib`, we write this as `Fork(Sum, Dim - Sum)`, using the `Fork` statistic combinator, where $\text{Fork}(\psi_1, \psi_2, \dots, \psi_r) = \psi_1 \curlyvee \psi_2 \curlyvee \dots \curlyvee \psi_r$.

```
pgd> s = Fork(Sum, Dim - Sum)
pgd> s(1, 0, 0, 1, 0)
<2, 3>
pgd> s(0, 1, 1, 0, 1)
<3, 2>
```

Note how the combinators `-` and `Fork` take *statistics* as arguments and return a new statistic.

Puzzle 17. Define a statistic (using mathematics, `frplib`, code, or even words) that gives the *difference* between the numbers of heads and tails in the five flips.

One answer: `Dim. - Sum * 2`. In `frplib`, we could define it as `2 * Sum - Dim`. Do you see why?

In general, we will use the word *value* with a specific meaning throughout. A value is a *tuple*, and because we almost exclusively use numeric FRPs here, by default, a value means a *tuple of numbers*. If a value is a tuple of dimension 1, we elide the distinction between the tuple and its single, numeric component, treating them interchangeably. We call such values *scalars*.

Definition 3. We will use the word **value** to mean a tuple of dimension ≥ 0 , and unless otherwise indicated, we will assume that this is a *tuple of numbers*.

A *statistic* is a function that takes a value (tuple of numbers) as input and returns a value (tuple of numbers) as output. Particular statistics may restrict the set of inputs they accept, so we speak of the *valid inputs* to a particular statistic. For instance, some statistics accept input tuples of particular dimensions, and some restrict the range of the input tuples' components. For simplicity, we impose one additional requirement on all the statistics we use: that all inputs of the equal dimension n return outputs of the same dimension n' , where n and n' themselves need not be equal. This makes it easier to transform FRPs and Kinds with statistics and causes no real practical loss. In cases where we want to simulate outputs of different dimension, we can pad the output tuple with a special numeric value called `nothing` in `frplib`.² The value `nothing` displays as an empty square $\square$ in `frplib` tuples.

```
pgd> nothing * 2
nothing
pgd> tup(1, 2, 3, nothing, nothing)
<1, 2, 3,  $\square$ ,  $\square$ >
```

²`nothing` is defined so that any operation on `nothing` and any other number produces `nothing`.

Here, the function `tup` creates a `frplib` tuple object³ with specified components. It is usually unnecessary to use `tup` as `frplib` will convert the values of FRPs and Kinds and outputs of Statistics automatically. But when using the transform operator on a tuple or when doing certain vector operations that `frplib` tuples support, we will want to use `tup` to ensure that we have a special `frplib` tuple.

³This extends and is distinct from Python's built-in tuples. Always use `tup` instead of Python's `tuple`.

Notational Convention: Function Evaluation. For a function ψ that takes an n -tuple as input, we are flexible with how its arguments are written when evaluating the function.

If $v = \langle v_1, \dots, v_n \rangle$, we treat the following as equivalent and interchangeable:

$$\psi(\langle v_1, \dots, v_n \rangle) \qquad \psi(v) \qquad \psi(v_1, \dots, v_n)$$

using whichever form is clearest and most convenient at any moment.

The third form treats a function that takes a single n -tuple as equivalent to a function that takes n numeric arguments. It allows us to drop the extraneous $\langle \rangle$ delimiters in the first form. All three forms work with statistics in `frplib`.

This convention is discussed in detail in Chapter 15 of Interlude F.

Definition 4. A **statistic** is a function that takes *values* in some set as inputs and returns a *value* as output. A value in the domain of the statistic is called a *valid input*. We require that all valid inputs of equal dimension produce output values of the same dimension.

If a statistic accepts values of dimension n and, given such, returns a value of dimension n' , we say that the statistic has **type** $n \rightarrow n'$. We call n the **codimension*** and n' the **dimension**. If a statistic has dimension 1 for all valid inputs, we say that it is a *scalar statistic*. If a statistic can accept input tuples of various lengths, then it will have more than one distinct type, but a statistic must have at most one type for each valid codimension.

In many cases, we will give statistics meaningful (or at least mnemonic) names. The names of most built-in statistics in `frplib` start with a capital letter. We will denote generic statistics by lowercase Greek letters, especially ψ (“psi”, pronounced like sigh), φ (“phi” pronounced fee or fi), ξ (“xi”, pronounced zigh or ksee), and ζ (“zeta”). Whenever you see ψ , φ , ξ , or ζ in this book, you know that they represent statistics. Finally, it is sometimes convenient to specify statistics as anonymous functions, such as $\langle x, y \rangle \mapsto 3x + 2y$ or $(3\blacksquare_1 + 2\blacksquare_2)$; see Chapter 11 in Interlude F.

In `frplib`, statistics are specially decorated Python functions.

A particular statistic may or may not have a single unique codimension. For example, a statistic that takes a point in the plane $\langle x, y \rangle$ and returns its distance from the origin $\sqrt{x^2 + y^2}$ has codimension 2 and dimension 1, so is of type $2 \rightarrow 1$. A statistic that maps two points in space $\langle x_1, y_1, z_1, x_2, y_2, z_2 \rangle$, encoded as a 6-tuple, to the midpoint between them $\langle \frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}, \frac{z_1+z_2}{2} \rangle$ has type $6 \rightarrow 3$. However,

*Pronounced “ko-dimension”.
Some call this *arity*.

many statistics naturally accept tuples of various lengths. For instance, the statistic that reverses a list, mapping $\langle x_1, x_2, \dots, x_n \rangle$ to $\langle x_n, x_{n-1}, \dots, x_1 \rangle$, has type $n \rightarrow n$ for every integer $n \geq 0$, and the statistic **Max** that computes the maximum of the components, mapping $\langle x_1, x_2, \dots, x_n \rangle$ to $\max(x_1, x_2, \dots, x_n)$, has type $n \rightarrow 1$ for every integer $n \geq 0$.⁴

⁴We define the maximum of an empty tuple to be $-\infty$.

A statistic represents a computation or algorithm that we can run on data. As we encounter different types of objects, it is worth keeping in mind that any function that maps values to values is a statistic.⁵ The most common uses of statistics are to extract answers to questions from the data and to transform data into a different, more useful format. In the former case, we think of the statistic as embodying the question in the sense that for any input, the statistic’s output *answers* the question for that input. We now turn to an extended example to illustrate this idea.

⁵Subject to the minor requirements in the definition.

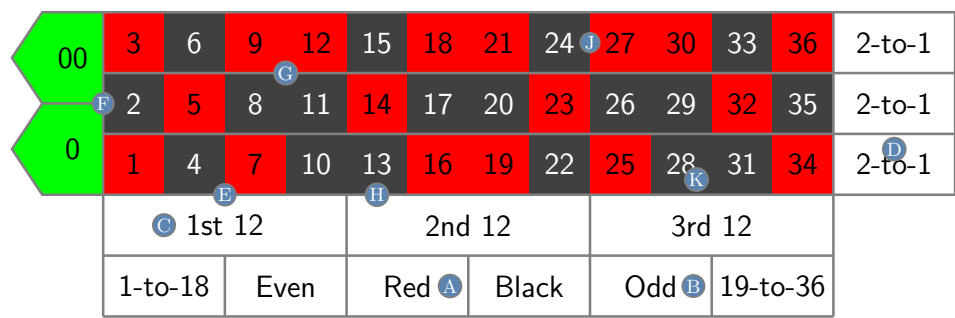


FIGURE 2.1. Betting table for roulette. The pocket numbers are oriented with top to the left, so what are called “columns” go from left to right. The labeled disks represent chip placements corresponding to plays in Table 2.1.

Example 2.1 Roulette, Part 1

A Nevada roulette wheel has 38 pockets along its edge that can catch and hold a small metal ball. Each pocket has a color and a unique number: two pockets are colored green with numbers 0 and 00 and the remaining pockets are numbered 1 through 36, with half colored black and half colored red.

A game of roulette begins with players making their bets by placing chips worth some amount of money on a betting table that displays an array of numbered, colored squares and surrounding, labeled tabs, as shown in Figure 2.1. Each chip placed specifies a *set of pockets* (based on where the chip is placed) and an amount bet (based what the chip is worth). A roulette bet is called a “play” in the official lingo. The players bet and then the ball is released into the

Play	Winning Pockets	Payoff	Sample
Even Money	A set of eighteen numbers: red, black, even, odd, 1–18, 19–36	1 to 1	A, B
Dozen	One of 1–12, 13–24, 25–36	2 to 1	C
Column	Twelve numbers in one “column”	2 to 1	D
Six Line	Six consecutive numbers (two “rows”)	5 to 1	E
Top Line	00, 0, 1, 2, 3	6 to 1	F
Corner	Four numbers that share a corner	8 to 1	G
Street	Three consecutive numbers (one “row”)	11 to 1	H
Split	A pair of adjacent numbers	17 to 1	J
Straight	A single number	35 to 1	K

TABLE 2.1. Common bets in roulette. Each such bet wins when the ball lands in a particular set of pockets. If a player loses a play, the amount bet is forfeit to the “house.” If the player wins, then the casino returns the bet plus a payoff that is a multiple of the amount bet. The last column gives the labels of chip placements in Figure 2.1 that exemplify the play.

spinning wheel. The ball rolls around the wheel and eventually comes to rest in one of the pockets. A play whose set of pockets includes the one with the ball wins; the player keeps the amount bet and receives a payoff that is a multiple of that amount, depending on the play. A play whose set of pockets does *not* include the one with the ball loses, and the player forfeits the amount bet.

On the betting table in Figure 2.1, squares numbered 1 through 36 are colored red or black and the tabs 0 and 00 are colored green, matching the wheel. The remaining regions specify a set of pockets like “Red” for all the red pockets or “2nd 12” for pockets 13–24. The basic plays are described in Table 2.1 in order of increasing payoff. For instance, a \$1 Top Line play will win \$6 if the ball stops in pockets 00, 0, 1, 2, or 3 or will lose \$1 otherwise. As the number of winning pockets for a play decreases, the payoff increases. The Sample column in the table indicates which chip placements in Figure 2.1 match that play.

In the `frplib` playground, load the example code as follows:

```
pgd> from frplib.examples.roulette import roulette
```

This imports an object `roulette` that has everything we will need for this example. Calling `roulette` as a function with no arguments returns a fresh

FRP representing a single spin of the roulette wheel, which we will use in Part 2 (Example 2.2). Here, we will use statistics that represent all of the standard plays, which are available as attributes of the `roulette` object. (Look at `roulette.plays` to see a list of these attribute names.)

As usual, we will use numeric FRPs, so we need a number for each possible outcome of a Roulette wheel spin. We will use the pocket number itself for all pockets except for pocket 00 to which we assign the value -1.

The questions driving our analysis of roulette are about the outcome of various plays and how much money will be won or lost on each. For instance: What is our payoff from a bet on Red? On Top Line? On a combination of the two? To every possible play we associate a statistic that accepts the number of the pocket in which the ball lands and returns the payoff from that play for that pocket. Each such statistic embodies the question “what is the payoff” for that particular play. Later, in Example 2.2, we will use this to assess which of the standard plays are best and worst and to devise a betting strategy for a single spin. (We might also wonder what our best betting strategy is for choosing plays on multiple spins, a question we take up in a later chapter.)

For now, we study the statistics associated with various plays. Plays like Red and Top Line are straightforward:

```
pgd> roulette.red
A Statistic 'red' that represents a $1 play on all Red pockets.
It expects 1 argument (or a tuple of that dimension) and returns a scalar.
pgd> roulette.top_line
A Statistic 'top_line' that represents a $1 play on pockets 00, 0, 1, 2, 3.
It expects 1 argument (or a tuple of that dimension) and returns a scalar.
pgd> roulette.red(3)
<1>
pgd> roulette.red(4)
<-1>
pgd> roulette.red((12,)) # Passing a 1-dimensional tuple is fine too
<1>
pgd> roulette.top_line(-1)
<6>
pgd> roulette.top_line(2)
<6>
```

```
pgd> roulette.top_line(17)
<-1>
```

Because statistics are just functions that take a value and return a value, Both `roulette.red` and `roulette.top_line` are statistics and thus are functions that take a pocket (as a number or a 1-dim tuple) and return the payoff of a \$1 bet if the ball stops in the specified pocket. The payoff is positive when you win and negative when you lose; for instance, you lose \$1 on Red when the ball lands in (black) pocket 4.

Plays like Straight, Corner, and Split offer many choices. For example, to specify a Straight play, we need to choose a pocket to bet on. The pocket is a *parameter* that determines the straight play. So `roulette.straight` is not itself a statistic but rather what we call a *statistic factory*: a function that returns a statistic based on specified parameters.

```
pgd> roulette.straight
Statistic factory representing $1 straight plays on the specified pocket
pgd> roulette.straight(16)
Statistic 'straight_16' that represents a $1 play on pocket 16.
It expects 1 argument (or a tuple of that dimension) and returns a scalar.
pgd> s16 = roulette.straight(16)
pgd> s16(16)
<35>
pgd> s16(17)
<-1>
```

Because `roulette.straight(16)` is a statistic, we could call it directly with, say, `roulette.straight(16)(17)`, but it is often clearer to give the statistic a name, like `s16`, and use that. Alternatively, if we have an `frplib` tuple, we can use the transform operator `^` to pass that value through the statistic:

```
pgd> pocket = tup(16)                # A 1-dimensional frplib tuple
pgd> pocket ^ roulette.straight(16)  # same as roulette.straight(16)(pocket)
<35>
```

We will frequently use `^` in this way when transforming FRPs and Kinds with statistics produced by factories, as we will see in the next two sections.

The Split play has a factory determined by two parameters, the adjacent pockets in numerical order:

```
pgd> roulette.split
Statistic factory representing $1 split plays on the adjacent pockets
(first smaller than second)
pgd> roulette.split(15, 18)
Statistic 'split_15_18' that represents a $1 play on adjacent pockets 15 and 18.
pgd> sp15_18 = roulette.split(15, 18)
pgd> sp15_18(16)
<-1>
pgd> sp15_18(18)
<17>
```

All of the foregoing examples have been \$1 bets, but in Roulette, we can bet any amount. To do this, we multiply the \$1 play statistics by the amount of the bet we wish to make. Multiplying a statistic by a number acts as a *statistic combinator* that creates a new statistic from the old one. For example, in the following, `r25` represents a \$25 bet on Red and `s16_10` represents a \$10 bet on Straight 16:

```
pgd> r25 = 25 * roulette.red
pgd> s16_10 = 10 * roulette.straight(16)    # == 10 * s16
pgd> r25(3)
<25>
pgd> r25(4)
<-25>
pgd> s16_10(16)
<350>
pgd> s16_10(17)
<-10>
```

Notice that `10 * s16(16)` is the same *value* as `(10 * s16)(16)` and as `s16_10(16)`, but `s16(16)` is a value that we scale, whereas `10 * s16` is a *statistic* that we evaluate at 16.

In Roulette, we are not limited to betting one standard play; instead, we can combine multiple plays into a single bet. Again, we create the corresponding statistics with statistic combinators, using both scaling of statistics by numbers

to select the bet size and addition with the `+` combinator to incorporate multiple plays into a single bet. For instance, a bet that combines a \$10 even play, a \$5 corner play on (25, 26, 28, 29), a \$20 straight play on 4, and a \$50 column play on the second column is represented by the following statistic:

```
pgd> comb = 10 * roulette.even + 5 * roulette.corner(25) +
           20 * roulette.straight(4) + 50 * roulette.column(2)
pgd> comb(26)
<130>
pgd> comb(4)
<655>
pgd> comb(-1)
<-85>
pgd> comb(32)
<85>
```

This is a special case of the `+` statistic combinator. As we will see, adding two statistics of the same dimension gives a new statistic that computes the sum of the original statistics.

Puzzle 18. Experiment with the Roulette statistics in the `frplib` playground. Devise various bets and create statistics that represent those bets. Evaluate your statistics on various pockets to confirm they act as you expect.

CONDITIONS. An important special case is a scalar statistic that tests whether its input satisfies some property. These statistics return an indication of whether the property is satisfied: true or false, yes or no. In practice, we use the value 1 for true and the value 0 for false.⁶ This type of statistic is called a **condition**.

Definition 5. A **condition** is a statistic that returns a Boolean value.

A Boolean value is either $\top$, read “true” or “top”, or $\perp$, read “false” or “bottom”. In practice, we identify $\top$ with the value 1 and $\perp$ with the value 0, but we distinguish them notationally when we need to emphasize that Boolean and number are conceptually distinct types.

We abuse notation a little by letting $\top$ and $\perp$ also denote *constant* conditions. In this case, $\top$ denotes the constant condition that returns true for *any* input value and

⁶Remember that we identify scalars like 1 and 0 with the 1-dimensional tuples $\langle 1 \rangle$ and $\langle 0 \rangle$.

$\perp$ denotes the constant condition that returns false for any input value.

In practice, conditions are *indicator functions* as described in Chapter 12 in Interlude F. We thus often write conditions using *Iverson braces* as described on page 614. Recall that a Boolean expression in Iverson braces gives the value 1 when the expression is true and the value 0 otherwise. For example:

- $\text{isEven}(x) = \{x \bmod 2 = 0\}$ is the condition with codimension 1 that tests whether its input is an even integer;
- $\xi(x) = \{\text{proj}_2(x) = 1\}$ is the condition that tests whether the second component of its input is the number 1, which has type $n \rightarrow 1$ for all $n \in [2..)$;
- $\{\text{proj}_2(\blacksquare) = 1\}$ is the same condition as in the previous item but written as an *anonymous function* (see Chapter 11), where $\blacksquare$ is a hole filled by the input;
- $\zeta(y) = \{\text{Sum}(x) > 2 \wedge \text{proj}_1(x) < 0\}$ tests whether the sum of all input components is bigger than 2 *and* the first component is negative.

We use $\wedge$ and $\vee$ for logical-and and logical-or in Boolean expressions.

2.1.1 Statistics in `frplib`

Because statistics are just functions, we already know how to define and work with them in the abstract. Yet it is practically useful to understand the various tools that `frplib` provides for defining and using statistics. In `frplib`, statistics are objects that act like specially decorated Python functions. We can call them like an ordinary function; we can query their properties and documentation; and we can use them in combinator expressions to produce new statistics. There are several ways to produce statistics in `frplib`:

- directly using *built-in statistics*,
- calling a *statistic factory* to create a statistic with specified properties,
- calling a *statistic combinator* to combine one or more statistics,
- creating a *custom statistic* by decorating a Python function.

We will now discuss each of these in turn. Note that the `frplib` playground `info` system offers help on these categories with topics '`statistics-builtins`', '`statistics-factories`', '`statistics-combinators`', and '`statistics-custom`'. You can also call the `cheatsheet()` function to see the `frplib` Cheatsheet. Follow along with the examples below to get a feel for how to create and use these.

A wide variety of **built-in statistics**, are pre-defined in `frplib` to handle frequently used computations. Examples include: (i) summary statistics like `Sum`, `Mean`, `Min`, `Max`, `Median`, and `StdDev`; (ii) standard mathematical functions like exponentials and logarithms (`Exp`, `Log`, `Log10`, `Log2`), roots (`Sqrt`), Euclidean length (`Abs`),

discrete math (`Floor` and `Ceiling`), and trigonometric functions (`Sin`, `Cos`, `Tan`); (iii) utility functions like `FromDegrees`, `FromRadians`, `Cases`, `Bag`, `Diff`, `Diff`s, and `Ascending`; and (iv) logical functions used for testing Boolean conditions like `And`, `Or`, `Not`, `All`, `Any`; and (v) *hole* `__` and its relatives `Id` and `Scalar`.⁷ Note that all of these (except *hole*, which is unique) start with a *capital letter*. There are some Python built-ins with similar names (e.g., `sum`, `max`, `and`), but these start with lowercase letters and are *not* `frplib` statistics.

⁷`Scalar` is like *hole* but raises an error if the input tuple has dimension > 1 .

We can query a statistic’s properties. Printing the statistic directly in the playground gives a short description:

```
pgd> Max
Monoidal Statistic 'max' that returns the maximum of all components
of the given value. It expects a tuple and returns a scalar.
```

(We’ll see what “Monoidal” means later, disregard it for now.) The name in quotes indicates the function from which this statistic object was derived. Calling `info` on any built-in statistic will give more detailed documentation.

```
pgd> info(Max)
# ... long description with examples (omitted here)
```

We can see its codimension, dimension, and type

```
pgd> codim(Max)
(0, inf)
pgd> dim(Max)
1
pgd> typeof(Max)
'[0..) -> 1'
```

The `codim` shows that `Max` accepts tuples of any size, from 0 to infinity; the `dim` shows that it always returns a scalar, and the `typeof` shows both these facts.

As the name suggests, a **statistic factory** is a function that builds and returns a *statistic* specified by the parameters passed as arguments to the function. A simple but often useful statistic factory is `Constantly`. This takes a value as an argument and returns a statistic that outputs that value for *every input*.

```
pgd> always123 = Constantly(1, 2, 3)
pgd> always123(10)
<1, 2, 3>
```

```
pgd> always123(0, 1, 10, 100)
<1, 2, 3>
pgd> always123()
<1, 2, 3>
```

Note that we could use the returned statistic directly without naming it, but if we plan to call it, such as with `Constantly(1, 2, 3)(10)` in lieu of `always123(10)`, the name makes the call easier to parse. Alternatively, we can use the transform operator `^` with an `frplib` tuple on the left and a statistic on the right. So

```
pgd> tup(5, 10, 20) ^ Constantly(1, 2, 3)
<1, 2, 3>
```

is the same as `Constantly(1, 2, 3)(5, 10, 20)`. This is often convenient for evaluating statistics returned by factories without naming them, and we will use it extensively when using such statistics to transform FRPs and Kinds.

Another example of a factory is `Append`. This takes a value and returns the statistic that appends the given value to its input:

```
pgd> phi = Append(1, 5, 25)
pgd> phi(0)
<0, 1, 5, 25>
pgd> phi(-2, -1, 0)
<-2, -1, 0, 1, 5, 25>
```

The statistic factory `Get` takes a fixed Python object that is indexable with `[]`, like a list or a dict, and returns a statistic that uses its input value as a key. (One can control exactly how that key is used with various options to `Get`).

```
pgd> zeta_a = Get([0, 10, 20, 30])
pgd> zeta_a(2)
<20>
pgd> zeta_d = Get({'-1': 10, '0': 90, '2': 999})
pgd> zeta_d(2)
<999>
```

An extremely common form of statistic are those that extract one or more components of a tuple. We call these **projection statistics**, and the `Proj` statistic factory is used to create them. While we can pass the desired components as arguments to a function (e.g., `Proj(2)`), we more commonly use `[]` with this factory to remind us of the indexing operation:

```
pgd> tup(1, 2, 3, 4, 5) ^ Proj[4]
<4>
pgd> foo = Proj[1, 5]
pgd> foo(1, 2, 3, 4, 5)
<1, 5>
```

Note that projection statistics are *1-indexed* like mathematical tuples, not 0-indexed like Python lists, so `Proj[1]` extracts the first component. Projection statistics are discussed in detail in Section 2.4.

The most fundamental statistic factories is the function `statistic` that converts an ordinary Python function to an `frplib` statistic object. The factories `scalar_statistic` and `condition` are similar but are specialized to produce particular types of statistics. We will discuss these in detail below when we cover custom statistics because they are most often used in that capacity.

Statistic combinators let us build new statistics from one or more other statistics. A fundamental combinator is *composition*⁸ in which the output of one statistic is passed as input to another. There are two equivalent ways to specify this, corresponding to the *pipeline* (§) and *syntactic* (◦) orders; for instance,

⁸See Interlude F, Chapter 13.

```
pgd> Sum ^ Abs      # Sum then Abs
pgd  Abs(Sum)      # Abs after Sum
```

both yield the statistic that computes the absolute value of the input component sum. They both map input value `x` to scalar value `Abs(Sum(x))`. The arrow of the transform operator `^` should evoke piping the result of `Sum` into `Abs`. It is important to note that the second form `Abs(Sum)` is *not* calling the function `Abs` with argument `Sum`; it uses evaluation as an *operator*, a metaphor that invokes the calling order in `Abs(Sum(x))`. The first form is often most convenient when chaining together statistics formed by complicated expressions, such as those produced by statistic factories, and the second form is typically more convenient when composing named statistics.

We often create statistics with expressions using arithmetic and logical operators as combinators. For example, what do you think the following statistic computes?

```
pgd> (3 * Sum + 10 * Product + 17)
```

For example, try it on `<1,2,3>`; you should get

```
pgd> (3 * Sum + 10 * Product + 17)(1, 2, 3)
<95>
```

This statistic takes 3 times the value of `Sum` on its input, adds 10 times the value of `Product` on its input, and adds 17. The numbers 3, 10, and 17 are automatically upgraded to statistics with the `Constantly` factory. The combinator `+` adds the tuples componentwise but requires that all the tuples have the same dimension or are scalars,⁹ or an error is raised.

The special built-in statistic `__` (read “hole”) is often useful in such expressions. Think of it as a hole that is filled in with whatever tuple is given as input. (Hole itself is the function that returns its input.)

```
pgd> phi = (3 * __ + tup(1, 2, 3))
pgd> phi(10, 100, 1000)
<31, 302, 3003>
pgd> x = tup(10, 100, 1000)
pgd> phi(x)
<31, 302, 3003>
pgd> phi( (10, 100, 1000) )
<31, 302, 3003>
```

Here we define a statistic that multiplies each component of the input by 3 and then adds a fixed tuple componentwise. This requires that the input have dimension 3. (Recall that the function `tup` creates the special tuples that `frplib` uses; look at the value of `tup(1, 2, 3)`.) Note following the Notational Convention described in the box earlier in this section, we can `phi` with a single tuple like `x` and `(10, 100, 1000)` or by giving its components as separate arguments.

Other frequently useful combinators are `Fork`, `ForEach`, and `IfThenElse`. As we have seen, `Fork(s1, s2, ..., sn)` creates a statistic that passes its input to each of the statistics `s1` through `sn` and concatenates their outputs in order into one tuple.

```
pgd> s3 = Fork(Min, Mean, Max)
pgd> s3(20, 6, 0, -10)
<-10, 4, 20>
```

`Fork` lets us operate on data with multiple statistics simultaneously.¹⁰

The `ForEach` combinator takes a statistic (of codimension 1) and applies it to each component of the input tuple, concatenating the resulting outputs in order into one tuple. Constant arguments are automatically converted to statistics with `Constantly`.

```
pgd> squares = ForEach(__ ** 2)
```

⁹Scalars are extended to tuples of higher dimension by repetition. So `1 + <2, 3, 4>` is `<3, 4, 5>`.

¹⁰Cf., the fork operator γ defined in Section 15.5

```

pgd> squares(1, 2, 3, 4, 5)
<1, 4, 9, 16, 25>
pgd> ones = ForEach(1)
pgd> ones(1, 2, 3, 4, 5)
<1, 1, 1, 1, 1>
pgd> tup(1, 2, 3, 4, 5) ^ ForEach((-1, 1))
<-1, 1, -1, 1, -1, 1, -1, 1, -1, 1>
pgd> tup(1, 2, 3, 4, 5) ^ ForEach(9 - 6 * __ + __ ** 2)
<4, 1, 0, 1, 4>

```

The combinator `IfThenElse` takes a condition¹¹ and two statistics. The statistic returned by `IfThenElse` first passes its input to the condition. If the condition evaluates to true (i.e., `<1>`), the input is passed to the first statistic whose value is returned; otherwise, the input is passed to the second statistic whose value is returned.

¹¹A **condition** is a Boolean, scalar statistic that returns `<0>` for false and `<1>` for true.

```

pgd> zeta = IfThenElse(Sum >= 2, Max, Min)
pgd> zeta(-4, 1, 0, 7)
<7>
pgd> zeta(-4, 1, 0, -3)
<-4>
pgd> xi = IfThenElse(Proj[1] >= 0, Proj[2, 3], Proj[4, 5])
pgd> xi(1, 20, 30, 40, 50)
<20, 30>
pgd> xi(-1, 20, 30, 40, 50)
<40, 50>
pgd> evens = ForEach( IfThenElse(__ % 2 == 0, __, nothing) )
pgd> evens(1, 2, 3, 4)
<□, 2, □, 4>
pgd> double_agt3 = ForEach(IfThenElse(Abs > 3, 2 * __, 0))
pgd> double_agt3(-1, 2, 3, -4, 5, 6, -7, -2)
<0, 0, 0, -8, 10, 12, -14, 0>

```

First, notice that the logical operators, like `>=` and `<` and `==`, are combinators that produce *conditions*. These combinators each act on two statistics to produce a condition, with constants automatically wrapped with `Constantly`:

```

pgd> tup(-4, 1, 0, 7) ^ (Sum >= 2)
<1>

```

```
pgd> tup(-4, 1, 0, -3) ^ (Sum >= 2)
<0>
```

With input tuple `x`, the condition `Sum >= 2` returns true when `Sum(x) >= 2` and false otherwise. The parentheses around the expression `Sum >= 2` are needed because of the low precedence Python assigns the `^` operator. (We will see other ways to create conditions below.) Second, a constant passed as one of the last two statistics to `IfThenElse` is automatically wrapped with `Constantly` to give a statistic. Third, in the last two examples, we use the `ForEach` combinator to apply the statistic with `IfThenElse` to each component of the input tuple. This selects out the even components in `evens` and doubles all components with absolute value > 3 in `double_agt3`.

Let's use these tools to create a statistic that truncates all components of its input to the range $[a, b]$ for $a \leq b$, mapping values below (above) this range to a (b). To start, let's define the statistic for a *specific* choice of parameters a and b , say 4 and 9:

```
pgd> trunc_comp_49 = IfThenElse(__ < 4, 4, IfThenElse(__ > 9, 9, __))
pgd> trunc49 = ForEach(trunc_comp_49)
pgd> trunc49(1, 4, 6, 9, 10)
<4, 4, 6, 9, 9>
```

This is fine, but if we want to use such a statistic for different values of a and b , we would need to repeat this code over and over with those values. Instead, we use the **factory trick**. We create a function that takes a and b as parameters and returns the corresponding statistic. Because a and b are defined variables inside the function, we can use them as if they are known, just like we did with 4 and 9 above. It is sufficient to define an ordinary Python function for this purpose, but `frplib` gives us some tools to make these nicer to use. Specifically, we will write a Python function but attach a **decorator** to the definition to give the function special properties. Here's what it looks like in simple form:

```
@statistic_factory
def trunc(a, b):
    trunc_component = IfThenElse(__ < a, a, IfThenElse(__ > b, b, __))
    return ForEach(trunc_component)
```

Ignore the first line for a moment. The rest is just an ordinary Python function with two parameters, our a and b . We then define the statistic as we did above for 4 and 9 but using the given values of `a` and `b` that are defined inside the function. This seems works fine:

```
pgd> tup(1, 4, 6, 9, 10) ^ trunc(4, 9)
<4, 4, 6, 9, 9>
pgd> tup(1, 4, 6, 9, 10) ^ trunc(3, 7)
<3, 4, 6, 7, 7>
```

Here again we use the transform operator `^` to evaluate the statistic returned by our factory. Now back to the first line: `@statistic_factory`. This is called a **decorator**; it modifies the function defined below it to add special features that depend on the decorator and any arguments given to that decorator.

The `@statistic_factory` clearly signals the intended use of the function and automatically constructs documentation for both the factory and the returned statistic. Specifically, it draws descriptive text for the *factory* from the factory's docstring and the help text for the returned statistic from that statistic. Here, we modify the latter's help text with the `.doc` property to give a more specific string that references the factory's parameters. We also want to correct a problem that arises if the parameters are out of order, to wit:

```
pgd> tup(1, 4, 6, 9, 10) ^ trunc(7, 3)
<7, 7, 7, 3, 3>    # Why is this the result?
```

which is not what we want. Figure 2.2 shows the improved version of the `trunc` that sets the help texts and handles the error. A cleaner, equivalent version, shown in Figure 2.3, uses the `auto_doc` option to the decorator to automatically generate documentation for the returned statistic. With this, `statistic_factory` generates the documentation string for both the factory and the returned statistic from the same source, one using the parameter names and one using their values. We can see the improvements, which are the same for both versions:

```
pgd> trunc
A factory producing a statistic that truncates each input component
to non-empty interval [a __ b], with components < a mapping to a
and components > b mapping to b.
pgd> trunc(3, 7)
A Statistic 'trunc(3, 7)' that truncates each input component to non-empty
interval [3_7], with components < 3 mapping to 3 and components > 7 mapping to 7.
pgd> tup(1, 4, 6, 9, 10) ^ trunc(7, 3)
Statistic factory trunc(a, b) requires a <= b, but 7 > 3'
pgd> tup(1, 4, 6, 9, 10) ^ trunc(3, 7)
<3, 4, 6, 7, 7>
```

```

@statistic_factory
def trunc(a, b):
    """truncates each input component to non-empty interval [a __ b], with
    components < a mapping to a and components > b mapping to b
    """
    if a > b:
        mesg = f'Statistic factory trunc(a, b) requires a <= b, but {a} > {b}'
        raise FactoryError(mesg)

    trunc_component = IfThenElse(__ < a, a, IfThenElse(__ > b, b, __))
    stat = ForEach(trunc_component)
    stat.doc = f"""truncates input components to non-empty interval [{a}__{b}], with
    components < {a} mapping to {a} and components > {b} mapping to {b}"""

    return stat

```

FIGURE 2.2. The improved version of the statistic factory `trunc` with better error detection and documentation.

We will use this factory trick frequently, and `frplib` provides decorators for defining different types of factories (e.g., `Statistic`, `Kind`, and `FRP`).

The `frplib` built-in statistics, factories, and combinators can produce a wide variety of useful statistics for analyzing the output of an FRP. For many problems, we need nothing more, but sometimes it is necessary or just easier to craft a **custom statistic**, a piece of code that computes the data-processing algorithm we need. For this, we define a decorated Python function to do the job.

Let's start with a simple example: a statistic that finds the smallest and largest gaps between successive components of an input tuple. Although we will use custom statistics below, note that this can be done with just combinators and built-ins:

```

pgd> big_jumps0 = Diff ^ Fork(Min, Max)
pgd> big_jumps0(1, 11, -9, 0, 4, 7, 13, 2)
<-20, 10>
pgd> big_jumps0(1, 2, 4, 8, 0)
<-8, 4>

```

It is a worthwhile exercise to figure out why this works. The built-in statistic `Diff` computes the tuple of successive differences between components of an input value. The `Fork` statistic factory takes multiple statistics and returns a statistic that applies them all to the input value, joining their results into the output tuple. The result shows that the biggest drop (i.e., minimum jump) is -20 (from 11 to -9) and the


```

@statistic_factory(auto_doc=True)
def trunc(a, b):
    """truncates each input component to non-empty interval [{a} __ {b}], with
    numbers < {a} mapping to {a} and numbers > {b} mapping to {b}
    """
    if a > b:
        mesg = f'Statistic factory trunc(a, b) requires a <= b, but {a} > {b}'
        raise FactoryError(mesg)

    trunc_component = IfThenElse(__ < a, a, IfThenElse(__ > b, b, __))
    return ForEach(trunc_component)

```

FIGURE 2.3. Cleaner version of the statistic factory `trunc` using `frplib` capabilities. The `auto_doc` argument to the decorator dynamically sets the doc string of the returned statistic using the values of `a` and `b` while using the names for the doc string of the factory itself. Note the braces around the parameter names in `trunc`'s doc string. Double braces escape substitution.

biggest jump is 10 (from 1 to 11).

But let's see instead how to do this as a custom statistic.¹² The statistic is defined using a Python function that we write to compute the statistic's value for any input. We make this a *statistic* by applying the `statistic` factory used as a *decorator* on the function's definition. This defines an `frplib` statistic called `big_jumps`:

¹²The functions defined here can be entered directly into the `frplib` playground or defined in a separate file and imported.

```

@statistic(dim=2)
def big_jumps(x):
    "finds the smallest and largest gaps between successive components"
    smallest, biggest = (infinity, -infinity)
    for index in range(len(x) - 1):
        diff = x[index + 1] - x[index]
        smallest = min(smallest, diff)
        biggest = max(biggest, diff)
    return (smallest, biggest)

```

The `@statistic(dim=2)` line is the decorator. Putting it before the definition is equivalent to first defining `big_jumps` without the decorator and then doing

```
big_jumps = statistic(big_jumps, dim=2)
```

which converts the original function into an `frplib` statistic object. The `dim=2` in the decorator is an optional argument that specifies the dimension of the statistic. This is used by `frplib` to optimize and to improve error detection. You can also specify its

codimension explicitly (with the `codim` optional argument), though `frplib` can infer this automatically under some conditions we will see in other examples. It would be perfectly valid to use `@statistic` plain, with no parentheses, as the decorator. The only difference is that `frplib` would not be able to infer the statistic's dimension. When using a decorator like `@statistic` or `@condition` or `@scalar_statistic` without specifying optional arguments, do not use `()`s, just the decorator plain.

The `def big_jumps(x):` line defines a function `big_jumps` that takes a single parameter `x`. When a custom statistic is defined with a single parameter and `codim` is *not* explicitly set to 1, then the parameter is a *tuple*. (When `codim` is explicitly set to 1, the single argument is passed to the statistic instead as a *number*, which is sometimes more convenient. See below.)

The string after the `def` line is a documentation string. We start with a lowercase letter and eschew a period because `frplib` incorporates that string into the help text for the statistic. A docstring is optional but having one is good practice.

A statistic receives a value (i.e., a single tuple of numbers) as input. The code defining `big_jump` labels this tuple as `x`. This tuple is an `frplib` tuple and is handled as it would be in ordinary code. In particular, tuples in Python are 0-indexed; `x[i]` (for non-negative index `i`) gives the `i + 1`st component. We loop over all but the last indices into `x`, compute the difference between successive components, and update our smallest and largest difference. As this is just ordinary Python code, we use the Python builtins `min` and `max`, not the `frplib` statistics `Min` and `Max`,¹³ to update the smallest and largest differences.

¹³In particular, `min` and `max` return numbers, but `Min` and `Max` return *values*.

Now we can apply and operate on `big_jumps` just as we would a built-in statistic.

```
pgd> big_jumps(1, 11, -9, 0, 4, 7, 13, 2)
<-20, 10>
pgd> dim(big_jumps)
2
pgd> help(big_jumps)
A Statistic 'big_jumps' that finds the smallest and largest gaps
between successive components. It expects a tuple and returns a 2-tuple.
```

There are a few variants worth remembering. If you define a statistic with more than one argument, those arguments are the *components* of the input tuple. The number of such arguments determines the codimension of the statistic.

```
@scalar_statistic
def square_distance(x, y):
```

```

"calculates square distance of a 2D point from the origin"
return x * x + y * y

```

The input tuple to this function is $\langle x, y \rangle$, but we get access to the components directly; `frplib` infers that the codimension of this statistic is 2. The decorator `@scalar_statistic` is equivalent to `@statistic(dim=1)` but is somewhat more readable.

If a statistic is given several arguments, the last arguments can be given default values which will be used if the input tuple does not contain those components

```

@scalar_statistic
def scaled_sum(x, y, scale=1, shift=0):
    "sums first two components, then scales by third, then shifts by fourth"
    return (x + y) * scale + shift

```

Then,

```

pgd> codim(scaled_sum)
(2, 4)
pgd> scaled_sum(1, 2)
<3>
pgd> scaled_sum(1, 2, 3)
<9>
pgd> scaled_sum(1, 2, 3, 7)
<16>

```

If a custom statistic is defined with more than one argument and the last one is *starred*, the last argument will be a tuple absorbing all remaining components. Unless specified by the `codim=` argument, the statistic's codimension will be from the number of non-starred arguments up to an arbitrarily large size. As a trivial example:

```

@statistic
def swap_firsts(x, y, *more):
    "swaps the first two components"
    # `more` is a Python tuple of all input components but the first two
    return (y, x, *more)

pgd> codim(swap_firsts)
(2, inf)
pgd> swap_firsts(1, 2, 3, 4, 5)
<2, 1, 3, 4, 5>

```

The star operation is used in two ways in `swap_first`. In the parameter list, it tells us that `more` is a tuple containing all remaining arguments. In the return value, the `*more` *unpacks* the tuple as separate arguments to the `()` operator that builds a tuple.

Ordinarily, when you define a custom statistic with one argument, the parameter is taken as a general tuple. But sometimes you want the parameter to be explicitly a *number* rather than a tuple. For this, you set `codim=1`; then the given value is unwrapped and the variable is used as a number in your code. For example,

```
@statistic(codim=1)
def winsor10(x):
    "thresholds its scalar argument at +/- 10"
    if abs(x) > 10:
        return 10
    return x
```

Notice that `x` is a number, and we use the Python built-in numeric function `abs` (in particular, not the statistic `Abs`).

Finally, we often want to define a *condition*, a Boolean-valued, scalar statistic whose value is interpreted as true (1) or false (0). For this we use the `@condition` decorator, which is a special case of `@scalar_statistic`.

```
@condition
def first_bigger(x):
    "returns true if first component bigger than second, else false"
    return x[0] > x[1]
```

This function can return Python Booleans (`True` or `False`), the tuples `<1>` or `<0>`, or any truthy and falsy objects, and they will automatically be converted to the values used for true (`<1>`) and false (`<0>`) in `frplib`.

```
pgd> first_bigger(1, 2)
<0>
pgd> first_bigger(2, 1)
<1>
```

As another example, the following condition tests if its argument – a scalar – is an integer. The `codim=1` argument to `condition` causes the parameter of the function to be interpreted as a number rather than a tuple.

```
@condition(codim=1)
```

```
def isInteger(x):
    "returns true if its scalar argument is an integer"
    return isinstance(x, int)
```

`isInteger(4)` is true, `isInteger(4.2)` is false.

Often, conditions are combinations of Boolean statements with logical-and/or/not, and for this we use the `And`, `Or`, and `Not` combinators, all of which take and return conditions.

```
pgd> isDivisibleBy6 = And(__ % 2 == 0, __ % 3 == 0)
pgd> pos_div2_or_3 = And(Scalar > 0, Or(__ % 2 == 0, __ % 3 == 0))
pgd> pos_integer = And(is_integer, Scalar > 0)
pgd> isOdd = Not(__ % 2 == 0)
```

Try evaluating these conditions on various values to make sure you understand their meaning. The combinators `All` and `Any` are like logical-and and logical-or applied to the components of a value. They take a condition (which must have dimension 1) and return a condition that is true if the given condition is true for *all* components or for *any* component of the value, respectively.

```
pgd> tup(1, 1, 0, 2.4) ^ All(isInteger)
<0>
pgd> tup(1, 1, 0, 2.4) ^ Any(isInteger)
<1>
```

Like `And`/`Or`/`Not`, both `All` and `Any` return conditions, so they do not need to be wrapped in a call to `condition`.

As a further example, consider a random system where we observe the values of five rolls of a six-sided die. Two questions we might ask are: How many rolls does it take until we see the first 6, if ever? In how many of the five rolls do we see the most common number that appears? We can define statistics to answer both questions. For the first question:

```
@scalar_statistic
def when_first_6(rolls):
    "counts rolls until a 6, or infinity if none"
    try:      # This will fail unless there is a roll of 6
        return 1 + rolls.index(6) # .index() of first 6 from 0
    except:   # There is no roll with value 6, do this
        return infinity
```

This statistic will accept a tuple of any dimension and always returns a scalar, as indicated by the `@scalar_statistic` decorator. We use `rolls.index` to search the components of the input tuple for the first 6 and return the index (from 1) of the first found. If there is no 6, `rolls.index(6)` raises an exception, which we catch in the `except` block to return the default value `infinity`.

```
pgd> when_first_6
A statistic 'when_first_6' that counts rolls until a 6, or infinity if none.
It expects a tuple and returns a scalar.
pgd> when_first_6(5, 1, 3, 4, 2)
<Infinity>
pgd> when_first_6(4, 4, 6, 2, 6)
<3>
```

We will approach the second question in two ways. The simplest approach is to count how many times each number appears and return the largest. We use a dictionary initialized with count 0 for every possible roll, increment the count for each roll, and then take the largest of the dictionary's values.

```
@statistic(dim=1)
def most_common_count(rolls):
    "returns number of rolls with most common value"
    counts = { roll: 0 for roll in xrange(1, 6) }
    for roll in rolls:
        counts[roll] += 1
    return max(counts.values())

pgd> most_common_count
A Statistic 'most_common_count' that returns number of rolls with most common value.
It expects a tuple and returns a scalar.
pgd> most_common_count(2, 4, 2, 5, 2)
<3>
```

A more general approach computes both the most common count and the value of a roll that occurred that many times.

```
@statistic(dim=2)
def most_common_roll(rolls):
    "returns a most common roll and its frequency"
```

```

counts = { roll: 0 for roll in irange(1, 6) }
best = (1, 0)
for roll in rolls:
    counts[roll] += 1
for roll, count in counts.items():
    if count > best[1]:
        best = (roll, count)
return best

```

We use the dictionary in the same way, but extract more information from it.

```

pgd> most_common_roll(2, 4, 2, 5, 2)
<2, 3>

```

Puzzle 19. Create a statistic like `most_common_roll` that produces a tuple giving the number of rolls with all six values, in order. What is the dimension of this statistic?

Puzzle 20. Create a statistic that answers the question of whether either pattern 1, 2, 3 or pattern 4, 5, 6 occurs in three successive rolls.

As another example, suppose we have data that represents a point inside a cube centered at the origin, aligned with the axes, and with edge length 2. We might ask whether the point lies on a corner, edge, face, or in the center of the cube. (See Example 2.3.) Because we use numeric data, we will encode the four cases as numbers: 3 for corner, 2 for edge, 1 for face, and 0 for center.

```

@statistic
def classify_point(x):
    "determines type of cube point (0=corner,1=edge,2=face,3=center)"
    distance_from_origin, = Norm(x)
    if distance_from_origin >= numeric_sqrt(3):
        return 3 # corner
    if distance_from_origin >= numeric_sqrt(2):
        return 2 # edge
    if distance_from_origin >= 1:
        return 1 # face
    return 0 # center

```

The statistic `classify_points` shows one of several possible ways to write this function. We use the built-in statistic `Norm` to compute the distance of the input point from the origin. Because `Norm` is a statistic, it returns an tuple, 1-dimensional in this case, so we *deconstruct* the tuple in the assignment to `distance_from_origin` to extract the first value. The trailing comma tells Python that we want the first element in a 1-dimensional tuple. (Deconstructing a 2-tuple looks like `a, b = two_tuple`, a 3-tuple like `a, b, c = three_tuple`, and so forth.) This statistic behaves as we would expect:

```
pgd> classify_point(1, 1, 1)
<3>
pgd> classify_point(1, 0, 1)
<2>
pgd> classify_point(1, 0, 0)
<1>
pgd> classify_point(0, 0, 0)
<0>
```

Some statistics require special handling of their arguments, and `statistic` has options to control that. For instance, in the statistic `quadratic_roots` for finding the roots of a quadratic, we want some extra precision. We use the `arg_convert` option to ensure that all the arguments are converted to high-precision real quantities. We also specify the codimension explicitly by giving three arguments in the function definition.

```
@statistic(dim=2, arg_convert=numeric.as_real)
def quadratic_roots(a, b, c):
    "returns roots of quadratic  $a x^2 + b x + c$ "
    if is_zero(a):
        root = -b/c
        return (root, root)
    center = -b / (2 * a)
    disc = numeric_sqrt(center * center - c / a)
    return (center - disc, center + disc)
```

The function `is_zero`, provided by `frplib`, checks if a quantity is zero to within numerical precision.

```
pgd> codim(quadratic_roots)
```



```
(3, 3)
pgd> quadratic_roots(1, 0, -1)
<-1, 1>
pgd> quadratic_roots(1, 2, 1)
<1, 1>
```

The `codim(quadratic_roots)` returns the lowest and highest valid codimension for the statistic; in this case, the codimension is exactly 3.

Finally, we rewrite the `trunc` statistics factory from Figure 2.2 using custom statistics rather than with built-ins and combinators as we did originally. The top version of the function shown in Figure 2.4 gives bare-bones functionality without error detection or careful documentation. The notable thing here – seemingly odd if you have not seen it before – is that we define a function (`trunc_ab`) within another function (`trunc`). But `trunc_ab` is just another local variable that happens to be a function; we can act on it and return it as with any local variable. The reason for this factory pattern is that the inner function can access the values of `a` and `b` with which the outer function is called. The returned statistic will *remember those values*. This is another instance of the [factory trick](#).

Within `trunc_ab`, the line

```
return (a if x_i < a else b if x_i > b else x_i for x_i in x)
```

loops through the components of the input tuple producing a sequence of truncated values that `frplib` wraps in a tuple. Note the similarity to the `IfThenElse` version in Figure 2.2. For each component of `x`, the inner expression gives `a` if `x_i < a`, `b` if `x_i > b`, or `x_i` otherwise.

As before, we can improve this function with error handling and more specific documentation on the statistic. In the version above, the docstrings on the functions `trunc` and `trunc_ab` are automatically used to provide help text on those objects. Most of the time, this is sufficient to provide good documentation for your custom statistics.

But in the case of statistics *factories*, it can be nice if the statistic returned has documentation that depends on the arguments to the factory function. Because Python docstrings cannot capture the values of local variables (e.g., `a` and `b`), we can use the `description` argument to the `@statistic` decorator to set the documentation string for the statistic. This method is illustrated by the bottom version in Figure 2.4, along with the error-handling method from earlier. Taking care like this to provide good documentation strings for our statistics is optional and, in cases like this,

```

@statistic_factory
def trunc(a, b):
    """truncates each component of input to [a __ b] for a <= b"""

    @statistic
    def trunc_ab(x):
        """truncates every input component to a specified interval"""
        return (a if x_i < a else b if x_i > b else x_i for x_i in x)

    return trunc_ab


@statistic_factory
def trunc(a, b):
    """truncates each component of input to [a __ b] for a <= b"""
    if a > b:
        mesg = f'Statistic factory trunc(a, b) requires a <= b, but {a} > {b}'
        raise FactoryError(mesg)

    doc = f""""A Statistic that truncates input components to [{a}__{b}],
    with numbers < {a} mapping to {a} and numbers > {b} mapping to {b}""""

    @statistic(name=f'trunc_{a}_{b}', description=doc)
    def trunc_ab(x):
        return (a if x_i < a else b if x_i > b else x_i for x_i in x)

    return trunc_ab

```

FIGURE 2.4. Two custom versions of the statistic factory `trunc` originally shown in Figure 2.2. The top version gives the bare bones functionality. The bottom version defines and returns a custom statistic rather than using built-ins, factories, and combinators. The bottom version carefully sets the documentation of the returned statistic based on the factory's arguments.

lengthens the code, but having our objects well documented makes the environment nicely discoverable. Compare these two versions of `trunc` closely to distinguish what is essential to the computation (the definition of `trunc_ab`) from these helpful extras (the descriptive strings and error message for `trunc` and `trunc_ab`).

Key Take Aways

A **statistic** is a data-processing algorithm, *a function that takes a value as input and returns a value as output*. A key use for statistics is to represent our *questions* about the data.

We use the word **value** to mean a tuple of dimension ≥ 0 , and unless otherwise indicated, we will assume that this means a *tuple of numbers*.

For our statistics, we require that all inputs of the equal dimension n return outputs of the same dimension n' , where n and n' need not be equal.

When a statistic accepts tuples of dimension n as input and returns tuples of dimension n' as output, we say that it has

- **type** $n \rightarrow n'$,
- **codimension** n , and
- **dimension** n' .

We usually try to give meaningful names to statistics, both in code and in mathematics. But we name *generic* statistics with Greek letters ψ , φ , ζ , and ξ , and we only use these letters for statistics.

A **condition** is a special case of a scalar statistic that returns a Boolean value, true or false. In practice, we identify true with the value 1 and false with the value 0. So, conditions are indicator functions, often written with Iverson braces and as anonymous functions when convenient.

In `frplib`, there are four ways to produce statistics:

- directly using *built-in statistics*,
- calling a *statistic factory* to create a statistic with specified properties,
- calling a *statistic combinator* to combine one or more statistics,
- creating a *custom statistic* by decorating a Python function.

Built-in statistics (other than “hole” `__`) in `frplib` are named with a capitalized word. These are distinct from Python built-ins that have lowercase names.

2.2 Transformed FRPs

An FRP represents some data that we can measure when observing the evolution of a random system. When that FRP is *fresh*, the associated data have not yet been observed and thus have an *uncertain* value. Uncertain here does not mean “arbitrary” or “completely unknown.” Our knowledge of the FRPs value is reflected in its Kind, and we can exploit this knowledge to make predictions. This is why a fresh FRP is a useful idea: we can reason about its value – and about what we might do with that value – *before* we observe the value. One of the principal ways we do this is by reasoning about the output of a data-processing algorithm that takes the FRP’s value, when available, as input.

A **transformed FRP** is created from a source FRP and a statistic. Its value is the output of the statistic when given the source FRP’s value as input. When the source FRP is fresh, so is the transformed FRP. A transformed FRP thus lets us reason about – and predict – the output of our statistics when applied to our data. Because we use statistics to embody questions we want to answer about our data, this allows us to reason about and predict the uncertain answers to our questions.

FRPs gives us a *physical framework* for understanding transformation that will be helpful for expressing this process mathematically later. When an FRP is activated, it sends its value via a signal through its All output port, we can capture that signal and transform it with a statistic, using the output to configure an empty FRP.

The statistic is computed with circuitry embedded in an adapter like that shown in Figure 2.5. The input value passes through the bottom port on the adapter, and the output is emitted from the top center port. The adapters for different statistics look the same, though they each have different circuitry and different labels.

The wiring diagram for building a transformed FRP is shown in Figure 2.6, with a spare, stylized form of wiring diagram shown in Figure 2.7. On the lower right, we have the source FRP which is to be transformed. On the upper right, we start with an *empty* FRP that will be reconfigured to the transformed FRP. We connect the output of the source’s All port to the new FRP’s Input port, passing the signal through the statistic adapter. Once this connection is made, the transformed FRP is configured, and the metallic display shows its Kind.¹⁴ Both the source and transformed FRPs start as fresh. Activating either FRP activates the other. The activated source FRP sends its value through the statistic to produce the value for the transformed FRP. However, each FRP is *observed* only by pushing its own button. Pushing the button on the transformed (source) FRP displays its value, and the source (transformed)

¹⁴We will see how to derive that Kind in the next section.

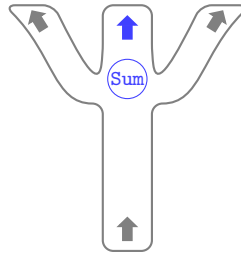


FIGURE 2.5. An adapter used to transform an FRP by the Sum statistic in the playground, which computes the sum of its input's components. The value comes from the source FRP's All output port through the bottom port on the adapter. It is transformed by the statistic and then emitted through the top center port. The other two ports on top simply copy the original value, allowing us to operate on the source FRP in multiple ways. Different statistics' adapters look the same but have different names and internal circuitry. That the adapter resembles the letter ψ will be a useful mnemonic.

FRP is activated but *not* observed.

The connection between the source FRP and the transformed FRP implies a deterministic relationship between their values. This is the key point to remember: when activated, **if the source FRP produces value v and the statistic is ψ , then the transformed FRP will produce value $\psi(v)$.**

The construction of a transformed FRP only works when all the possible values of the source FRP are valid inputs to the statistic. In this case, we say that the source FRP and statistic are *compatible*. Attempting to connect an FRP with an incompatible statistic will raise an error indicator.

Definition 6. An FRP X and a statistic ψ are **compatible** if every possible value of X is a valid input to ψ .

This implies that the dimension of X is equal to a codimension of ψ and that whatever value X produces can be passed as input to ψ .

The physical wiring of a transformed FRP shown in Figures 2.6 and 2.7 can be expressed mathematically or in code in several ways. In the algebraic interpretation, we express transformation with two *binary operators*: using either the **transform operator** $\blacksquare \hat{\ } \blacksquare$ or the **overloaded evaluation operator** $\blacksquare(\blacksquare)$ as we prefer. The holes $\blacksquare$ are to be filled in by the operands, a statistic for one and an object to be transformed (e.g., FRP, Kind, or Statistic) for the other.¹⁵ The transform operator takes an object on the left and a statistic on the right; the overloaded evaluation operator takes the statistic on the left (before the parentheses) and the object on the

¹⁵By analogy, we would write the addition operator as $\blacksquare + \blacksquare$ with a hole on each side of the plus sign.

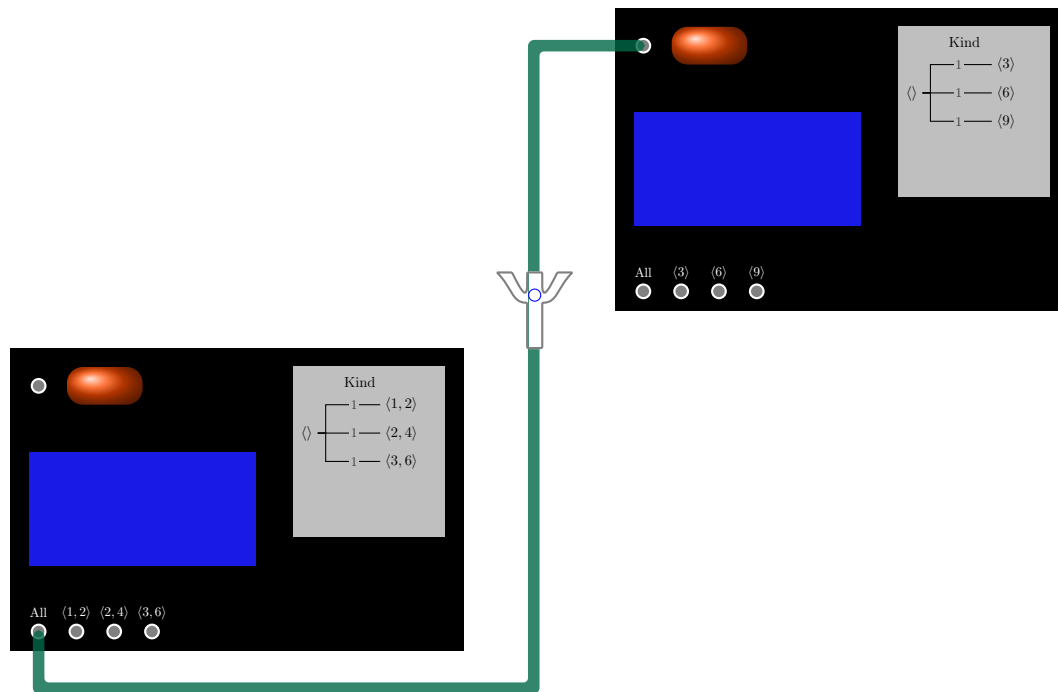


FIGURE 2.6. Wiring diagram for creating a transformed FRP. The source FRP is on the lower left, and the transformed FRP is on the upper right. The wire connects the source's All output port to the transform's input port *through* the statistic adapter. Both FRPs are fresh, but pushing either button will activate both FRPs, first the source and then the transform immediately after. On activation, the source's value is transformed by the statistic and becomes the value of the transformed FRP.

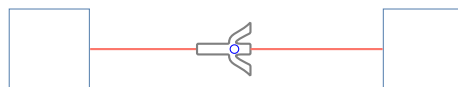


FIGURE 2.7. A stylized form of the wiring diagram in Figure 2.6, with the source FRP on the left and the transformed FRP on the right. This elides the details (displays, buttons, ports) of the FRPs, showing only boxes, wires, and an adapter. We can put labels in the boxes or on the adapter to show the names of the FRPs and statistic. We use this form more frequently in what follows.

right (inside the parentheses) The arrow $\hat{}$ is intended to evoke the wiring connection in Figure 2.7 from source to transformed through the statistic.

So if X is an FRP and ψ is a statistic (`psi` in `frplib`), we can write the transforms mathematically as either

$$X \hat{} \psi \qquad \text{or} \qquad \psi(X)$$

and in `frplib` as either

$$X \hat{} \text{psi} \qquad \text{or} \qquad \text{psi}(X)$$

In mathematical work, we almost always use the overloaded evaluation operator form $\psi(X)$, but in code, we use both forms frequently for reasons we will see. The notation $\psi(X)$ looks like we are passing the FRP to the statistic, but we are *not*. We are reusing the *syntax* of evaluation as an operator. This is discussed at length below.

We can also express transformation as a simple procedure

FRP Y :

```
 $x \leftarrow X$ 
return  $\psi(x)$ 
```

that defines the transformed FRP $Y = X \hat{} \psi$. In `frplib`, this looks almost exactly the same, using the syntax of a decorated function definition and replacing the yield operator $\leftarrow$ with `= yield`:

```
@frp
def Y():
    x = yield X
    return psi(x)
```

The procedure for Y extracts the value of X when it is available and uses the output of ψ on that value as the value of Y . When doing transformations by statistics only, we mostly use the algebraic form, saving the procedural form for combinations of several operations. But all these different forms are interchangeable.

To get a feel for how transforms work, we will continue our study of Roulette.

Example 2.2 Roulette, Part 2

In Example 2.1, we looked at the rules of Roulette and examined the statistics corresponding to the various bets (or “plays”) we can make in the game. Here, we will build the Data FRP for a spin of the roulette wheel and transform it to derive feature FRPs for each play.

As before, we start by loading the `roulette` object into the `frplib` playground:

```
pgd> from frplib.examples.roulette import roulette
```

Each call to `roulette` as a function with no arguments returns a fresh FRP representing a single spin of the roulette wheel.

```
pgd> roulette()
An FRP with value <21>
pgd> roulette()
An FRP with value <0>
pgd> R = roulette()
```

Having observed two spins, we create another FRP `R` represents a spin of the roulette wheel as a random quantity. The system we are studying is a single spin of the wheel, and `R` serves as the “Data FRP” for this system, from which we will derive and predict answers to various questions. This FRP is fresh

```
pgd> R.is_fresh
True
```

so we can think of it as a spin that has not yet occurred. The pocket the ball will land in is uncertain.

Will a \$1 play on Red win for this spin? What about a \$1 straight play on 16? Or a \$10 split play on 12 and 15? Each of these questions is embodied by a statistic, `roulette.red`, `roulette.straight(16)`, and `10 * roulette.split(12, 15)`, respectively. The *answers* to these questions are represented by transformed FRPs, transforming `R` by each of these statistics.

```
pgd> Red = R ^ roulette.red
pgd> Str16 = R ^ roulette.straight(16)
pgd> Spl12_15 = R ^ (10 * roulette.split(12, 15))
```

We could equivalently define `Red = roulette.red(R)` or define


```
@frp
def Spl12_15():
    pocket = yield R
    play1 = pocket ^ roulette.split(12, 15)
    return 10 * play1
```

Here, we use that `play1` is an `frplib` tuple to pass it to the statistic `roulette.split(12, 15)` with the transform operator to get the value of a \$1 play. This could be written even more concisely as

```
@frp
def Spl12_15():
    pocket = yield R
    return pocket ^ (10 * roulette.split(12, 15))
```

if one chooses. However, note that we keep each `yield` on its own line.

In the framework of Figure 1.12, `R` plays the role of Data FRP and each of these transformed FRPs has the role of a *feature FRP*. Because `R` is fresh, so are all these:

```
pgd> (Red.is_fresh, Str16.is_fresh, Spl19_12.is_fresh)
(True, True, True)
```

This brings us to an essential point. These feature FRPs are random quantities that *represent* specific physical outcomes in the system, even when we do not know their values and even before they *have* values. They are related to each other in ways that reflect the logic of the system, and we can reason about those relationships. For instance, 16 and 12 are red pockets, but 15 is not. So if the straight play on 16 wins, then so does the play on red, but the split play loses because 16 is neither 12 nor 15. Similarly, if the split play wins, then the straight play must lose. The point is that what we know about the system, about the Kind of the Data FRP `R`, and about the statistics that mediate these relationships gives us information about the values of these FRPs.

Observing the value of `Red` leaves the other FRPs activated but *not* observed.

```
pgd> Red
An FRP with value <-1>
pgd> (R.is_fresh, Red.is_fresh, Str16.is_fresh, Spl19_12.is_fresh)
(False, False, False, False)
```

Yet in light of the observation of **Red**'s value, our *knowledge* about the values of other FRPs here has changed. Because the pocket is not red, the straight play must have lost, as 16 is red. Though we have not *observed* the value of **Str16**, we can deduce it. Updating our knowledge with observational constraints is one of the Big 3+1 and is covered in detail in Chapter 5. But it is worth a quick look here at how constraining these FRPs with the observation of **Red**'s value changes our knowledge because it reinforces the point of how the relations among the data and feature FRPs allows us to reason about questions.

Constraining the FRP **Straight** with the observation of **Red**'s value, gives us a new FRP related to **Straight** whose Kind reflects our updated knowledge of **Straight**'s value. The update Kind is

```
<> ----- 1 ---- -1
```

showing that after seeing that **Red** loses, we *know* that **Straight** loses as well, despite not yet having observed **Straight**. As an aside, we can define the FRP representing the straight 16 play constrained by the observation that red loses with the following procedure:

```
@frp 1
def Straight_updated(): 2
    pocket = yield R 3
    yield given(roulette.red(pocket) == -1) 4
    return pocket ^ roulette.straight(16) 5
```

Lines 1 and 2 give the syntax for a procedure defining the FRP **Straight_updated**. Line 3 extracts the value (i.e., pocket) from **R** when available. Line 4 constrains the value to be consistent with the observation that **Red** loses. And line 5 transforms the value to the outcome of the straight 16 play for that pocket. We can then look at `kind(Straight_updated)` to see the updated Kind above. In the next section, we will see how to do this directly with a Kind procedure.

The change in our knowledge about **Sp112_15** is more subtle: initially, the play **Sp112_15** could win if the ball landed in 2 out of the 38 pockets, but once we have observed that the pocket is not red, the split play can only win with 1 out of 20 pockets. The original Kind (`kind(R) ^ roulette.split(12, 15)`) is

```
,---- 18/19 ---- -1
<> -|
```

```
`---- 1/19 ----- 17
```

and the updated Kind constrained by the observation that Red loses is

```
,---- 0.95000 ----- -1
<> -|
`---- 0.050000 ----- 17
```

which is different than before the observation.

We can see that our knowledge about an FRPs value lies on a spectrum, from the perfect knowledge of observing the value – or merely deducing it logically – through various degrees of uncertainty.

Puzzle 21. The procedure above defines the updated FRP (that we named `Straight_updated`) of `Straight` constrained by the observation that Red loses. From this, take a guess at one way to write a procedure defining the updated FRP from `Sp112_15` based on the same observation.

Notice that we can package all of these plays into a single FRP. Using the Fork combinator described earlier, we have

```
pgd> my_plays = Fork(roulette.red, roulette.straight(16),
...>                10 * roulette.split(12, 15))
pgd> Plays = R ^ my_plays
pgd> Plays
An FRP with value <-1, -1, -10>
```

If this had had value $\langle -1, -1, 170 \rangle$, we could have deduced that R has value 15. Yet R remains activated but not observed. Finally, we observe R

```
pgd> R
An FRP with value <17>
```

We can simulate many spins of the wheel and look at the values of the pocket and all these plays:

```
pgd> FRP.sample(1000000, R ^ Fork(__, my_plays))
      Summary of Output Values
+-----+-----+-----+
| Values          | Count | Proportion |
+=====+=====+=====+
```

<-1, -1, -1, -10>	26461	2.646%
<0, -1, -1, -10>	26207	2.621%
<1, 1, -1, -10>	26368	2.637%
<2, -1, -1, -10>	26336	2.634%
<3, 1, -1, -10>	26446	2.645%
<4, -1, -1, -10>	26601	2.66%
<5, 1, -1, -10>	26276	2.628%
<6, -1, -1, -10>	26372	2.637%
<7, 1, -1, -10>	26495	2.65%
<8, -1, -1, -10>	26480	2.648%
<9, 1, -1, -10>	26193	2.619%
<10, -1, -1, -10>	26476	2.648%
<11, -1, -1, -10>	26096	2.61%
<12, 1, -1, 170>	26531	2.653%
<13, -1, -1, -10>	26546	2.655%
<14, 1, -1, -10>	26227	2.623%
<15, -1, -1, 170>	26335	2.634%
<16, 1, 35, -10>	26262	2.626%
<17, -1, -1, -10>	26119	2.612%
<18, 1, -1, -10>	26045	2.604%
<19, 1, -1, -10>	26415	2.642%
<20, -1, -1, -10>	26282	2.628%
<21, 1, -1, -10>	26235	2.623%
<22, -1, -1, -10>	26200	2.62%
<23, 1, -1, -10>	26340	2.634%
<24, -1, -1, -10>	26166	2.617%
<25, 1, -1, -10>	26299	2.63%
<26, -1, -1, -10>	26306	2.631%
<27, 1, -1, -10>	26306	2.631%
<28, -1, -1, -10>	26128	2.613%
<29, -1, -1, -10>	26268	2.627%
<30, 1, -1, -10>	26138	2.614%
<31, -1, -1, -10>	26477	2.648%
<32, 1, -1, -10>	26600	2.66%
<33, -1, -1, -10>	26199	2.62%
<34, 1, -1, -10>	26190	2.619%

```
| <35, -1, -1, -10> | 26329 |      2.633% |
| <36, 1, -1, -10>  | 26250 |      2.625% |
+-----+-----+-----+
```

Compare with Figure 2.1. The first component of each value is the pocket the ball lands in, and the remaining are the outcomes of the corresponding plays. These values reflect the relationship between the Data FRP `R` and the feature FRPs `Red`, `Spl12_15`, and `Str16`. If we exclude the pocket from the table, we get

```
pgd> FRP.sample(1_000_000, R ~ my_plays)
      Summary of Output Values
+-----+-----+-----+
| Values          | Count | Proportion |
+=====+=====+=====+
| <-1, -1, -10>  | 500415 | 50.04% |
| <-1, -1, 170>  | 26318  | 2.632% |
| <1, -1, -10>   | 420597 | 42.06% |
| <1, -1, 170>   | 26241  | 2.624% |
| <1, 35, -10>   | 26429  | 2.643% |
+-----+-----+-----+
```

whose values reflect the logical relations among the plays.

The takeaways from this example include illustrating how to create transformed FRPs, showing how transforming an FRP by a statistic induces a relationship between the FRPs that we can reason about, and seeing how those connections impact the life cycle (fresh, activated, observed) of the FRPs themselves. We have not yet discussed our probabilistic assumptions about the system or analyzed the system and made predictions, though we can get a sense of it in the above demo table. We will dive more deeply into that in our next look at roulette in Example 2.5.

Puzzle 22. Define a combined play in `frplib` by

```
comb = 10 * roulette.even + 5 * roulette.corner(25) +
       20 * roulette.straight(4) + 50 * roulette.column(2)
```

What are the possible values of the following FRP?

```
roulette() ^ comb
```

Simulate the outcome of this play on many spins.

The previous example shows how we observe data from a random system and extract information from it using statistics to answer our questions. Each statistic, like `roulette.red` or `roulette.straight(16)`, takes the data as input and computes an answer to one question. The example also illustrates how we do our analysis of the system before the FRPs are (fully) observed. If we *had* the data in hand – the pocket the ball lands in – we could simply apply the statistic to the observed value to compute our answer, and there would be nothing uncertain and nothing to predict. But of course by that point, it is too late to bet. If we are choosing among various plays, it does us no good to answer the question after betting is closed. These predictions are made before all the data is observed – and the answers to our questions determined – so that we can make decisions or take actions when they can be useful. This is why we transform the Data FRP into a feature FRP. The feature FRP represents a specific random outcome that we will care about and that we want to predict. A feature FRP like `roulette.red(R)` is only activated when `R` is activated, but before knowing its value, we can compute its Kind and thus predict its value.

There are three ingredients in transforming an FRP by a statistic:

1. a fresh FRP X to be transformed, which we call the source FRP,
2. a statistic ψ for which any possible value of X is a valid input, embodied by an adapter as in Figure 2.5,
3. a fresh, empty FRP that is automatically reconfigured into the transformed FRP Y .

The wire from the X 's All output port through the ψ adapter to the Y 's input port carries X 's value when it is produced to produce Y 's value. When X produces value v , Y produces value $\psi(v)$.

Our algebraic notation for the transformed Y is either $Y = X \hat{\psi}$, with the transform operator or $Y = \psi(X)$ with the overloaded evaluation operator. The latter takes a moment to get used to but is very convenient in mathematical work. Now wait, you say? A statistic is a function from values to values; it does *not* accept an FRP as an argument. And you are correct! We write $\psi(X)$ not to mean that we pass the FRP to the statistic but instead coopting the evaluation syntax as a *operator*, a metaphor for the fact that we *will* pass X 's value through ψ to get the value of

the transformed FRP. Writing the transformed FRP as $\psi(X)$ evokes this connection. Think of X as a placeholder in this expression for the value it will eventually produce.

Definition 7. If X is an FRP and ψ is a compatible statistic, then X transformed by ψ is the FRP that produces value $\psi(v)$ when X produces value v .

We denote the transformed FRP by either $X \hat{\sim} \psi$ or $\psi(X)$.

If X has dimension n and ψ has type $n \rightarrow n'$, then $\psi(X)$ has dimension n' .

Observe that the types compose. If X has type $0 \rightarrow n$ and ψ has type $n \rightarrow n'$, then $\psi(X)$ has type $0 \rightarrow n'$. That is, $0 \rightarrow n$ connected to $n \rightarrow n'$ gives us $0 \rightarrow n'$, with the common n 's in the middle absorbed.¹⁶

In `frplib`, we can also use both forms interchangeably; they are equivalent. If `X` is an FRP and `psi` is a statistic, you can write the transformed FRP as `X  $\hat{\sim}$  psi` or `psi(X)`, as you prefer. From Example 2.2, for instance

```
R  $\hat{\sim}$  roulette.red
roulette.red(R)
```

both transform the FRP `R` by the statistic `roulette.red`. The two forms differ in the order in which the statistic and FRP are given but lead to the same result. The second (mathematical) form is easy to use when the statistic has a name. The first (operator) form is nicer when the statistic itself is more complicated. Compare, for instance

```
R  $\hat{\sim}$  roulette.straight(16)
roulette.red(16)(R)
```

Here, the operator form reads more naturally because the extra parentheses in the second line can be harder to parse.

Note Well! The mathematical notation $\psi(X)$ for a transformed FRP, and the analogous `psi(X)` in `frplib`, are intended to evoke the physical transformation of the source FRP in Figure 2.6: passing X 's value – whatever it is when it is produced – through the statistic ψ to produce the transformed value.

But keep this in mind: we are *not* literally evaluating the function ψ with argument X (or `psi` with argument `X`). Rather, as a metaphor, we are co-opting the notation of evaluation as an *operator* on the source FRP by the statistic. Think of X in this expression as a “hole” that we will fill with X 's value *when it becomes available*, thus passing that value to ψ .

¹⁶This is reminiscent of matrix multiplication: an $m \times p$ matrix times a $p \times n$ matrix gives an $m \times n$ matrix.

If it seems like a lot to ask that we overwrite an established syntax like function evaluation, take solace in that this convention is both very convenient and meaningful, once you get used to it.

Example 2.3. $\mathcal{C}$ is a cube with side length 2, aligned with the coordinate axes, and centered on the origin. Let P be the FRP of dimension 3 whose value represents a random point from among the corners of $\mathcal{C}$, the midpoints of $\mathcal{C}$'s edges, the centers of $\mathcal{C}$'s faces, or the center of $\mathcal{C}$ (the origin). How far is the point from the origin? For this question, we define the statistic φ of type $3 \rightarrow 1$ that computes the distance in space from its input to the origin:

$$\varphi(x, y, z) = \sqrt{x^2 + y^2 + z^2}.$$

The FRP $D = \varphi(P)$ has dimension 1 and size 4, with possible values $\sqrt{3}$, $\sqrt{2}$, 1, and 0. Its value answers our question.

It is worth seeing how to build the FRPs P and D from the previous example in `frplib`. The built-in statistic `Norm` is equal to the statistic φ in the example. We first build the Kind of P , then build P and transform it to get D :

```
coord = [-1, 0, 1]
k = uniform((x, y, z) for x in coord for y in coord for z in coord)
P = frp(k)
D = Norm(P)    # or equivalently D = P ^ Norm
```

We first create a sequence of tuples¹⁷ whose components are all -1, 0, or 1. The line

```
(x, y, z) for x in coord for y in coord for z in coord
```

generates such a sequence by looping over all 27 combinations of components. The mathematical equivalent of this sequence is the set¹⁸

$$\{(x, y, z) \mid x, y, z \in [-1..1]\}.$$

Then, we use the `uniform` Kind factory to construct the Kind with equal weights on all such points, use `frp` to create a fresh FRP with that Kind, and finally create the transformed FRP.

¹⁷Here, `frplib` converts the Python tuples to special `frplib` tuples automatically, so calling `tup(x, y, z)` is optional.

¹⁸This is set builder notation; see equation 9.1 in Interlude F.

Example 2.4. An Olympic archery target has 10 equal width rings (two in each of five colors: white, black, blue, red, gold). Any shot outside those rings counts as a miss, scoring 0 points. Otherwise, a shot is scored $1, 2, \dots, 10$ by the region it hits, with 1 for the outermost white ring and increasing towards the center (gold). An archer shoots 6 arrows in a round.

Let A be the 6-dimensional FRP that represents the shots of one archer during a single round. The values of A are tuples whose entries are the archer's scores on the six successive shots. So, $\langle 2, 4, 10, 9, 7, 10 \rangle$ and $\langle 5, 6, 0, 1, 3, 10 \rangle$ are two possible values of A .

We define three relevant statistics of type $6 \rightarrow 1$.

- **score** is the archer's total score during the round;
- **hits** is the number of times the archer hits the target; and
- **bulls** is the number of times the archer hits the center-most "bull's eye" region

Here, we will define these statistics in two ways to make their meaning clearer – in code and mathematically. Look at both definitions for each function to see how they are doing the same thing.

In Python, we define these statistics like ordinary functions, except apply a decorator *decorator* (`@scalar_statistic`) before the definition so `frplib` knows they are statistics, which gives them useful attributes and convenient operations.

```
@scalar_statistic(codim=6)
def score(v)
    "Archer's total score in one round."
    return Sum(v)

@scalar_statistic(codim=6)
def hits(v)
    "Archer's number of hits in one round."
    hits = 0
    for shot in v:
        if shot > 0:
            hits += 1
    return hits

@scalar_statistic(codim=6)
def bulls(v)
```

```

    "Archer's number of bull's eyes in one round."
    bulls_eyes = 0
    for shot in v:
        if shot == 10:
            bulls_eyes += 1
    return bulls_eyes

```

To define the mathematical versions of these statistics, we use *indicators*, described in Chapter 12 of Interlude F, and summation notation $\sum_{i=1}^6 x_i = x_1 + \dots + x_6$. One way to write indicators is to put a Boolean expression in Iverson braces, like $\{v_i > 0\}$. It is a mathematical analogue of an if-then-else: If $v_i > 0$, then return 1 else return 0. A sum like $\sum_{i=1}^6 v_i$ is a mathematical analogue of a loop where we accumulate the sum one term at a time, giving $v_1 + v_2 + \dots + v_6$.

Our three statistics are then defined as

- $\text{score}(v) = \sum_{i=1}^6 v_i$;
- $\text{hits}(v) = \sum_{i=1}^6 \{v_i > 0\}$, where the indicator $\{v_i > 0\}$ equals 1 if $v_i > 0$ or 0 otherwise; and
- $\text{bulls}(v) = \sum_{i=1}^6 \{v_i = 10\}$, where the indicator $\{v_i = 10\}$ equals 1 if $v_i = 10$ or 0 otherwise.

The first adds the score from each arrow; the second adds 1 for each arrow if it scores more than 0, else adds 0; and the third adds 1 for each arrow that hits the bulls-eye. Compare these closely to the code versions above.

The transformed FRPs $\text{score}(A)$, $\text{hits}(A)$, and $\text{bulls}(A)$ represent the total score, the number of hits, and the number of bull's eyes the archer shoots for the round. All three FRPs depend on the value of A when it is available. Once the round is complete, A is activated (though not necessarily observed) and so are these three FRPs.

Puzzle 23. For each $s \in [0..60]$, define a condition (i.e., Boolean statistic) that the archer scores (strictly) more than s in total over all six arrows. Give code and mathematical versions of this statistic. Consider using the factory trick.

The previous example fits the framework of Figure 1.12. A is the Data FRP; its

value is the data with which our questions will be answered. The transforms $\text{score}(A)$, $\text{hits}(A)$, and $\text{bulls}(A)$ are feature FRPs whose values will answer our questions. We want to predict these answers.

But the example also reveals a need for tweaking our framework slightly. In a real round of archery, the archer takes one shot at a time, so if we were watching the evolution of this random system, we would obtain *partial information* about the data. That is, we might see the results of the shots as they occur (and thus before all the shots have been taken).

Such partial information would affect our predictions. For instance, if you observe that the archer's first shot is a bull's eye, your subsequent predictions about $\text{score}(A)$, $\text{hits}(A)$, and $\text{bulls}(A)$ will be higher than they would have been without that information.

This relates to the Element FRPs in Figure 1.12, which we did not describe in the example. Here, the Element FRPs are the FRPs representing the archer's individual shots, call them $A_1, A_2, \dots, A_6$. Each has dimension 1 and size 11, with possible values $0, 1, \dots, 10$. A is built from these pieces by what we will call an *independent join*, as described in Chapter 4:

$$A = A_1 \star A_2 \star A_3 \star A_4 \star A_5 \star A_6.$$

It is easier to understand, model, and analyze A through these building blocks, which is why we have use Element FRPs in the framework.

The tweak to our framework is that we sometimes want to derive feature FRPs from some of the Element FRPs that will be observed *before* the Data FRP is observed. For example, we can construct an FRP, call it T , that represents the archer's first three shots. T is built from A_1, A_2, A_3 (as an independent join). If we want to make predictions using (or about) the total score from the three shots, we can define a transformed FRP $\text{Sum}(T)$ that represents this partial score. It is a perfectly valid and useful Feature FRP whose value may be observed before the data are fully available, and it depends on only some of the Element FRPs.

Figure 2.8 updates our framework slightly with a dashed arrow from the Element FRPs to the Feature FRPs, allowing for the possibility that some Feature FRPs may be defined in terms of only some elements and thus may be observed before the Data FRP produces its value. This lets us use partial information in our analysis and predictions while the random process is still evolving.

The two paths (dashed and undashed) in Figure 2.8 from Element FRPs to Feature FRPs give the same information. Any transform of T by a statistic, for

instance, can be expressed as a transform of A by a different but related statistic. The role of the dashed path is to emphasize the timing of the information we obtain.

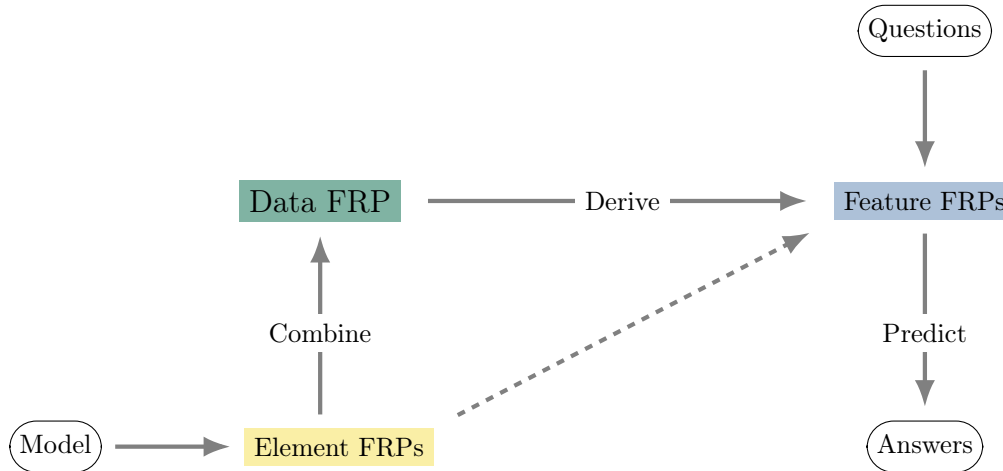


FIGURE 2.8. Update to Figure 1.12 that shows the role of the model, motivating questions, and predictions of their answers. The dashed arrow indicates that some feature FRPs may be defined in terms of selected components and thus may be activated before the Data FRP is.

Figure 2.8 updates the schematic in Figure 1.12 to show the role of the model, our motivating questions, and prediction of the answers to those questions. The Model is a collection of assumptions that we use to build the Element FRPs by specifying their Kinds. Using transforms and joins, we Combine the Elements into an FRP that represents all the data that will be observed from the system, the Data FRP. The Questions we want to answer with these data focus our attention on specific features of the data, and we transform the Data FRP with statistics (along with observations) to build the FRPs that represent these features. In many cases, we observe partial information about the features as the system evolves. The dashed arrow in the Figure indicates that some Feature FRPs may also be expressed as transforms of selected Element FRPs and thus can activate before the Data FRP does. We then use risk-neutral prices (expectations) to predict the values of the Feature FRPs and so predict the Answers to our Questions. We will consider the role of the Model more closely in the next section.

TRANSFORMS WITH INLINED STATISTICS. In `frplib`, we have two equivalent ways to express the transform of an FRP X by a statistic `psi`: either $X \sim \text{psi}$ or `psi(X)`. The former makes it clear that transformation is an *operation* on the FRP; the latter mimics our mathematical notation, coopting the syntax of functional

evaluation for this distinct operation. Whether in code as `psi(X)` or in mathematics as $\psi(X)$, we have seen that this does *not* mean evaluating the statistic with FRP as an argument. Rather it stands as a meaningful shorthand for plugging the value of the FRP into the statistic *when it is available*, which gives us a new FRP.

When the statistic is particularly simple or common, we take this notational conceit one step further in mathematical use and **inline** the statistic in this expression. This means that we write the statistic in terms of the FRP itself. For example, consider three simple statistics

$$\begin{aligned}\psi(x) &= x^2 \\ \varphi(x) &= 4x - 3 \\ \xi(x) &= e^{-2x}\end{aligned}$$

If X is a 1-dimensional FRP, then we can look at the transformed FRPs: $\psi(X)$, $\varphi(X)$, and $\xi(X)$. This is fine and works well. But often with such simple statistics, we do not want to bother naming a statistic just to write out the transform so instead we can write

$$\begin{aligned}\psi(X) &\text{ as } X^2 \\ \varphi(X) &\text{ as } 4X - 3 \\ \xi(X) &\text{ as } e^{-2X}\end{aligned}$$

We say that the right side gives the transformation in *inlined form* and allow both the left and right sides here to be used interchangeably.

Note what an expression like X^2 is really saying. It looks like we are “squaring” an FRP, just as $\psi(X)$ looks like we are evaluating ψ *at* an FRP. *But we are not*. Both are short-hands, using the syntax of evaluation to represent the transformation by a statistic.

Puzzle 24. Use the following statistics in inlined form to transform the scalar FRP X .

- $\langle t \rangle \mapsto 2 - 3t + 4t^2$
- $\langle x \rangle \mapsto 4 \cos(2\pi X - 3)$
- $\langle s \rangle \mapsto 7(t - 3)^3 + 2$
- $\langle s \rangle \mapsto \log(X + 4)$

Assume that the statistics are compatible with X in each case.

There are two types of functions that are especially convenient to use in inlined form: projections (cf. Section 15.2) and indicators (cf. Chapter 12).

For a positive integer i , the projection function proj_i extracts the i th component from a tuple of dimension $\geq i$. For instance, $\text{proj}_2(\langle 10, 20, 30 \rangle) = 20$ and $\text{proj}_1(\langle 99, 0 \rangle) = 99$. If Y is an FRP of dimension d and $i \in [1..d]$, the transformed FRP $\text{proj}_i(Y)$ is the scalar FRP representing Y 's i th component. We write the inlined form of this FRP as Y_i , and call the FRPs $Y_1, Y_2, \dots, Y_d$ the **scalar components** of Y .

For example, if Z is a 2-dimensional FRP and statistic $\zeta(a_1, a_2) = -a_1 a_2$, we write the transformed FRP $\zeta(Z)$ in inlined form as $-Z_1 Z_2$.

Indicators are functions that return either 0 or 1. One way that we express the evaluation of an indicator function is by putting a Boolean expression in *Iverson braces*.¹⁹ The result has the value 1 if the expression is true and 0 if the expression is false. Thus, we have some variable u and write (i) $\{u > 0\}$, (ii) $\{u = 2\}$, and (iii) $\{u < -2 \vee u > 2\}$, we get (i) 1 if u is positive else 0, (ii) 1 if u equals 2 else 0, (iii) 1 if u is less than -2 or greater than 2, else 0. We use indicators as statistics to embody the question of whether a particular observable outcome occurs.

¹⁹See page 614.

For example, if S is a scalar statistic, we write the statistics $\langle u \rangle \mapsto \{u > 0\}$, $\langle u \rangle \mapsto \{u = 2\}$, and $\langle u \rangle \mapsto \{u < -2 \vee u > 2\}$ in inlined form as $\{S > 0\}$, $\{S = 2\}$, and $\{S < -2 \vee S > 2\}$, respectively. Here, $\{S > 0\}$ is the transform of S whose value is 1 when S is positive and 0 otherwise. $\{S < -2 \vee S > 2\}$ is the transform of S whose value is 1 when S is less than -2 or bigger than 2 and 0 otherwise. And do forth.

Note that we can use the inlined forms of other statistics within the Iverson braces as well. For instance, using the 2-dimensional Z from above, we might consider the FRP $\{Z_1 + Z_2 > 10\}$, which the transform of Z that produces the value 1 when the sum of Z 's components is bigger than 10, or 0 otherwise.

Inlined Statistics. When working with a transform of an FRP X by a simple statistic, we often **inline** the statistic, writing the expression for the statistic in terms of X directly instead of defining an explicit statistic. This obviates the need to repeatedly name the simple statistics we use frequently.

We reserve inlined form for simple and commonly used statistics to save the effort and noise of repeatedly naming such basic functions. This includes simple arithmetic, projections, and conditions²⁰ that we use frequently. If the statistic is complicated enough, it is usually better to give the statistic a name.

²⁰For projections, see Section 2.4 below and Chapter 15 of Interlude F. For conditions, see Chapter 5. In Interlude F, Chapter 12 explains indicators in detail, and Chapter 11 on anonymous functions is also relevant.

Key Take Aways

A **transformed FRP** is created from a source FRP and a statistic. Its value is the output of the statistic when given the source FRP's value as input. The transformed FRP is built by connecting the source's All output port through an adapter that computes the statistic and into the input port of an empty FRP, which is immediately reconfigured. When the source FRP is fresh, so is the transformed FRP.

If the source FRP produces value v and the statistic is ψ , then the transformed FRP will produce value $\psi(v)$.

An FRP X and a statistic ψ are **compatible** if every possible value of X is a valid input to ψ .

If X and ψ are compatible, we denote the transformed FRP by $\psi(X)$. We say that $\psi(X)$ is the transform of X by ψ . (This is *not* the evaluation of ψ with X as an argument but rather an operation on X by ψ .) In `frplib`, we can represent the transform of FRP `X` by statistic `psi` by *either* `X ~ psi` or `psi(X)`.

For simple and familiar statistics (like squaring), we often write the transformed FRP with the statistic in **inlined form** (like X^2), where we do not name the statistic but write its formula in terms of the FRP.

Consideration of transforms leads to a refinement of our framework that relates our data, our model, our questions, and our predictions. See Figure 2.8.

2.3 Transformed Kinds

Looking back on Roulette (Examples 2.1 and 2.2), we have an FRP `R` that represents the pocket in which the ball eventually lands on a spin of the wheel and we have statistics for each play we might make. Consider again about the relation between FRPs `R` and `roulette.red(R)` (or equivalently `R ~ roulette.red`). For each possible value of `R`, we can look at Figure 2.1 to see if the corresponding pocket is red, and then we know the value that `roulette.red(R)` produces in that case. Table 2.2 shows the mapping explicitly, if somewhat laboriously. When `R` has value v , `roulette.red(R)` has value `roulette.red(v)`.

Now, if we do a large demo of `R`, simulating many spins of the wheel with `FRP.sample(n, R)` for some large `n`, we get the values of FRPs $R_1, R_2, \dots, R_n$ that are all clones of `R` and we get the values of the corresponding transformed FRPs

R	Pocket Color	<code>roulette.red(R)</code>
-1, 0	Green	-1
2, 4, 6, 8, 10, 11, 13, 15, 17, 20, 22, 24, 26, 28, 29, 31, 33, 35	Black	-1
1, 3, 5, 7, 9, 12, 14, 14, 16, 18, 19, 21, 23 25, 27, 20, 32, 34, 36	Red	1

TABLE 2.2. The possible values of `roulette.red(R)` as a function of the value of `R` in Roulette.

`roulette.red(R1)`, `roulette.red(R2)`, ..., `roulette.red(Rn)` that are all clones of `roulette.red(R)`. The demo collects the values of the former into a table by value (call it the source table), and we can collect the value of the latter by value (call it the transformed table). The transform relation between `R` and `roulette.red(R)` tells us what the transformed table looks like. Specifically, denote by p_i the proportion of spins in the source table for pocket i , for each $i \in [-1..36]$. If pocket i satisfies `roulette.red(i) = -1`, then all the spins in pocket i in the source table will be in the -1 (non-red) row of the transformed table. If pocket i satisfies `roulette.red(i) = 1`, then all the spins in pocket i in the source table will be in the 1 (red) row of the transformed table. Looking over all possible values, we see that the proportion of -1s in the transformed table is the sum of the proportions of *all non-red pockets*, i.e.,

$$p_{-1} + p_0 + p_2 + p_4 + p_6 + p_8 + p_{10} + p_{11} + p_{13} + p_{15} + p_{17} + p_{20} + p_{22} + p_{24} + p_{26} + p_{28} + p_{29} + p_{31} + p_{33} + p_{35},$$

the sum of p_i over all i for which `roulette.red(i)` equals -1. Similarly, the proportion of 1s in the transformed table is the sum of the proportions of *all red pockets*, i.e.,

$$p_1 + p_3 + p_5 + p_7 + p_9 + p_{12} + p_{14} + p_{16} + p_{18} + p_{19} + p_{21} + p_{23} + p_{25} + p_{27} + p_{30} + p_{32} + p_{34} + p_{36},$$

the sum of p_i over all i for which `roulette.red(i)` equals 1. This is true for any demo we run and no matter what the Kind of `R`. We can do this explicitly in `frplib`:

```
pgd> from frplib.examples.roulette import roulette
pgd> R = roulette()
pgd> demo_R = FRP.sample(10_000, R)
```



```
pgd> demo_Red = demo_R.transform(roulette.red)
```

```
pgd> demo_R
```

```
Summary of Output Values
```

```
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 304   | 3.04%      |
| 0      | 259   | 2.59%      |
| 1      | 231   | 2.31%      |
| 2      | 245   | 2.45%      |
| 3      | 246   | 2.46%      |
| 4      | 275   | 2.75%      |
| 5      | 236   | 2.36%      |
| 6      | 267   | 2.67%      |
| 7      | 261   | 2.61%      |
| 8      | 268   | 2.68%      |
| 9      | 268   | 2.68%      |
| 10     | 247   | 2.47%      |
| 11     | 219   | 2.19%      |
| 12     | 269   | 2.69%      |
| 13     | 254   | 2.54%      |
| 14     | 295   | 2.95%      |
| 15     | 259   | 2.59%      |
| 16     | 272   | 2.72%      |
| 17     | 258   | 2.58%      |
| 18     | 270   | 2.7%       |
| 19     | 295   | 2.95%      |
| 20     | 241   | 2.41%      |
| 21     | 241   | 2.41%      |
| 22     | 288   | 2.88%      |
| 23     | 234   | 2.34%      |
| 24     | 281   | 2.81%      |
| 25     | 281   | 2.81%      |
| 26     | 266   | 2.66%      |
| 27     | 237   | 2.37%      |
| 28     | 263   | 2.63%      |
| 29     | 296   | 2.96%      |
```

30	285	2.85%
31	257	2.57%
32	272	2.72%
33	248	2.48%
34	266	2.66%
35	290	2.9%
36	256	2.56%

```

+-----+-----+-----+
pgd> demo_Red
      Summary of Output Values
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 5285  | 52.85%     |
| 1      | 4715  | 47.15%     |
+-----+-----+-----+

```

Try this yourself. The values you get will differ because of randomness, but you can check here and in your results that the counts and proportions add up as we have described. Try it with statistics for various plays, using the same demo for R.

As the demo size gets larger and larger, the proportions in the tables better and better reflect the relative weights of the pockets in $\text{kind}(\mathbf{R})$ and $\text{kind}(\text{roulette.red}(R))$. Thus, we know how the Kind is transformed by `roulette.red`: if w_i is the weight on pocket i in $\text{kind}(R)$ for $i \in [-1..36]$, then $\text{kind}(\text{roulette.red}(R))$ is

$$\langle \rangle \begin{cases} a & \langle -1 \rangle \\ b & \langle 1 \rangle \end{cases}$$

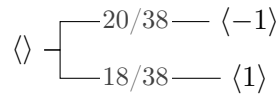
where

$$a = \sum_{i=-1}^{36} w_i \{\text{roulette.red}(i) = -1\}$$

$$b = \sum_{i=-1}^{36} w_i \{\text{roulette.red}(i) = 1\},$$

where the indicator term in Iverson braces selects only the weights for which the condition is true. For instance, if we make the reasonable assumption that all 38 pockets have the same weight, say $1/38$ for concreteness, then we get the following

Kind for `roulette.red(R)`:



Those two green slots ensure that the Red play has more ways to lose than to win.

What we have just done here is to show that *the relation between a source and transformed FRP creates a relation between their Kinds*. The Kind of a fresh FRP captures all of our knowledge about the FRP's value, so when we transform an FRP we can translate our knowledge about the source into knowledge about the transformed FRP. This *transformation of Kinds* is completely general and works for any Kind and compatible statistic.

Definition 8. A Kind k and a statistic ψ are **compatible** if every possible value of k (i.e., every leaf node) is a valid input to ψ .

This requires that the dimension of k is equal to a codimension of ψ .

The transformation of Kinds parallels the transformation of FRPs. We have a source Kind and a statistic and produce a transformed Kind. Physically, we read the transformed Kind off the metallic display of the transformed FRP. Algebraically, we express the transformation with the same two binary operators but using Kinds as operands in place of FRPs. In this context, the **transform operator** $\blacksquare \hat{\blacksquare}$ takes a Kind in the left hole²¹ and a statistic in the right hole, and the **overloaded evaluation operator** $\blacksquare(\blacksquare)$ takes a statistic in the left hole (before the parentheses) and a Kind in the right hole (within the parentheses). The arrow $\hat{}$ is intended to evoke the wiring connection in Figure 2.7 from source to transformed through the statistic.

²¹Throughout, $\blacksquare$ represents a hole to be filled by an argument or operand. See also Chapter 11.

So if k is a Kind and ψ is a compatible statistic (`psi` in `frplib`), we can write the transforms mathematically as either

$$k \hat{} \psi \qquad \text{or} \qquad \psi(k)$$

and in `frplib` as either

$$k \hat{} \text{psi} \qquad \text{or} \qquad \text{psi}(k)$$

Notice the exact parallel with how we use these operators to transform FRPs, as shown in the previous section (page 118).

In mathematical work, we almost always use the overloaded evaluation operator form $\psi(k)$, but in code, we use both forms frequently for reasons we will see. Like the notation $\psi(X)$, $\psi(k)$ looks like we are passing the Kind to the statistic, but we are *not*. We are reusing the *syntax* of evaluation as an operator. See the discussion of this point in the previous section on page 126.

We can also express transformation as a simple procedure:

```
Kind  $\ell$ :
     $x \leftarrow k$ 
    return  $\psi(x)$ 
```

that defines the Kind $\ell = k \hat{\ } \psi$. In `frplib`, this looks almost exactly the same, using the syntax of a decorated function definition and replacing the yield operator $\leftarrow$ with `= yield`:

```
@kind
def ell():
    x = yield k
    return psi(x)
```

These procedures exactly parallel the FRP procedures for transforming with statistics (cf. page 118). But the interpretation of the $\leftarrow$ (`yield`) lines requires more explanation. With an FRP X , the line $x \leftarrow X$ (`x = yield X`) extracts *the* value of X , when it is available, and binds it to the variable x . But a Kind does not have just one value; it has a value for every leaf node. In a Kind procedure, the $x \leftarrow k$ extracts *every* value (leaf node) of k *in parallel* and executes the ensuing steps for *each* value. The `return` statement gives the corresponding value in the transformed Kind, and the weights are automatically combined for all source values that produce the same transformed value. So think of the procedure for ℓ as a rule: if x is any value of k , it becomes the value $\psi(x)$, and all values that transform in the same way have their weights automatically added together.

To illustrate the meaning of $\leftarrow$ (`yield`) further, we will consider a more complicated example. Technically, the procedure for the Kind `twoDiceSum` that follows combines multiple operations including a transform, but this example should help

clarify.

Kind `twoDiceSum`:

```
roll1 ← uniform(1, 2, ..., 6)
roll2 ← uniform(1, 2, ..., 6)
return roll1 + roll2
```

The Kind `uniform(1, 2, ..., 6)` is the Kind of an FRP representing the roll of a balanced 6-sided die. As the name suggests, `twoDiceSum` is the Kind of an FRP representing the sum of two rolls of balanced, 6-sided dice. Why?

The first line of the procedure binds `roll1` to each possible roll of the first die, i.e., 1 through 6. For each value of `roll1`, the remaining steps are executed. The second line of the procedure binds `roll2` to each possible roll of the second die, also 1 through 6. Any such combination of values becomes the transformed value `roll1 + roll2` in the resulting Kind. Over all 36 such combinations, we get 11 unique values, and the procedure properly adds up the weights of those combinations that map to the same value. There are more ways to get 7 than 2, for instance, and if you look at the Kind, you can see that.

Another way to think about this procedure is as a rule: “For any value `roll1` from the Kind `uniform(1, 2, ..., 6)` and for any value `roll2` from the Kind `uniform(1, 2, ..., 6)`, we assign their combined weights to the value `roll1 + roll2` in the transformed Kind.”

Puzzle 25. Translate the procedure defining the Kind `twoDiceSum` into an `frplib` Kind procedure.

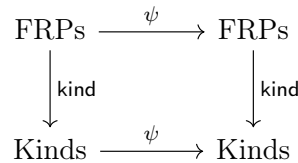
The values that we bind at one step in a procedure can be used anywhere in later steps of that procedure. In the following puzzle, we are looking at the Kind of an FRP representing a simple system. I flip a balanced coin. If it is tails, you win \$3. If it is heads, I roll a balanced 6-side die and you get the value of that roll in \$.

Puzzle 26. Interpret the following Kind procedure (in `frplib` code) in terms of the system just described:

```
@kind
def maybe_roll_dollars():
    flip = yield choice(0, 1)
    win = yield constant(3) if flip == 0 else uniform(1, 2, ..., 6)
    return win
```

Notice that the Line 4 depends on the value of `flip`, and that Lines 4 and 5 are executed for both possible values.

A critical point is that the transformation of Kinds is *consistent* with the transformation of FRPs: **we get the same transformed Kind if we first transform the source FRP and then take its Kind or if we first take the Kind of the source FRP and then transform that**. This is illustrated by the diagram



The nodes in this graph represent the set of FRPs and the set of Kinds; the edges represent operations that map one set into the other. The horizontal edges (labeled ψ) represent transformation by the statistic ψ , either transformation of the FRPs or transformation of the Kinds. The vertical edges (labeled `kind`) represent the mapping from an FRP to its Kind. Every path along the direction of the arrows represents a *composition* of these two operations. There are two paths in this graph from the top left to the bottom right. We can first transform an FRP by a statistic ψ then compute its Kind (with `kind`), or we can compute an FRPs Kind and then transform that Kind by the statistic ψ . *Both paths lead to the same result.*²²

In particular, if X is an FRP and ψ is a compatible statistic then

$$\psi(\text{kind}(X)) = \text{kind}(\psi(X)). \quad (2.1)$$

This means that: **the transform of X 's Kind by ψ equals the Kind of the transformed FRP $\psi(X)$.**

We can take equation (2.1) as the definition of a transformed Kind, yet it is also helpful to have a more operational definition. The key observation is that the possible values of a transformed FRP $\psi(X)$ are obtained from the possible values of X by applying ψ . So, we start by applying ψ to the value at every leaf node in `kind(X)`. However, it is possible that two distinct values of X map to the same value, so these two branches in `kind(X)` would map to a single branch in `kind( $\psi(X)$ )`.

If two distinct values v and v' of X map to the same value $u = \psi(v) = \psi(v')$, then the weights of v and v' should *add* into the weight of u for the Kind of $\psi(X)$. The reason for this is exactly the logic of the transformed demo table at the beginning of this section. If we ran a demo with a large number of clones of X and transformed them by the statistic ψ , then every time either v or v' appears as a value of X , we

²²See Chapter 13 in Interlude F. This requirement makes the graph a *commutative diagram*.

get the value u for $\psi(X)$. So the proportions for v and v' add into the proportion for u in what we see. This gives us a procedure for computing the transformed Kind: apply ψ to the values at the leaf nodes of $\text{kind}(X)$ and then combine the branches that map to the same value, adding their weights.

Definition 9. If X is an FRP with Kind k and ψ is a compatible statistic, then the transformed Kind, denoted $\psi(k)$, equals the Kind of the transformed FRP $\psi(X)$, as in equation (2.1).

To find $\psi(k)$, we create a tree where

1. the leaf nodes have values in the set of $\psi(v)$ for v in the values of k ; and
2. the weight associated with a value u is the sum of weights in k for all values v with $u = \psi(v)$.

That is, **we first apply the statistic ψ to each value of k and then combine branches that map to the same value, adding their weights.**

Figures 2.9 and 2.10 show simple examples of this process, in two steps. First, we apply the statistic to the value at each leaf node, showing the new value for each leaf across the blue bar, which represents the action of the statistics adapter. Second, if these transformed values are equal for any leaf nodes, we combine their branches and add their weights, giving the transformed Kind shown on the right of each Figure. In the first Figure, several values map to each of $\langle 1 \rangle$ and $\langle 2 \rangle$, and these branches are combined. In the second Figure, each of the original values maps to a distinct value, so no branches are combined.

The Kinds in both these Figures have width 1. The same idea applies to any Kind: apply the statistic to the values and then combine branches that map to the same value. But if the Kind has width bigger than 1, we first convert it to “canonical form”, – which is always a width 1 tree – as described in Chapter 3 before transforming it.

We can also express the transformation of a Kind k by a compatible statistic ψ mathematically using the **kernel** function of the Kind (see equation 1.1). This treats the Kind k as a function from values to their weights, returning 0 for any value that is not a leaf node of Kind. (The Kernel first reduces k to an equivalent width 1 tree to get a weight associated with each value.) If we define $\tilde{k} = \psi(k)$ to be the transformed Kind, then in terms of the kernels we have

$$\tilde{k}(v) = \sum_u k(u) \{v = \psi(u)\}, \quad (2.2)$$

where again the indicator term is 1 if $\psi(u) = v$ and 0 otherwise. The sum is over all

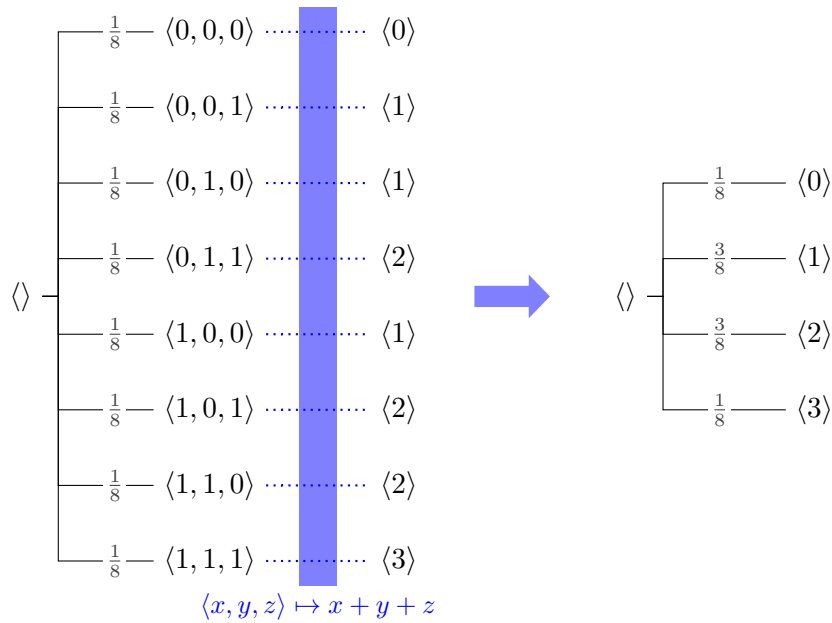


FIGURE 2.9. The transform of a three-dimensional Kind by the statistic that maps input $\langle x, y, z \rangle$ to $x + y + z$, shown in two steps. First, we apply the statistic to the values of the original Kind. Second, we combine branches that map to the same value, summing their weights to obtain the weight of the combined branch.

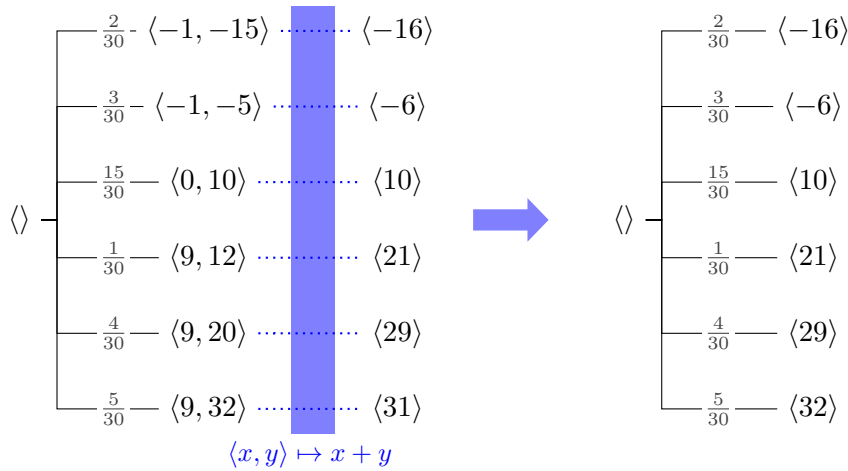


FIGURE 2.10. The transform of a two-dimensional Kind by the statistic that maps input $\langle x, y \rangle$ to $x + y$, shown in two steps. First, we apply the statistic to the values of the original Kind. Second, we combine branches that map to the same value, summing their weights, but here all values map to distinct values, so no branches are combined.

values of u but the summand is non-zero only when u is a possible value of k . This tells us that the weight on value v in the Kind $\psi(k)$ equals the sum of all weights on branches in k whose values map to v under ψ . This is easiest to visualize when the Kind k has width 1, but we will see in the next Chapter that it is always possible to convert the Kind to this form.

Equation (2.2) can be equivalently expressed in terms of FRPs X and Y , where $Y = \psi(X)$. Then, we denote the kernels of $\text{kind}(X)$ and $\text{kind}(Y)$ by k_X and k_Y , respectively. Translating (2.2), we get

$$k_Y(v) = \sum_u k_X(u) \{v = \psi(u)\}. \quad (2.3)$$

Example 2.5 Roulette, Part 3

Continuing with Roulette from Examples 2.1 and 2.2. Up to this point, we have focused on the random quantities – the spin FRP and its transforms by statistics representing plays. Now, we turn to the question of modeling the spin of the wheel, which reduces to giving the Kind of the spin.

A reasonable starting point is to assume *symmetry*: that all pockets on the wheel have equal weight. This is the Kind that is available in `frplib` as `roulette.kind`, and is the same as `uniform(-1, 0, ..., 36)`. Take a look at it in the playground, and notice that

```
pgd> R = roulette()
pgd> Kind.equal(roulette.kind, kind(R))
True
```

this is the Kind of the FRP returned by `roulette()`.

As a model for a game of chance, this uniform Kind makes sense, and we will see that casinos have a reason for keeping their wheels as close to this ideal as they can. Of course, a roulette wheel is a physical device, so perfect symmetry is an illusion, and the real proportions we see on repeated spins will almost certainly deviate from the uniform ideal. If the deviation is not too great, then this seems like a reasonable basis for our analysis. We will return to this issue shortly.

So, assume that a spin of the wheel (i.e., `R`) has uniform Kind `roulette.kind`. What do we know about the payoffs of different plays? Are some plays better

than others? Which?

Begin by considering the standard plays listed in Table 2.1. Note that as with FRPs, we can represent the transform of a Kind k by statistic psi as either $\text{psi}(k)$ or $k \sim \text{psi}$. Here are the transformed Kinds for a selection of standard plays. (See also the puzzle below.)

```
pgd> k = roulette.kind # Equal to kind(R), call it k for brevity
pgd> k ~ roulette.even
      ,---- 10/19 ---- -1
<> -|
      `---- 9/19 ----- 1

pgd> k ~ roulette.top_line
      ,---- 33/38 ---- -1
<> -|
      `---- 5/38 ----- 6

pgd> k ~ roulette.six_line(20)
      ,---- 16/19 ---- -1
<> -|
      `---- 3/19 ----- 5

pgd> k ~ roulette.column(2)
      ,---- 13/19 ---- -1
<> -|
      `---- 6/19 ----- 2

pgd> k ~ roulette.corner(8)
      ,---- 17/19 ---- -1
<> -|
      `---- 2/19 ----- 8

pgd> k ~ roulette.straight(12)
      ,---- 37/38 ---- -1
<> -|
      `---- 1/38 ----- 35
```

We now have one interpretation of these Kinds. For instance, in many of the spins of the wheel, the **Even** play will win \$1 about 9/10's as often as it will lose \$1. Or equivalently, **Even** will win \$1 in about 9/19ths of the spins and lose \$1 in about 10/19ths of the spins. Similarly, a **Column(2)** play will win \$2 in about 6/19ths of the spins and lose \$1 in about 13/19ths of the spins. We can draw analogous conclusions from the other Kinds as well.

Puzzle 27. Compute a transformed FRP for one of each of the standard plays listed in Table 2.1, look at its Kind and risk-neutral price (expectation). What do you conclude?

Note that for plays like **Column** and **Corner**, we specify the particular play by giving the lowest numbered pocket. For example, the following are statistics for a \$1 play: `roulette.column(2)` for the middle column starting with pocket 2, `roulette.corner(8)` for the four pockets 8, 9, 11, 12. For **Split**, you need to specify the adjacent positions in order, e.g., `roulette.split(24,27)` or `roulette.split(23,24)`. Enter `info(roulette.split)` etc. in the playground for details.

Notice that these results do not really recommend either an **Even** or **Column(2)** play. Suppose we run 19,000,000 spins. Then **Even** plays will win \$1 in about 9,000,000 of those spins and lose \$1 in about the other 10,000,000 – for a roughly \$1,000,000 loss! **Column(2)** will win \$2 in about 6,000,000 of those spins and will lose \$1 in about 13,000,000 of those spins, for a net loss of roughly $2 \cdot 6,000,000 - 13,000,000 = -1,000,000$ dollars.

The “risk-neutral price” of an FRP is a measure of how much the payoff of an FRP is worth,²³ based on a prediction of the FRPs value. In `frplib`, we compute risk-neutral prices with the **expectation operator** `E` (which is written mathematically as $\mathbb{E}$). We can do this for all of the above plays:

```
pgd> E(k ~ roulette.even)
-1/19
pgd> E(k ~ roulette.top_line)
-3/38
pgd> E(k ~ roulette.six_line(20))
-1/19
pgd> E(k ~ roulette.column(2))
-1/19
pgd> E(k ~ roulette.corner(8))
```

²³When we account for our individual aversion to the risk of losing larger amounts of money. See Chapter 6 for the full story.

```

-1/19
pgd> E(k ~ roulette.straight(12))
-1/19

```

The first thing to notice is that *all the prices are negative*. A negative price means that someone would have to pay *you* that amount to take the transaction. We can also view these as the predicted payoffs in dollars from an FRP with the corresponding Kinds. A negative payoff means that you lose that much money on the play. Second, the numbers fit with our rough calculations above. With 19,000,000 spins, we saw we would lose about \$1,000,000 on Even plays, which is about $-1/19$ payoff per spin on average. Finally, some bets are worse than others, but in terms of our predicted payoff, others seem interchangeable. It's almost as if the casino arranged things in their favor.

Perhaps we might think we can get a better outcome if we combine plays and/or bet different amounts. For instance,

```

pgd> comb = 10 * roulette.even + 5 * roulette.corner(25) +
          20 * roulette.straight(4) + 50 * roulette.column(2)
pgd> k ~ comb
,---- 13/38 ---- -85
|---- 10/38 ---- -65
|---- 1/38 ----- -40
|---- 1/38 ----- -20
<> -+---- 5/38 ----- 65
    |---- 5/38 ----- 85
    |---- 1/38 ----- 110
    |---- 1/38 ----- 130
    `---- 1/38 ----- 655
pgd> E(k ~ comb)
-85/19

```

Thinking about it for a bit, we can see that this hope is in vain. Again, imagine that we do 19,000,000 spins of the wheel. Placing a combined play like this means we place all of the bets included, though with different amounts. But each of the bets has a negative risk-neutral price, meaning we will lose money on average on all of the bets in our 19,000,000 spins. Losing money on multiple bets cannot lead to a gain. Mathematically, we will see that this is reflected in

a key idea: the expectation operator E (mathematically denoted $\mathbb{E}$) is a *linear operator*. We will return to this soon.

Because any play in roulette can be rewritten as a combination of straight plays on different pockets, we see that every play in roulette is on average a loser for the player and a winner for the casino. This gives us some insight into why casinos might want to ensure that their wheels give a Kind as close to uniform as possible. Suppose that because of physical irregularities, some of the pockets have greater weight than others. How does this change our assessment of the plays?

We can use the function `roulette.biased_kind` to investigate this. For instance, the call `roulette.biased_kind(a, pockets=[2, 4, 6, 8])` creates a Kind where pockets 2, 4, 6, and 8 have weight a and all other pockets have weight 1. Try it:

```
pgd> a = symbol('a')
pgd> k = roulette.biased_kind(a, pockets=[2, 4, 6, 8])
pgd> k ^ roulette.even
# omitted
pgd> E(k ^ roulette.even)
# omitted
pgd> substitution( E(k ^ roulette.even), a=2 )
# omitted
```

Here, we keep the relative weight of 2, 4, 6, and 8 unspecified, using a “symbol” a . In the last line, we substitute in the value 2 for a to get a definition number. You can view this Kind and these expectations in the playground, but here let’s reason it out using what we know about the transform of Kinds by a statistic.

The transformation of a Kind by a statistic first transforms all the values by the statistic and then combines branches that map to the same value, adding their weights. The `roulette.even` statistic maps 2, 4, 6, 8, 10, ..., 36 to 1 and all other pockets to -1. The weights mapping to 1 add up to $4a + 14$ and the weights mapping to -1 add up to 20. The Kind `k ^ roulette.even` thus looks like

$$\langle \rangle - \begin{cases} 20 & \langle -1 \rangle \\ 4a + 14 & \langle 1 \rangle \end{cases}$$

For reasons we will see, the expectation of this Kind (i.e., its risk-neutral price) is

$$\frac{4a - 6}{4a + 34},$$

which is positive when $a > 3/2$. For instance, $a = 2$, we have an expected payoff for the **Even** play of $1/21 \approx 0.0476$. If $a = 1.6$, we get about 0.01. Look at the Kind for these values of a to get a feel for how big of a difference this makes, e.g.,

```
pgd> substitution( k ~ roulette.even, a=2 )
pgd> substitution( k ~ roulette.even, a=1.6 )
```

Note that these Kinds will be in canonical form (i.e., with normalized weights) unless you call `unnormalized` on them.

Puzzle 28. Can you find a play that gives a better expected payoff with this particular biased wheel?

Finally, you might wonder if we can make roulette advantageous to the player with some “system” for choosing plays on *multiple* spins based on the results of previous spins. The short answer is no, and we will see why later in the book.

One note on notation here. We can apply the expectation operator to either Kinds or FRPs. The expectation of an FRP depends only on the FRP’s Kind and not on its specific value; all FRPs with the same Kind have the same expectation. So the expectation of an FRP is just the expectation of its Kind. In the previous example, for instance, we could write interchangeably `E(kind(R) ~ roulette.even)` or `E(R ~ roulette.even)`.

Example 2.6. In Example 2.3, we considered an FRP that represents a random point chosen from the corners, edge midpoints, face centers, and center of a cube of edge-length 2 centered on the origin. We then defined the transform of this FRP by the statistic that computes the distance of a point to the origin. Assuming that the source FRP has a Kind with equal weights on every branch, we compute the transformed Kind as shown in Figure 2.11. This takes the two steps described earlier: apply the statistic to the values at every leaf node and then combine branches that map to the same value, adding their weights.

The result is consistent with our intuition. There are 8 corners, 12 edges, 6 faces, and 1 center, and the Kind shows that a large demo of the FRPs with the transformed Kind will give the distances from the origin to corner, edge midpoint, face center, and center in just these proportions.

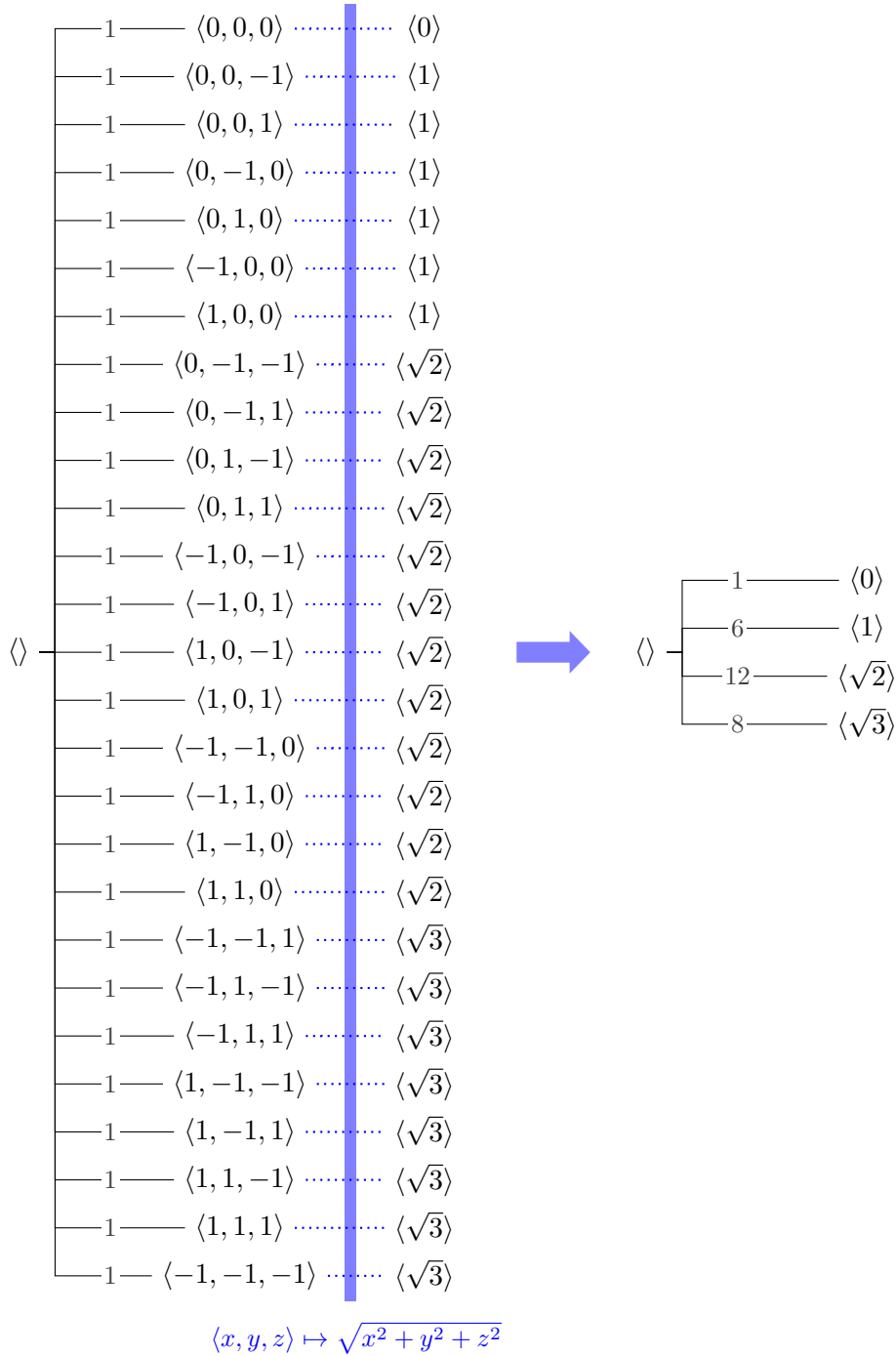
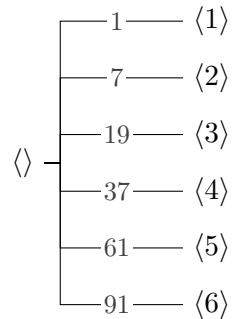


FIGURE 2.11. Computing the transformed Kind in Example 2.6.

Example 2.7. The Kind in Figure 1.10 describes FRPs that represent the roll of three, balanced, six-sided dice. Consider two questions: (i) What is the largest value among the dice? (ii) How many of each value did we see in the roll? We define statistics to capture these questions. Each will take a 3-tuple as input giving the values of the red, green, and blue dice. For (i), we want to compute the maximum of the three dice; this is a built-in statistic in `frplib`, called `Max`, so we just use that name here with type $3 \rightarrow 1$.

For each leaf node in Figure 1.10, we compute the maximum of the dice. Because there are only 6 possible maximums and 216 leaf nodes, multiple branches will have the same maximum. Combining those branches and adding their weights gives us the Kind:



Try this transformation *by hand* to get a feel for the operation. There is one branch with maximum 1 (all three dice equal to 1), seven with maximum 2 (three with a 2 and two 1s, three with a 1 and two 2s, and one with three 2s), and so on up to 91 branches with a maximum of 6.

For question (ii), our statistic ξ will have type $3 \rightarrow 6$, returning a tuple v where v_i counts the number of dice that show i pips on top, for $i \in [1..6]$. Define ξ mathematically by

$$\begin{aligned} \xi(r, g, b) = & \langle \{r = 1\} + \{g = 1\} + \{b = 1\}, \{r = 2\} + \{g = 2\} + \{b = 2\}, \\ & \{r = 3\} + \{g = 3\} + \{b = 3\}, \{r = 4\} + \{g = 4\} + \{b = 4\}, \\ & \{r = 5\} + \{g = 5\} + \{b = 5\}, \{r = 6\} + \{g = 6\} + \{b = 6\} \rangle. \end{aligned}$$

Here, terms like $\{g = 3\}$ are *indicators*, as described in Chapter 12 of Interlude F. For instance, $\{g = 3\}$ is 1 when $g = 3$ and 0 otherwise. The term in the i th component of the tuple returned by ξ counts the number of dice equal to i .

We can define this statistic in the playground by defining this as a Python

function. Again, we use a `@statistic` “decorator” to convert the function to a `Statistic` object with useful properties.

```
@statistic(dim=6)
def xi(r, g, b):
    "Counts the number of dice (r, g, and b) with each value."
    return [(r == i) + (g == i) + (b == i) for i in irange(1,6)]
```

We can pass `xi` individual values for the three dice or a single 3-tuple,²⁴ and using the FRP `Roll` from earlier, we can compute the transformed FRP `xi(Roll)` and its Kind with *either* `kind(xi(Roll))` or `xi(kind(Roll))` by equation (2.1).

²⁴Because `xi` has three explicit parameters, the playground can infer `codim=3` automatically. With a single parameter, it helps to give `codim` explicitly. If `xi` instead had parameters `r, g, b, *more`, it would have type $n \rightarrow 6$ for every $n \in [3..)$.

Key Take Aways

The transform of an FRP by a statistic induces a consistent transform operation on Kinds. If X is an FRP and ψ is a compatible statistic then

$$\psi(\text{kind}(X)) = \text{kind}(\psi(X)). \quad (2.4)$$

That is: **The transform of X ’s Kind by ψ equals the Kind of the transformed FRP $\psi(X)$.**

In general, if k is a Kind, we write either $k \hat{\psi}$ or $\psi(k)$ for the Kind k transformed by ψ . In `frplib`, we denote the transform of a Kind `k` by statistic `psi` by *either* `k ^ psi` or `psi(k)`, interchangeably. We can also transform Kinds using a Kind procedure.

To find $\psi(k)$, we create a tree where

1. the leaf nodes have values in the set of $\psi(v)$ for v in the values of k ; and
2. the weight associated with a value u is the sum of weights in k for all values v with $u = \psi(v)$.

That is, **we first apply the statistic ψ to each value of k and then combine branches that map to the same value, adding their weights.**

We can express this in terms of the kernel function of Kind k and transformed Kind $\tilde{k} = \psi(k)$: to be the transformed Kind,

$$\tilde{k}(v) = \sum_u k(u) \{v = \psi(u)\}, \quad (2.5)$$

Expressed in terms of FRPs with $Y = \psi(X)$, this translates to the equivalent expression:

$$k_Y(v) = \sum_u k_X(u) \{v = \psi(u)\}, \quad (2.6)$$

where k_X and k_Y are the kernels of $\text{kind}(X)$ and $\text{kind}(Y)$.

2.4 Projections and Marginals

If you have an FRP representing a random graph, you might focus on a subgraph. For an FRP representing a random image, you might have questions about one or more particular pixels. When modeling repeated roulette spins, you might analyze the outcome of bets on only some of those spins. This operation – *extracting* selected parts of a value – is so common and useful that it merits special attention. Statistics that extract one or more components from a tuple are called *projections*.

Definition 10. A **projection** is a statistic that maps a tuple to a tuple of smaller dimension containing only specified components of the original tuple.

A projection ψ of type $n \rightarrow m$, with $m \leq n$, has the form

$$\psi(\langle x_1, x_2, \dots, x_n \rangle) = \langle x_{i_1}, x_{i_2}, \dots, x_{i_m} \rangle \quad (2.7)$$

where $1 \leq i_1 < i_2 < \dots < i_m \leq n$ are the indices of the selected components.

Remember that we elide the distinction between lists of length 1 and scalars, so we can write a projection of type $n \rightarrow 1$ as $\psi(x) = x_i$ for some $i \in [1..n]$.

Mathematically, we denote projection statistics specially: using the name **proj** with the selected indices in the subscript, as described in detail Chapter 15 of Interlude F. For instance,

$$\begin{aligned} \text{proj}_3(\langle 10, 20, 30, 40, 50 \rangle) &= 30 \\ \text{proj}_{3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 30, 50 \rangle \\ \text{proj}_{1,3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 30, 50 \rangle. \end{aligned}$$

If we want to project onto a *range* of components, say all the components from index i up to and including index j , we write the range as $i..j$ in the subscript. If either

endpoint is missing, the range extends all the way to the corresponding end.

$$\begin{aligned}\text{proj}_{1..3}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20, 30 \rangle \\ \text{proj}_{3..}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 30, 40, 50 \rangle \\ \text{proj}_{..3}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20, 30 \rangle.\end{aligned}$$

We also use the functions with base name $\overline{\text{proj}}$ to indicate a projection that *excludes* the listed components. For instance,

$$\begin{aligned}\overline{\text{proj}}_3(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20, 40, 50 \rangle \\ \overline{\text{proj}}_{3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20, 40 \rangle \\ \overline{\text{proj}}_{1,3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 20, 40 \rangle \\ \overline{\text{proj}}_{1,2,3,5}(\langle 10, 20, 30, 40, 50 \rangle) &= 40 \\ \overline{\text{proj}}_{2..4}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 50 \rangle \\ \overline{\text{proj}}_{3..}(\langle 10, 20, 30, 40, 50 \rangle) &= \langle 10, 20 \rangle \\ \overline{\text{proj}}_{..4}(\langle 10, 20, 30, 40, 50 \rangle) &= 50.\end{aligned}$$

In `frplib`, we access projections using the pre-defined statistics `Proj[indices...]`, where the indices start *counting from 1*. For example:

```
pgd> x = tup(10, 20, 30, 40, 50)
pgd> x
<10, 20, 30, 40, 50>
playground> Proj[3](x)
30
playground> Proj[3,5](x)
<30, 50>
playground> Proj[1,3,5](x)
<10, 30, 50>
```

(We can pass ordinary Python tuples to these statistics, but we use the function `tup` to obtain the more flexible tuples that are used for the values of FRPs and Kinds.) Between the brackets, `Proj` can accept multiple individual indices or a sequence of indices, like `Proj[(1,3,5)]` or `Proj[range(2,5)]`. Note that `range(a,b)` includes `a` but not `b`. These also accept Python “slices”, where `i : j` consists of indices from `i` up to but *not including* `j` (or to the end if `j` is excluded) and where `i : j : k` consists of indices from `i` up to but not including `j` skipping by `k`. So, `Proj[1:5:3]` is the

same as `Proj[1,4]` and `Proj[1:6:2]` is the same as `Proj[1,3,5]`.

For convenience, we can use *negative indices* to indicate component indices *starting from the end*, with `-1` the last component, `-2` the second to last, and so forth.

```
pgd> Proj[-1](x)
50
pgd> Proj[-4,-3](x)
<20, 30>
pgd> Proj[2,-1](x)
<20, 50>
```

The family `Projbar` is also defined to mimic the $\overline{\text{proj}}$ functions:

```
pgd> Projbar[3](x)
<10,20,40,50>
pgd> Projbar[3,5](x)
<10,20,40>
pgd> Projbar[1,3,5](x)
<20, 40>
pgd> Projbar[1,2,3,5](x)
40
```

These also accept negative indices to count from the end.

As with any other statistic, we can use projections to transform FRPs and Kinds. But `frplib` also supports a shorthand using the `[]` operator with indices *directly* on the FRP or Kind. For example, if we make the following definitions

```
pgd> bits = uniform(0,1) ** 7    # A 7-dimensional Kind of random bits
pgd> B = frp(bits)
pgd> first_bit = Proj[1]
pgd> odd_bits = Proj[1:8:2]
pgd> even_bits = Proj[2:8:2]
pgd> last_bit = Proj[-1]
pgd> last_two_bits = Proj[6:]
```

then we can apply the statistics in several equivalent ways

```
pgd> B
An FRP with value <0, 1, 1, 1, 1, 0, 1>
pgd> # Equivalent ways to apply first_bit (common value at end)
```

```
pgd> B ^ first_bit
pgd> first_bit(B)
pgd> B ^ Proj[1]
pgd> Proj[1](B)
pgd> B[1]
An FRP with value <0>
```

```
pgd> # Equivalent ways to apply odd_bits (common value at end)
pgd> B ^ odd_bits
pgd> odd_bits(B)
pgd> B ^ Proj[1:8:2]
pgd> Proj[1:8:2](B)
pgd> B[1:8:2]
An FRP with value <0, 1, 1, 1>
```

```
pgd> # Equivalent ways to apply last_bit (common value at end)
pgd> B ^ last_bit
pgd> last_bit(B)
pgd> Proj[-1](B)
pgd> Proj[7](B)
pgd> B[-1]
pgd> B[7]
An FRP with value <1>
```

Try these out yourself, and examine the results.

Puzzle 29. In the playground, demonstrate as many equivalent ways as you can to transform the FRP `B` by the statistics `even_bits` and `last_two_bits`.

The same approaches work with Kinds as well as FRPs. For instance, `bits` above is the Kind of `B`, and the following expressions are equivalent for producing the transformed Kind with `last_two_bits`:

```
pgd> bits ^ last_two_bits
pgd> last_two_bits(bits)
pgd> bits ^ Proj[6:]
pgd> Proj[6:](bits)
pgd> Proj[6,7](bits)
```

```

pgd> Proj[-2:](bits)
pgd> Proj[-2,-1](bits)
pgd> ProjBar[1:6](bits)
pgd> bits[6:]
pgd> bits[-2,-1]
pgd> bits[6,7]
      ,---- 1/4 ---- <0, 0>
      |---- 1/4 ---- <0, 1>
<> -|
      |---- 1/4 ---- <1, 0>
      `---- 1/4 ---- <1, 1>

```

While this may initially seem like a lot of choices, there are just a few shorthands to keep in mind. The $\sim$ operator (think arrow) transforms an object (FRP, Kind, statistic) on the left side by a statistic on the right side. This is convenient to use when the statistic is derived from a factory or from a statistic expression (as described earlier). A shorthand for this *looks like* a function call, where we pass the object to be transformed to the statistic. We tend to use this form when statistics have names because it matches our mathematical notation (e.g., $\psi(B)$) and captures the spirit of what the transform means. Because statistics are objects, they can be stored in variables and thus given names; when we use two expressions for the same statistic, the results are interchangeable. Thus for instance, `last_two_bits(bits)` and `Proj[6:](bits)` are the same because `Proj[6:]` is the statistic that we named `last_two_bits`. And finally for convenience, we allow direct application of the `[]` indexing operator to FRPs and Kinds, equivalent to the corresponding `Proj`, to make this common operation easier. Note that the indexing scheme – based on Python – provides multiple ways to generate the same set of indices, with lists or slices or counting from the end and so forth.

As Figure 2.8 suggests, we often build FRPs (and Kinds) that embody complicated information in high-dimensional tuples, so it is very common to work with transforms by projection. To express the relation between an FRP and another derived by projection and between a Kind and another derived by projection, we use the specific label *marginal*.

Definition 11. If X is an FRP and $X' = \text{proj}_{i_1, i_2, \dots, i_m}(X)$, then X' is an FRP obtained by applying a projection statistic to X , then we call X' a **marginal**

(FRP) of X . It is specifically identified by the indices $i_1, i_2, \dots, i_m$.

If k is a Kind and $k' = \text{proj}_{i_1, i_2, \dots, i_m}(k)$, then k' is Kind obtained by applying a projection statistic to k , then we call k' a **marginal (Kind)** of k' . It is specifically identified by the indices $i_1, i_2, \dots, i_m$.

The process of transforming an FRP or Kind by a projection is sometimes called **marginalization**. The process of collecting the marginal FRPs for the projections onto every scalar component is called decomposing an FRP into its components.

Suppose X is an FRP of dimension n . We know that X produces a value that is a tuple of n numbers. The i^{th} component of X , for $i \in [1..n]$,²⁵ is just the FRP that gives us the i^{th} element of the list that X produces, which is exactly the value of the *transformed* FRP $\text{proj}_i(X)$. If we define $X_i = \text{proj}_i(X)$ for $i \in [1..n]$, we call $\langle X_1, X_2, \dots, X_n \rangle$ the *scalar components* of X .

²⁵ $[1..n]$ is an *increment*, the set of integers from 1 to n . See Section 9.2 in Interlude F.

Definition 12. If an FRP X has dimension n , then we can decompose it into **scalar components**, scalar FRPs $X_1, \dots, X_n$ with $X_i = \text{proj}_i(X)$ for each $i \in [1..n]$. We call $X_1, \dots, X_n$ the components FRPs of X , or just the components of X for short.

Puzzle 30. What are the components of the FRP B in the illustration on page 155? What are their Kinds?

Remember that, despite the special name, projections are just statistics, and thus satisfy all the properties of statistics. For instance, equation 2.1 holds, so if $X_i = \text{proj}_i(X)$, then $\text{kind}(X_i) = \text{proj}_i(\text{kind}(X))$. We get this Kind by taking the i th component of each value and combining all the branches (adding their weights) that have the same value of this component.

We can use what we know about the transformation of Kinds to understand the Kind of a projection more deeply. Recall that when we transform a Kind k by a statistic ψ , we apply ψ to each value of k and then combine branches that map to the same value, adding their weights. When ψ is a projection this takes an easy form. Let's start with a simple case:

```
pgd> k = weighted_pairs(((1, 2, 1), '1/2'), ((1, 1, 1), '1/4'),
...> ((3, 2, 1), '1/8'), ((3, 2, 3), '1/8'))
pgd> k
,---- 2/8 ---- <1, 1, 1>
|---- 4/8 ---- <1, 2, 1>
```

```

<> -|
    |---- 1/8 ---- <3, 2, 1>
    `---- 1/8 ---- <3, 2, 3>

```

We first consider $k \sim \text{Proj}[2]$ and carry out the transformation steps.

1. *Apply $\text{Proj}[2]$ to every value of k .*

We look at the second component in the tree above and, in the same order, get values 1, 2, 2, 2.

2. *Combine branches that map to the same value.*

The lower three branches of k all map to 2, so we combine these branches. The $\langle 1, 1, 1 \rangle$ branch is the only one mapping to 1.

3. *Add the weights of the combined branches, with the mapped valued, to form the transformed Kind*

The 1 branch keeps its weight $1/4$, and the three 2 branches have total weights $1/2 + 1/8 + 1/8 = 3/4$.

The resulting Kind is

```

,---- 1/4 ---- 1
<> -|
    `---- 3/4 ---- 2

```

We have made a branch for each projected value with weights equal to the sum over all the original branches. It is useful to see this transformation in terms of the Kind's *kernel function*. Define the transformed Kind $k_2 = \text{proj}_2(k)$. The kernel treats the Kind as a function that maps values to their canonical weights (or 0 if the value is not possible). The weight on value $\langle y \rangle$ in k_2 is obtained by adding the weights of all values in k with second component y :

$$k_2(y) = \sum_{x,z} k(x, y, z), \quad (2.8)$$

by equation (2.2). The values $\langle x, y, z \rangle$ we are summing over are all possible values with second component y . If $\langle x, y, z \rangle$ is not a possible value for k , the corresponding term in the sum will be 0.

Next consider $k \sim \text{Proj}[1, 3]$ and apply the same idea. All the possible projected values are $\langle 1, 1 \rangle$, $\langle 3, 1 \rangle$, and $\langle 3, 3 \rangle$. We add up the weights for all branches in K with each of these values giving the Kind:


```

,---- 6/8 ---- <1, 1>
<> -+---- 1/8 ---- <3, 1>
`---- 1/8 ---- <3, 3>

```

Puzzle 31. Express the $k \hat{=} \text{Proj}[1, 3]$ in terms of its kernel function analogously to the previous equation.

The same idea works in general. Let k be an n -dimensional Kind and let $1 \leq i_1 < i_2 < \dots < i_d \leq n$ be the components onto which we are projecting, with $1 \leq j_1 < j_2 < \dots < j_{n-d} \leq n$ denoting the remaining components. Let $k_{\rightarrow} = \text{proj}_{i_1 \dots i_d}(k)$. Then in terms of Kernels, we have

$$k_{\rightarrow}(x_{i_1}, \dots, x_{i_d}) = \sum_{x_{j_1}, \dots, x_{j_{n-d}}} k(x_1, x_2, \dots, x_n). \quad (2.9)$$

Again, we are summing the weights of k over all values that project to the specified value on the chosen components.

Key Take Aways

A **projection** statistic is a statistic that extracts specified components of a tuple. Mathematically, we write proj_i for the projection statistic that extracts the i th component. We allow multiple components to be listed in the subscript, including a comma-separated list (e.g., $\text{proj}_{1,3,5}$) and a range (e.g., $\text{proj}_{2..4}$).

In `frplib`, projections are constructed using the `Proj` statistic factory, which accepts component indices (from 1), comma-separated lists, and slices in `[]`s.

A transformed FRP (Kind) with a projection statistic is called a *marginal* FRP (Kind) to indicate that it is built by extracting components from a higher-dimensional FRP (Kind). Following the ordinary procedure for transforming Kinds, we get the marginal Kind by adding up the weights for all values of the original Kind that project to the same value. This is expressed in terms of Kernels in equation (2.9).

An FRP can be decomposed into its **scalar components**, the transforms under projection on each individual component.

2.5 Extended Examples

In this section, we develop two extended examples that use statistics in interesting ways: to express and answer questions about the objects produced by a random process and as procedures for estimating the specification of a random system whose specification is unknown. These examples also illustrate how we use FRPs and Kinds to model random systems. Keep an eye out throughout for the patterns in Figure 2.8.

RANDOM GRAPHS. A **graph** is a mathematical structure that describes pairwise relationships among various entities. See Interlude F examples 10.18 and 10.19 for an overview and Interlude G for a detailed discussion. A graph has a set of nodes representing the entities and a set of edges representing the relationships between pairs of entities. Here, we will restrict our attention to *undirected, simple graphs with no loops*. This means that an edge connecting a pair of nodes has no preferred direction (undirected); that there can be at most one edge between any two nodes (simple); and that edges can only connect distinct nodes (no loops). Our nodes here are integers from 1 up to the total number of nodes, and so an edge can be specified by a *set* of two nodes, $\{i, j\}$, indicating that an edge connects nodes i and j in the graph.

We will construct FRPs that represent random generated graphs in this family and use statistics to interrogate and predict properties of these graphs. To load the tools we will need into the playground, enter

```
pgd> from frplib.examples.random_graphs import *
```

at the terminal prompt. The `*` loads all the available tools; you can also list specific functions to import if you prefer. To get details on what is available in the module, enter

```
pgd> import frplib.examples.random_graphs
pgd> help(frplib.examples.random_graphs)
```

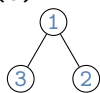
Follow along with the computations as we proceed.

The function `random_graph` is an FRP *factory* that returns an FRP representing a random graph that meets our specification. In particular,²⁶ `random_graph(n, p)` returns a fresh FRP for a random graph on nodes $[1..n]$ for positive integer n and $0 \leq p \leq 1$, with $p = 1/2$ the default if p is not supplied. The parameter p specifies the chance that any particular edge appears in the graph. With $p = 0$, the graph will have no edges; with $p = 1$, it will have every possible edge. A few examples:

²⁶These are called Erdős-Renyi random graphs.

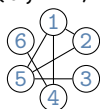
```
pgd> random_graph(3)
```

An FRP with value



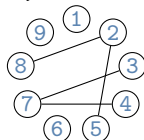
```
pgd> random_graph(6, '1/3')
```

An FRP with value



```
pgd> random_graph(9, '1/9')
```

An FRP with value



When you enter these into the playground, you will not see the pictures for these graphs. Instead, the FRPs returned by `random_graph` have values that represent the graph in a compact way: as tuples of 0s and 1s, where a 1 indicates that there is an edge between a particular pair of nodes.

The meaning of these tuples is illustrated in Figure 2.12 for several small values of n . The dimension of these tuples is $\frac{n(n-1)}{2} = \binom{n}{2}$, as this is the number of possible edges. The edges in a graph with n nodes can be described by an $n \times n$ “adjacency” matrix²⁷ A , where the entry in row i and column j , denoted A_i^j , equals 1 if there is an edge between nodes i and j and 0 otherwise. Because our graphs are undirected, the edge relation is symmetric, so $A_i^j = A_j^i$ for all i and j . Because our graphs have no loops, $A_i^i = 0$ for all i . As a consequence, the entire matrix A is determined by the upper right triangle, strictly above the main diagonal. For instance, in the matrix in the first row (and middle column) of Figure 2.12, the upper right triangle is highlighted, with $A_1^2 = 1$, $A_1^3 = 1$, and $A_2^3 = 0$. Reading these entries left-to-right and row-by-row as a single list gives the tuple $\langle 1, 1, 0 \rangle$, as shown in the right column in the Figure.

²⁷See sections 17.2 and 17.3 in Interlude F for more on matrices.

In the text, we will use the graph pictures wherever possible in lieu of the tuples, for clarity, but you will see the tuples when you play with these in the playground. The `show_graph` function takes a value tuple from a random graph or a random graph FRP and pops up a picture of the graph in a browser tab. The `adjacency_matrix` and `adjacency_list` functions takes a tuple or FRP and show you the edges in a different way, as illustrated below.

We start with $n = 3$ nodes to get a feel for the possibilities.

```
pgd> G_0 = random_graph(3)
```

```
pgd> G_0
```

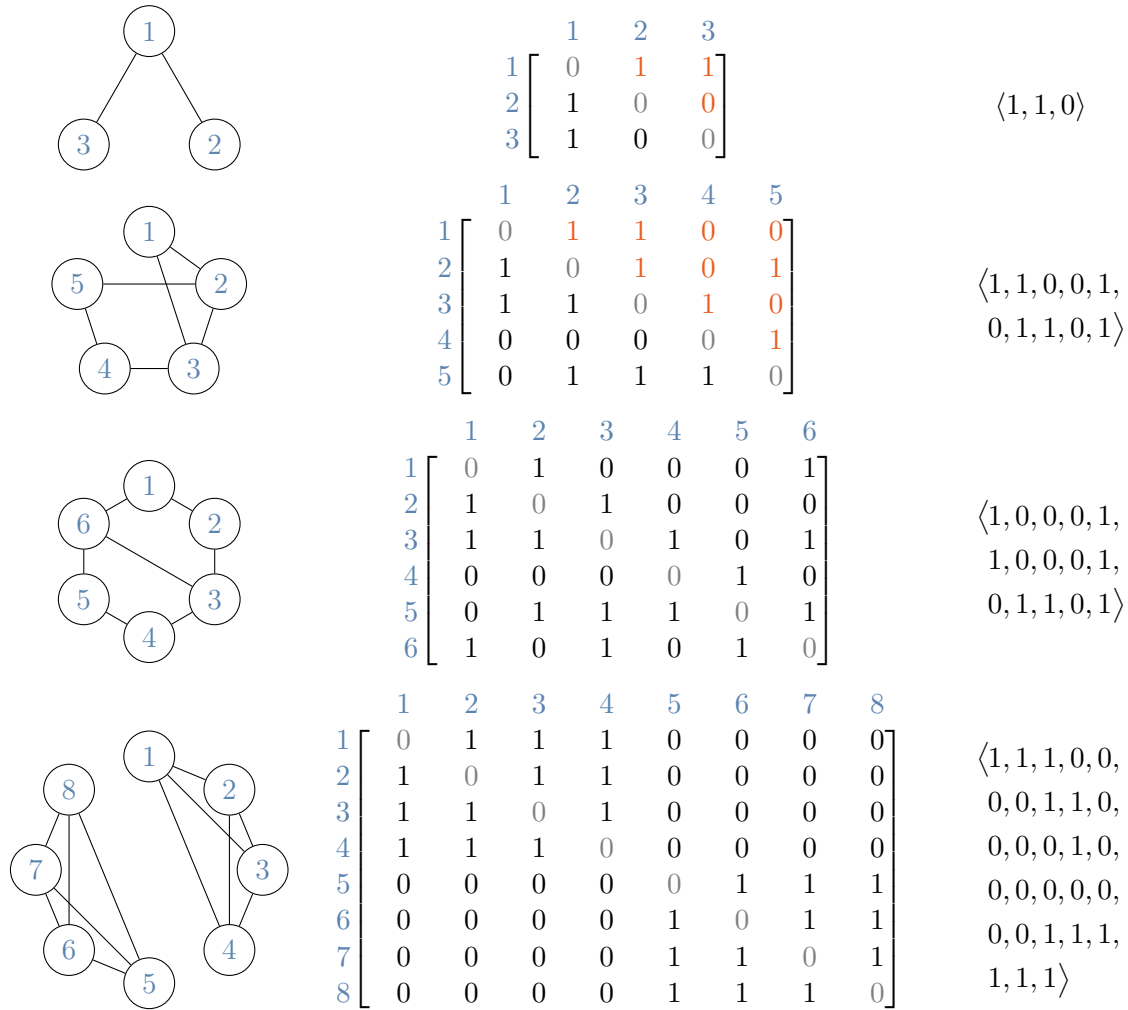


FIGURE 2.12. The representation of undirected simple graphs without loops as numeric tuples. The left column shows the graph. The middle columns gives the graph's "adjacency" matrix that shows which pair of nodes have an edge between them (with a 1 in the corresponding entries). The tuple is laid row-wise into the upper right triangle of the matrix, strictly above the main diagonal, and reflected by symmetry column-wise in the lower left triangle. The main diagonal entries are always zero because there are no loops. The right column gives the corresponding tuple representation, a list of 0s and 1s.

An FRP with value 

```
pgd> show_graph(G_0)
```

An FRP with value <1, 0, 1> # Picture shows in web browser not in terminal

```
pgd> adjacency_matrix(G_0)
```

```
  1 2 3
1 0 1 0
2 1 0 1
3 0 1 0
```

```
pgd> adjacency_list(G_0)
```

```
{1: [2], 2: [1, 3], 3: [2]}
```

For each node, the adjacency list gives the nodes connected to it by an edge. Note that each call to `random_graph` returns a fresh clone, so `G_0` is not the same as the earlier FRP with $n = 3$. Similarly,

```
pgd> G_1 = random_graph(3)
```

```
pgd> G_1
```

An FRP with value 

```
pgd> adjacency_matrix(G_1)
```

```
  1 2 3
1 0 0 0
2 0 0 1
3 0 1 0
```

```
pgd> adjacency_list(G_1)
```

```
{2: [3], 3: [2]}
```

Now we can ask some questions of our random graphs. We are interested in various properties of the graphs produced by these FRPs. For instance: Are two particular nodes connected by an edge? How many edges are associated with each node? How many total edges are in the graph? Can any node be reached from any other by following edges? If not, how many “connected components” does the graph have? Are there cyclical paths? Is the graph a tree? For each such question, we formulate a statistic that answers that question when given a graph on input. Our emphasis is not on answering the questions for a particular graph, but rather on *predicting the answer* for any setting of n and p . In what follows, we will exemplify this for a variety of statistics and graphs, and along the way, we will have to face some of the computational and mathematical challenges in answering our questions.

The function `has_edge` is a *statistic factory* because it *creates a statistic* that tests whether a graph has an edge between two specified nodes. For instance, `has_edge(2, 4)` is a statistic that tests whether there is an edge between nodes 2 and 4. We view the statistic's return value as a Boolean, with 0 for false ($\perp$) and 1 for true ($\top$). A Boolean statistic is called a *condition* because it indicates when a particular condition is met. We can check whether `G_0` and `G_1` have various edges:

```
pgd> G_0 ~ has_edge(1,3)
<0>
pgd> G_1 ~ has_edge(2,3)
<1>
pgd> has_edge(1,2)(G_0)
<1>
pgd> has_edge(1,2)(G_1)
<0>
pgd> edge13 = has_edge(1,3)
pgd> edge13(G_1)
<0>
```

Here, we see three equivalent ways of using the statistic returned from the factory to transform an FRP: using the `~` operator, applying the statistic directly, and naming the statistic to apply it later by name. The `~` operator makes the meaning of the operation clearer in this case, so it is particularly useful when transforming with statistic factories.

An FRP with (Boolean) values 0 and 1 – as produced by transforming another FRP with a condition – is called an **event**. If the FRP has value 1, the event is said to have *occurred*, otherwise not.

The statistics `is_connected`, `is_acyclic`, and `is_tree` are conditions that test, respectively, (i) if every node can be reached from every other node by following edges, (ii) if the graph has no cycles,²⁸ and (iii) if the graph is a *tree* – a connected, acyclic graph.

```
pgd> is_connected(G_0)
An FRP with value <1>
pgd> is_connected(G_1)
An FRP with value <0>
pgd> is_acyclic(G_0)
```

²⁸A cycle exists if there is a non-trivial path along distinct edges from some node to itself.

An FRP with value <1>

```
pgd> is_tree(G_1)
```

An FRP with value <0>

The statistic `connected_components` has type $\binom{n}{2} \rightarrow n$. It maps a graph's tuple representation to a tuple with one integer component per node, and two nodes have the same integer if they are connected by a path in the graph. For example:

```
pgd> connected_components(G_0)
```

An FRP with value <1, 1, 1>

```
pgd> connected_components(G_1)
```

An FRP with value <1, 2, 2>

We will see other graph statistics below.

The Kinds for `random_graph(3,p)` are shown in Figure 2.13 for $p = 1/2$, $p = 1/3$, and $p = 3/5$. These have dimension 3 and size $8 = 2^3$. If you compute these Kinds in the playground, it will show weights with the same relative size but normalized so that they sum to 1. As we saw in Section 1.3, Kinds whose weights differ by a constant scaling factor will produce the same demos and so are in practice equivalent.

When $p > 1/2$, the weights increase with the number of edges in the graph, as in the Kind on the right of Figure 2.13. When $p < 1/2$, the weights decrease with the number of edges, as in the Kind on the middle of Figure 2.13. To understand this, let us look at the Kind of the event that a particular edge (say $\{2,3\}$) is in the graph. We can compute the Kind of the random graph FRP transformed by `has_edge(2,3)` or transform the Kind of the random graph FRP by `has_edge(2,3)`. By equation (2.1), both give the same result.

```
pgd> kind(random_graph(3,1/2) ^ has_edge(2,3))
```

```
,---- 1/2 ---- 0
```

```
<> -|
```

```
`---- 1/2 ---- 1
```

```
pgd> kind(random_graph(3,1/3)) ^ has_edge(2,3)
```

```
,---- 2/3 ---- 0
```

```
<> -|
```

```
`---- 1/3 ---- 1
```

```
pgd> kind(random_graph(3,3/5)) ^ has_edge(2,3)
```

```
,---- 2/5 ---- 0
```

```
<> -|
```

```
`---- 3/5 ---- 1
```

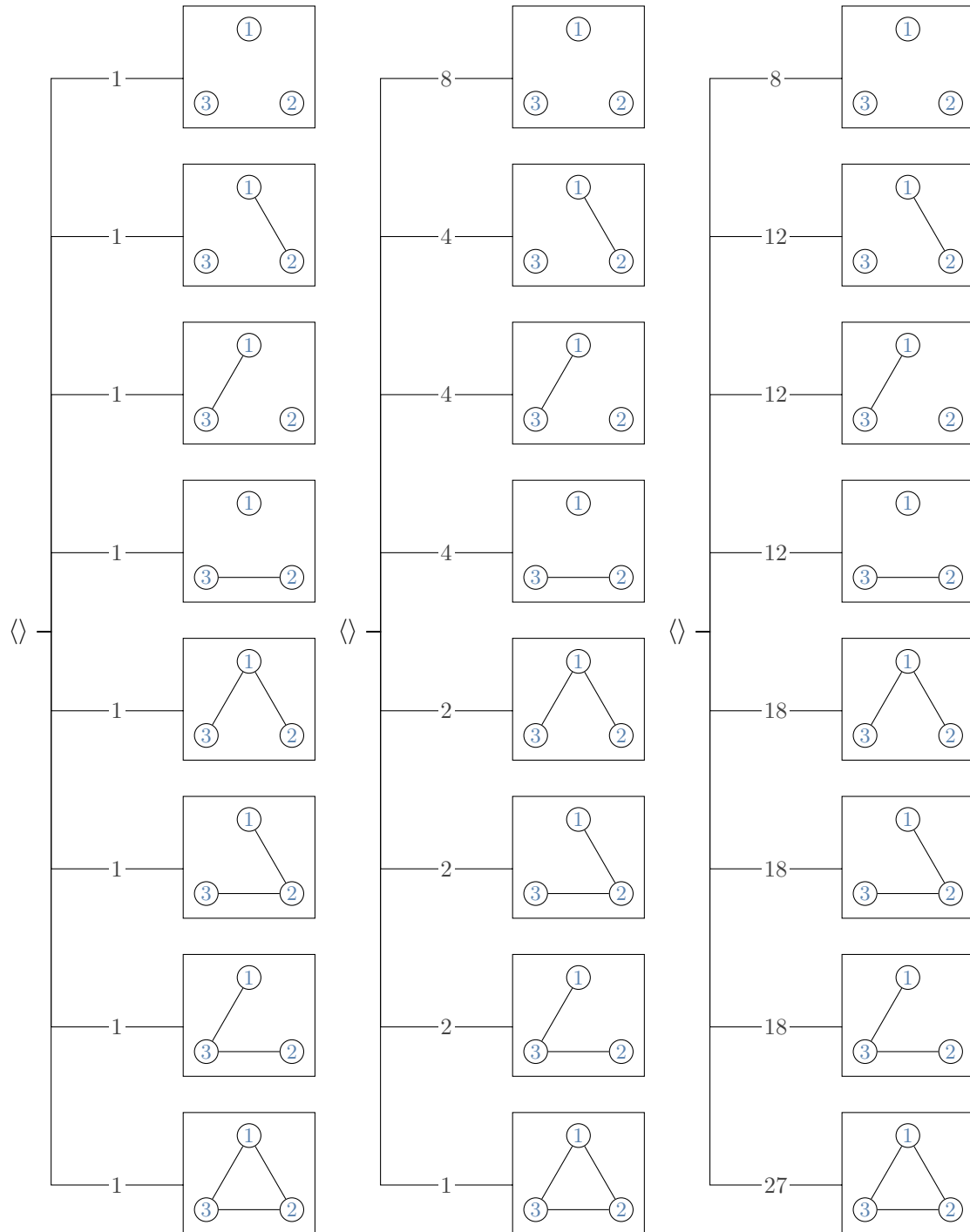


FIGURE 2.13. The Kinds of $\text{random\_graph}(3)$ (left), $\text{random\_graph}(3, 1/3)$ (center), and $\text{random\_graph}(3, 3/5)$ (right). The Kinds are not in canonical form.

Recall the steps for transforming a Kind: apply the statistic to the value at every leaf node then combine branches that map to a common value by adding the weights. Referring to Figure 2.13, we see that the first three and the fifth branches have values without a $\{2,3\}$ edge, and the others have values with a $\{2,3\}$ edge. Summing the weights for these branches: when $p = 1/2$, we get 4 without the edge and 4 with; when $p = 1/3$, we get 18 without and 9 with; when $p = 3/5$, we get 50 without and 75 with. This gives respective Kinds

$$\langle \rangle \begin{cases} 4 & \langle 0 \rangle \\ 4 & \langle 1 \rangle \end{cases} \quad \langle \rangle \begin{cases} 18 & \langle 0 \rangle \\ 9 & \langle 1 \rangle \end{cases} \quad \langle \rangle \begin{cases} 50 & \langle 0 \rangle \\ 75 & \langle 1 \rangle \end{cases}$$

where the weights on 0 and 1 have ratios 1 to 1, 2 to 1, and 2 to 3. Renormalizing the weights to sum to 1, we get the same Kind the playground computes.

It seems that for these `random_graph(3,p)` Kinds, the ratio of the weight on $\langle 1 \rangle$ to the weight on $\langle 0 \rangle$ is $p/(1-p)$. Let's think about what this means in terms of the demos we ran in Section 1.3. If we obtain a large number of `random_graph(3,p)` FRPs, check if they have a $\{2,3\}$ edge (i.e., transform them with `has_edge(2,3)`), and tabulate the results, we will see approximately *a proportion p of them have such an edge* and a proportion $1-p$ will not.

Puzzle 32. Do this test yourself for a variety of $0 \leq p \leq 1$. Use

```
FRP.sample(1_000_000, random_graph(3,p) ~ has_edge(2,3))
```

to compute a demo for each p you choose.

Are the results consistent with the Kind? Do the results depend on which edge you choose?

The vast, apparently infinite number of FRPs stored at the Marketplace are not arranged in any systematic way; when you obtain one from the market, it is an arbitrary clone of the FRPs of its Kind. The demo tells us that if we take all the FRPs whose Kind is `kind(random_graph(3,p))`, activate them, and apply `has_edge(2,3)` to their value, the *average* of the results will be p . So p is the proportion of `random_graph(3,p)` FRPs for which the event that it has an edge $\{2,3\}$ occurs. In that sense, p quantifies the chance that you receive an FRP that represents a graph with such an edge. This is one way to think about the idea of *probability*: the probability of an event is the average value of the condition, a proportion, over all FRPs in the Marketplace of the same Kind.

An important observation for the future:

```
pgd> E(random_graph(3,1/2) ~ has_edge(2,3))
1/2
pgd> E(random_graph(3,1/3) ~ has_edge(2,3))
1/3
pgd> E(random_graph(3,3/5) ~ has_edge(2,3))
3/5
```

The expectation – the risk-neutral price, our prediction of the FRP’s value – is exactly that average! The probability that an event occurs is just our best prediction of whether its defining condition is true.

Puzzle 33. In the playground, compute the transformed Kind of `random_graph(3,p)` by the statistic `is_connected`, for p equal to $1/2$, $1/3$, and $3/5$. What is the probability that you obtain a connected graph in these three cases?

Answers: $5/8, 1/2, 7/12$

We can ask other questions as well and define statistics to represent them. How many edges does the graph have? (`edge_count`) How many connected components does the graph have? (`connected_component_count`) What are they? (`connected_components`) How many neighbors does each node have? (`degrees`) Is the graph free of cycles? (`is_acyclic`) It is useful to compute these first for random graphs with 3 nodes because the Kinds have small enough size that we can easily see how the transformation steps are working.

```
pgd> kind(random_graph(3,1/2)) ~ edge_count
,---- 1/8 ---- 0
|---- 3/8 ---- 1
<> -|
|---- 3/8 ---- 2
`---- 1/8 ---- 3
pgd> kind(random_graph(3,1/3)) ~ edge_count
,---- 8/27 ----- 0
|---- 12/27 ----- 1
<> -|
|---- 6/27 ----- 2
`---- 1/27 ----- 3
```

```
pgd> kind(random_graph(3,3/5)) ^ edge_count
,---- 8/125 ---- 0
|---- 36/125 --- 1
<> -|
|---- 54/125 --- 2
`---- 27/125 --- 3
```

Calculate these Kinds by hand from Figure 2.13 and compare your results.

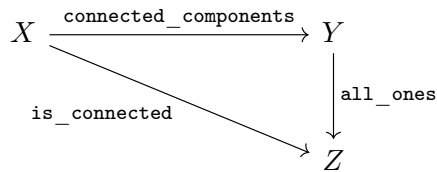
```
pgd> K_G3h = kind(random_graph(3))
pgd> connected_components(K_G3h)
,---- 4/8 ---- <1, 1, 1>
|---- 1/8 ---- <1, 1, 2>
<> -+---- 1/8 ---- <1, 2, 1>
|---- 1/8 ---- <1, 2, 2>
`---- 1/8 ---- <1, 2, 3>
```

This Kind describes the connected components of `random_graph(3)`'s; the values give a label for each node (with labels starting from 1) where two nodes with the same label belong to the same connected component. Observe that from this information we can also answer other questions. Is the graph connected? It is if all components' labels are equal to 1. How many connected components are there? It is the largest component label. What is the *pattern* of component sizes? List the sizes of the components in decreasing order. We can answer these questions in two ways, either directly from the Kind `K_G3h` or by further transforming `connected_components(K_G3h)`. For instance,

```
pgd> all_ones = condition(lambda v: all(vi == 1 for vi in v))
pgd> K_G3h ^ connected_components ^ all_ones
,---- 1/2 ---- 0
<> -|
`---- 1/2 ---- 1
pgd> is_connected(K_G3h)
,---- 1/2 ---- 0
<> -|
`---- 1/2 ---- 1
```

The statistic `all_ones` is a condition that returns true when all components of its input equal 1. (The `lambda` syntax in Python defines *anonymous functions*, which are

discussed in Chapter 11 of Interlude F. The term `lambda args: expr` is a function taking `args` as arguments and returning the value of expression `expr`, which may involve the given arguments.) Here, we use the ability of the `^` operator to *chain* several statistics together,²⁹ transforming first by `connected_components` then by `all_ones`. The statistic `is_connected` is *equal* to the statistic `connected_components ^ all_ones` that first applies `connected_components` and then applies `all_ones` to the returned result. We can visualize these relationships in a diagram



where X , Y , and Z all stand for Kinds or for FRPs and the arrows represent transformations by statistics. In this diagram, any two distinct paths from one node to another represent equal functions, so we say this diagram *commutes*.³⁰

Similarly, using the built-in statistic `Max` to compute the maximum label:

```

pgd> K_G3h ^ connected_components ^ Max
      ,---- 4/8 ---- 1
<> -+----- 3/8 ---- 2
      `---- 1/8 ---- 3
pgd> connected_component_count(K_G3h)
      ,---- 4/8 ---- 1
<> -+----- 3/8 ---- 2
      `---- 1/8 ---- 3
pgd> K_G3h ^ connected_components ^ connected_component_sizes
      ,---- 1/8 ---- <1, 1, 1>
<> -+----- 3/8 ---- <2, 1, 0>
      `---- 4/8 ---- <3, 0, 0>

```

where the statistic `connected_component_sizes` takes a tuple of component labels and returns the components' sizes in descending order. The 0's are padded out to n entries to ensure that our statistic has the required dimension. Thus the value `<2, 1, 0>` occurs in the three labelings of components where there is one component of size 2 and one component of size 1. It is worth deriving these transformed Kinds by hand where the sizes are small enough to understand the operation clearly.

Another common question we might ask about a random graph is how many neighbors each node has. We can use the statistic `degrees` to answer this question.

²⁹See Chapter 13 in Interlude F for a detailed discussion of function composition.

³⁰See Section 13.2.

```
pgd> degrees(K_G3h)
,---- 1/8 ---- <0, 0, 0>
|---- 1/8 ---- <0, 1, 1>
|---- 1/8 ---- <1, 0, 1>
|---- 1/8 ---- <1, 1, 0>
<> -|
|---- 1/8 ---- <1, 1, 2>
|---- 1/8 ---- <1, 2, 1>
|---- 1/8 ---- <2, 1, 1>
`---- 1/8 ---- <2, 2, 2>
```

But more typically we are less interested in what happens at particular nodes than at understanding the *pattern* in the numbers of neighbors globally over the graph. For this we can transform the previous Kind by the statistic **Ascending** that sorts the tuple in increasing order:³¹

³¹There is also an analogous Descending statistic.

```
pgd> K_G3h ^ degrees ^ Ascending
,---- 1/8 ---- <0, 0, 0>
|---- 3/8 ---- <0, 1, 1>
<> -|
|---- 3/8 ---- <1, 1, 2>
`---- 1/8 ---- <2, 2, 2>
```

This elides the distinction between degree profiles that differ only by labeling of the nodes. Observe how such profiles are combined into a single pattern and their weights added by the Kind transformation.

Puzzle 34. Explain the following computations with reference to Figure 2.13.

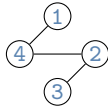
```
pgd> is_acyclic(K_G3h)
,---- 1/8 ---- 0
<> -|
`---- 7/8 ---- 1
pgd> kind(random_graph(3, 3/5)) ^ is_acyclic
,---- 27/125 ---- 0
<> -|
`---- 98/125 ---- 1
```

For $n = 4$ nodes, `random_graph(4,p)` has dimension 6 and size $64 = 2^6$.

```
pgd> G = random_graph(4)
```

```
pgd> G
```

An FRP with value



Aside. We might ask another type of question about the presence of edges: if we have a `random_graph(3,p)` FRP X and we have observed that the value of the transformed FRP $X \sim \text{has\_edge}(1,2)$ is 1 – that is, we *know* that X ’s value has an edge $\{1,2\}$ – what can we say about whether X has an edge $\{2,3\}$? To incorporate our partial knowledge of X ’s value, we use an *observational constraint* as discussed in detail in Chapter 5. For instance,

```
pgd> X = random_graph(3, '1/3')
pgd> kind(X | has_edge(1,2)) ^ has_edge(2,3)
      ,---- 2/3 ---- 0
<> -|
      `---- 1/3 ---- 1
```

The `|` is read as “given”; it takes an FRP or Kind on the left and a condition on the right and applies the constraint that the condition is true. We see here that the partial information about edge $\{1,2\}$ does not change the Kind of $X \sim \text{has\_edge}(2,3)$. We will explore this more later.

As will become clearer in Chapter 4, for each p , the events

$$\text{random\_graph}(3,p) \sim \text{has\_edge}(i,j)$$

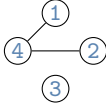
over all edges $\{i,j\}$ are the *Element FRPs* of Figure 2.8 from which `random_graph(3,p)` is built. For any p you choose, such as $3/5$, try

```
pgd> edge_kind = weighted_as(0, 1, weights=[1 - p, p])
pgd> K_p = edge_kind * edge_kind * edge_kind
pgd> Kind.equal(K_p, kind(random_graph(3,p)))
True
pgd> frp(edge_kind) * frp(edge_kind) * frp(edge_kind)
An FRP with value <1, 0, 1>
```

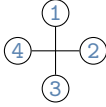
Here, the function `weighted_as` is a *Kind factory*; look at `edge_kind` and see that it is Kind of the `has_edge(i,j)` transformed FRP that we saw above. The independent join (with operator `*`) builds a three-dimensional Kind that equals our random graph Kind. The last line builds the corresponding FRP.

You might want to return to this after reading Chapter 5.

```
pgd> clone(G)
```

An FRP with value 

```
pgd> clone(G)
```

An FRP with value 

Look at `kind(random_graph(4))`, `kind(random_graph(4, 1/3))`, and `kind(random_graph(4, 3/5))` in the playground.

We can now ask some questions about random graphs with 4 nodes.

```
pgd> K_G = kind(G)      # equal to kind(random_graph(4))
```

How many edges does the graph have?

```
pgd> edge_count(K_G)
,---- 1/16 ---- 0
|---- 4/16 ---- 1
<> -+---- 6/16 ---- 2
    |---- 4/16 ---- 3
    `---- 1/16 ---- 4
```

Try answering this same question with `random_graph(4, 1/3)` and `random_graph(4, 3/5)` and compare the resulting Kinds.

How many connected components does the graph have?

```
pgd> connected_component_count(K_G)
,---- 38/64 ----- 1
|---- 19/64 ----- 2
<> -|
    |---- 6/64 ----- 3
    `---- 1/64 ----- 4
```

Notice how the weights shift towards fewer components when p is larger and towards more components when p is smaller.

```
pgd> kind(random_graph(4, '3/5')) ^ connected_component_count
,---- 0.76550 ----- 1
|---- 0.19354 ----- 2
<> -|
```



```

      |---- 0.036864 ---- 3
      `---- 0.004096 ---- 4
pgd> kind(random_graph(4, '1/3')) ^ connected_component_count
      ,---- 0.27572 ----- 1
      |---- 0.37311 ----- 2
<> -|
      |---- 0.26337 ----- 3
      `---- 0.087791 ---- 4
pgd> kind(random_graph(4, '1/20')) ^ connected_component_count
      ,---- 0.0018012 ---- 1
      |---- 0.030973 ----- 2
<> -|
      |---- 0.23213 ----- 3
      `---- 0.73509 ----- 4

```

Is the graph a tree? Is it connected? Is it free of cycles?

```

pgd> is_tree(G)
An FRP with value <1>
pgd> is_tree(K_G)
      ,---- 11/16 ---- 0
<> -|
      `---- 5/16 ----- 1
pgd> is_connected(G)
An FRP with value <1>
pgd> is_connected(K_G)
      ,---- 13/32 ---- 0
<> -|
      `---- 19/32 ---- 1
pgd> is_acyclic(G)
An FRP with value <1>
pgd> is_acyclic(K_G)
      ,---- 13/32 ---- 0
<> -|
      `---- 19/32 ---- 1

```

For small p , we saw that the graph is likely to have many connected components and

```
pgd> kind(random_graph(4, '1/20')) ^ is_acyclic
```

```
,---- 0.00051509 ---- 0
<> -|
`---- 0.99948 ----- 1
```

suggesting that the graph is very likely to be a *forest*³² of small trees.

³²A *forest* is a collection of trees.

Does the graph have edges $\{1,2\}$ and $\{3,4\}$? For this question, we create a statistic using a *combinator*: a function that takes two or more statistics and returns a new one. In this case we use the `And` combinator.

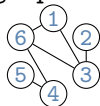
```
pgd> has_12_34 = And(has_edge(1,2), has_edge(3,4))
pgd> has_12_34(G)
An FRP with value 0
pgd> has_12_34(K_G)
,---- 3/4 ---- 0
<> -|
`---- 1/4 ---- 1
```

In all these cases, transforming the FRP with the statistic gives us an answer to the corresponding question *for that graph*, and transforming the Kind tells us how the answer varies over all FRPs of the original Kind. The transformed Kind lets us *predict* the answer to the question.

Generating random graphs is a relatively fast operation, so we can generate random graphs of substantial size and interrogate their properties.

```
pgd> G6 = random_graph(6)
```

An FRP with value



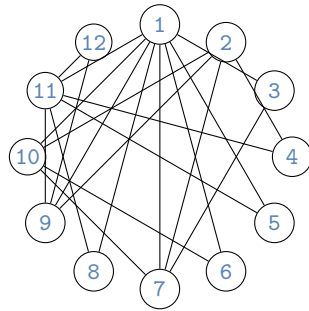
```
pgd> G6 ^ Fork(is_connected, is_acyclic, is_tree)
```

An FRP with value <1, 0, 0>

Here, `Fork` is a statistic combinator that applies the listed statistics in parallel to the same values and concatenates the results in order into one tuple. So we see that `G6` is connected because the first component of the tuple is 1 and that it has cycles and is not a tree because the last two components are 0.

```
pgd> G12 = random_graph(12, '1/3')
```

An FRP with value



```
pgd> is_tree(G12)
```

An FRP with value <0>

```
pgd> connected_components(G12)
```

An FRP with value <1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1>.

(It may be slow to evaluate its kind.)

```
pgd> G12 ^ connected_components ^ connected_component_sizes
```

An FRP with value <12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0>.

(It may be slow to evaluate its kind.)

```
pgd> G12 ^ degrees ^ ascending
```

An FRP with value <2, 2, 2, 2, 2, 2, 4, 4, 4, 4, 6, 8>.

(It may be slow to evaluate its kind.)

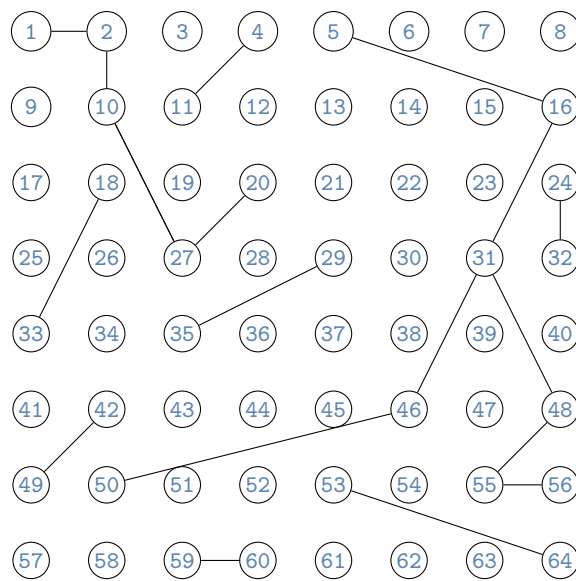
```
pgd> G12 ^ Fork(edge_count, degrees ^ Sum / 2)
```

An FRP with value <21, 21>.

Even with only 21 out of 66 possible edges, G12 is connected.

```
pgd> G64 = random_graph(64, '1/128')
```

An FRP with value



```
pgd> is_forest(G64)
An FRP with value <1>
```

The transformed FRPs here describe the properties of the graphs represented by **G6** and **G12**. Though we can generate FRPs easily, the playground is warning us that computing the Kind of **G12**, which has size 2^{66} , might be ... time consuming. Because p is small for **G64**, most of the nodes of this graph are isolated, leaving a forest of small trees. The condition `is_forest` tests whether a graph's connected components are all trees. It is defined using the function `ForEachComponent`, from `frplib.examples.random_graphs`, that takes a graph statistic (here `is_tree`) and applies it to the subgraph of each connected component.

As discussed earlier, our main goal is not just to interrogate the properties of particular graphs but to predict the properties of the random graphs before we activate the FRPs. Our basic approach to doing this is to compute the transformed *Kinds* with the statistics that represent our questions. When $n = 6$, the Kind of `random_graph(n,p)` has size $2^{15} = 32,768$. We can compute its Kind directly, but we can see that the size is growing fast. Indeed, for `random_graph(8,p)` the Kind has size $2^{28} = 268,435,456$, which is within the range of computational feasibility to compute but rather inconvenient. For general n , we get $2^{n(n-1)/2}$ – exponential growth! Directly computing the Kinds is not always computationally feasible, so we need to bring other strategies to bear.

First, if we can compute directly with reasonable efficiency, we do, and for many problems this is enough. When that gets difficult – such as when the Kind of the Data FRP has a large size – we can focus our attention on computing the Kinds of the feature FRPs (transforms of the Data FRP relevant to our questions, cf. Figure 2.8), which usually have much smaller size and dimension. We can try to devise computational methods and algorithms that allow us to efficiently compute answers to our questions. And where possible and necessary, we apply *mathematical reasoning* to derive those answers, or approximate those answers to specified accuracy if an exact answer is elusive, or if necessary, put useful bounds on the answers. Ideally, we could answer our questions in full generality, but often we face a trade-off between the precision of our answers and the specificity of the assumptions we need to make. As such we often tailor our reasoning and analysis to specific questions or special cases that are amenable to analysis. The combination and interaction of computational and mathematical reasoning offers us a diverse and powerful range of tools for answering probabilistic questions. We will see this theme reprised many times as we proceed.

As an example, suppose we want to predict the number of edges in a graph like **G64**.

The Kind of the FRP `G64` has size 2^{2015} , so direct computation is quite out of reach. We want the Kind of the transformed FRP, `kind(G64) ~ edge_count`, but computing it *that way* is consequently infeasible. However, using a computational technique we will describe in Section 7.1, we can find the Kind quickly to use in practice. The function `fast_edge_count` in `frplib.examples.random_graphs` computes the Kind of `random_graph(n,p) ~ edge_count` for a specified n and p . For instance:

```
pgd> E64 = fast_edge_count(64, '1/128')
```

```
pgd> is_kind(E64)
```

```
True
```

```
pgd> clean(E64, '1e-6')
```

```
,---- 1.3583E-7 ----- 0
|---- 0.0000021562 ---- 1
|---- 0.000017105 ----- 2
|---- 0.000090420 ----- 3
|---- 0.00035830 ----- 4
|---- 0.0011353 ----- 5
|---- 0.0029961 ----- 6
|---- 0.0067741 ----- 7
|---- 0.013395 ----- 8
|---- 0.023532 ----- 9
|---- 0.037188 ----- 10
|---- 0.053399 ----- 11
|---- 0.070253 ----- 12
|---- 0.085273 ----- 13
|---- 0.096064 ----- 14
|---- 0.10096 ----- 15
|---- 0.099416 ----- 16
|---- 0.092094 ----- 17
|---- 0.080532 ----- 18
|---- 0.066682 ----- 19
<> -+---- 0.052427 ----- 20
|---- 0.039236 ----- 21
|---- 0.028016 ----- 22
|---- 0.019125 ----- 23
|---- 0.012505 ----- 24
|---- 0.0078458 ----- 25
```

```

|---- 0.0047308 ----- 26
|---- 0.0027455 ----- 27
|---- 0.0015356 ----- 28
|---- 0.00082890 ----- 29
|---- 0.00043229 ----- 30
|---- 0.00021807 ----- 31
|---- 0.00010651 ----- 32
|---- 0.000050422 ----- 33
|---- 0.000023156 ----- 34
|---- 0.000010325 ----- 35
|---- 0.0000044738 ---- 36
`---- 0.0000018851 ---- 37

```

The `clean` function here trims off branches with weights less than the given threshold 10^{-6} to make it easier to view and interpret the Kind. (You can just view E64 to compare.) We see that the largest weights are in the range 12 to 19, and

```
pgd> E(E64)
63/4
```

is our best prediction of the number of edges.

With some mathematical reasoning that we will derive later, we can go further and derive the *exact* Kind for `edge_count(random_graph(n, p))` for any n and p . Try `exact_edge_count(n,p)` to see this Kind, e.g., compare `exact_edge_count(64, '1/128')`. For very large n and very small p , we can also get an excellent approximation to `exact_edge_count(n,p)` that is quite fast.

Another approach we can use get results for large Kinds is *sampling*. We can compute transformed FRPs fairly easily, so by running demos of the transformed FRP, we get an empirical approximation of the Kind. As an example, compare the following demo with the exact Kind:

```
pgd> FRP.sample(1_000_000, is_connected(random_graph(6)))
```

Summary of Output Values

```

+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| 0      | 185191 | 18.52% |
| 1      | 814809 | 81.48% |
+-----+-----+-----+

```

```
pgd> kind(random_graph(6)) ~ is_connected
      ,---- 0.18506 ---- 0
    <> -|
      `---- 0.81494 ---- 1
```

Sampling gives us a numeric approximation of the Kind. The sample will always be somewhat inaccurate, but we can quantify that inaccuracy enough to make it useful.

We can answer the question of whether a graph is a tree even for very large n in the special case where $p = 1/2$. The function `exact_is_tree` computes that Kind for each n .

RANDOM IMAGES. Our next example considers FRPs that generate random binary images. Such an image is a rectangular grid of “pixels” whose value can be either 0 or 1. We will focus on 32×32 images, though the logic of the example works with any size. In the playground, load the tools needed for this example with

```
pgd> from frplib.examples.random_images import (
...>     random_image, as_image, add_images, show_image,
...>     image_distance, closest_image_to,
...>     erode, dilate, max_likelihood_image,
...>     ImageModels, pixel0, pixel1
...> )
```

or if you prefer to avoid typing, just do

```
pgd> from frplib.examples.random_images import *
```

As with the random graphs example, we will show the values of our FRPs as images, but in the playground you will see tuples, as described below. You can use the `show_image` function on a tuple or image FRP to pop up a view of the image in a browser window.

The function `random_image` is an FRP factory that returns an FRP representing a random binary image. (The default is a 32×32 image.)

```
pgd> random_image()
```

An FRP with value



```
pgd> random_image(p='1/8')
```

```

An FRP with value 
pgd> random_image(p='1/6', base=ImageModels.image('P'))
An FRP with value 
pgd> random_image(p='1/8', base=ImageModels.image('minus'))
An FRP with value 
pgd> random_image(p='1/8', base=ImageModels.image('E'))
An FRP with value 
pgd> random_image(p='1/8', base=ImageModels.image('blocks'))
An FRP with value 
pgd> my_image = as_image(pixel0 * 416 + pixel1 * 64 + pixel0 * 64 +
...>      pixel1 * 64 + pixel0 * 416)
pgd> random_image(p='1/4', base=my_image)
An FRP with value 
pgd> ImageModels.register_image('my_image', my_image) # save image

```

These FRPs represent binary images with a random scattering of black pixels superimposed on a base image. Black pixels have a value 1 and white pixels a value 0. We can think of the images as pixelwise sum of a base image and random “noise”, where pixels add with an exclusive-or: $1 \oplus 0 = 1 = 0 \oplus 1$ and $0 \oplus 0 = 0 = 1 \oplus 1$. (Note the last equality.) The `base` argument to `random_image` sets the base image (by default all white). You can use pre-defined images in the `ImageModels` object, define your own with `as_image`, or add custom images to `ImageModels`. The `p` argument to `random_image` determines the intensity of the noise, with larger p making noisy pixels more prevalent.³³ For `my_image` above, we specify the pixel values left-to-right then top-down: 416 0’s followed by 64 1s then 64 0s then 64 1s and 416 0s.

These FRPs’ values are tuples of dimension 2+1024, with the first two components the image width and height followed by 1024 0s and 1s arranged row-wise from the top left to the bottom right of the image. The Kinds of these FRPs have size 2^{1024} , so we will not be computing them directly. But we can use mathematical and computational

³³Remember that you can use `help` in the playground to get full documentation on all these functions.

reasoning to operate on the Kinds that we need. (As all the images here have Kinds of large size, I will elide the “It may be slow to evaluate its kind” warnings in playground output in this example.)

The example code defines a variety of statistics that operate on images.

```
pgd> black_pixels(random_image())    # Number of black pixels in image
An FRP with value <489>
pgd> clockwise(ImageModels.image('F'))
```



```
pgd> noisyF = random_image(p='2/15', base=ImageModels.image('F'))
```



An FRP with value

```
pgd> reflect_image_vertically(noisyF)
```



An FRP with value

```
pgd> largest_cluster_size(random_image(p='1/3'))
```

An FRP with value <98>

where `ImageModels.image('F')` returns a pre-defined image that happens to look like an F.

Our goal in the remainder of this example is to develop a way to *reconstruct* an underlying image from a noisy observed image. We will build an FRP that represents a random image with an *unknown* base image, but we *assume* that the base image belongs to a known collection of possible base images that we call a *model*. We will use statistics here as procedures for reconstructing the unknown base image. These statistics have type $2 + \text{width} * \text{height} \rightarrow 2 + \text{width} * \text{height}$ (e.g., $1026 \rightarrow 1026$) and map an observed, noisy image to an image that is an estimated reconstruction of the unknown base image. Our random image FRP represents the data, and the FRP transformed by these statistics represents our estimated “reconstruction” of the unknown base image. We want to produce a reconstruction that is close on average to the true base image.

The `ImageModels` object has various methods for working with random image models. A model is specified by a set of *candidate* base images and a parameter $0 \leq p \leq 1$ that determines noise prevalence. When we observe an image generated under a particular model, one of the candidate base images is selected and noise with the specified prevalence is superimposed. The function `ImageModels.observe`

returns an FRP representing a random image with a specified model. Several sets of candidate base images have been pre-defined with short identifiers. You can list the identifiers with `ImageModels.models()` and list a model's base images with `ImageModels.model(id)`. `ImageModels.observe` accepts a model identifier or a list of images in the first argument. It uses $p = 1/8$ by default, but you can override this with the named `p` argument. You can also register new sets of base images with `ImageModels.register_model`.

```
pgd> ImageModels.model('efh')
```



```
pgd> data, truth = ImageModels.observe('efh', p='1/4')
```

```
pgd> data
```

An FRP with value 

We first see the three base images that might underlie our data, but we do not know which one it is. `ImageModels.observe` returns both an FRP (`data`) representing the observed data – a random image that superimposes noisy pixels on an unknown base image from the models – and the true base image (`truth`) on which the data is based. In practice, `truth` would be *unknown*, but for our purposes here, it is convenient to have access to it so that we can evaluate our procedure. Our goal is to devise a statistic that transforms the FRP `data` to an FRP whose value is an image as close as possible to the unknown `truth`.

From the image shown here, you can probably guess the unknown `truth`, but that's ok. The methods described here apply even when it hard to distinguish between the model images. Using an easier problem like this will clarify the ideas.

Puzzle 35. Create two or more images with `as_image` and use them to register a new model in `ImageModels`. Generate data from this model and look at the images.

Our first reconstruction statistic will be an algorithm to “de-noise” the image. It operates on the image by eliminating pixels that look like noise but retaining those that do not. The idea is that the de-noising statistic will preferentially remove the scattered blocks from noise while retaining significant structure from the base image. This procedure neither accounts for nor requires knowledge of the possible base images. To define our statistic, we need to define two operations on images:

image *dilation* and image *erosion*. For both these functions, we think of an image's pixels as lying on the grid of points in the plane with integer coordinates. If $\mathcal{I}$ is a binary image, we can think of it as a set of integer coordinates at which there is a black/1 pixel. The dilation of an image $\mathcal{I}$ with respect to a *dilation element* $\mathcal{D}$ – a small set of integer coordinates – has black/1 pixels at coordinates³⁴

$$\text{dilate}_{\mathcal{D}}(\mathcal{I}) = \{ \langle x+i, y+j \rangle \mid \langle x, y \rangle \in \mathcal{I} \wedge \langle i, j \rangle \in \mathcal{D} \}. \quad (2.10)$$

Put another way, we overlay a copy of the dilation element at every black pixel in the image. For example, using the default dilation element $\mathcal{D}_0 = \{ \langle i, j \rangle \mid i, j \in \{-1, 0, 1\} \}$, a small square centered at 0, we have



where the dilated image is 34×34 . Dilation spreads out every black pixel in an image with a copy of the dilation element, so the dilated image is bigger by two pixels in each dimension.

Erosion is the dual operation to dilation. We specify an *erosion element* $\mathcal{E}$ – again, a small set of integer coordinates – and the resulting image has black/1 pixels at coordinates

$$\text{erode}_{\mathcal{E}}(\mathcal{I}) = \{ \langle x, y \rangle \mid \langle x+i, y+j \rangle \in \mathcal{I} \text{ for all } \langle i, j \rangle \in \mathcal{E} \}. \quad (2.11)$$

We include all the pixels at which a copy of the erosion element is completely contained in the image. For example, using the default erosion element $\mathcal{E}_0 = \{ \langle i, j \rangle \mid i, j \in \{-1, 0, 1\} \}$, a small square centered at 0, we have

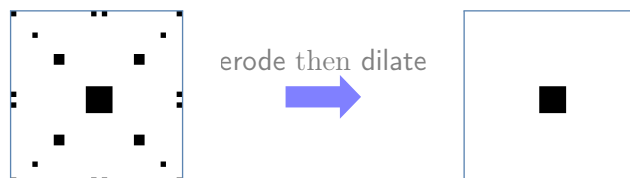


where the eroded image is 30×30 , two pixels smaller than the original image in each dimension. Try eroding and dilating a few of the pre-defined images to get a feel for what these do.

³⁴In defining a set, the $\mid$ is read “such that” or “given” and the $\wedge$ is logical-and. The condition after the $\mid$ constrains the elements of the set. See Chapter 9.

Our statistic for reconstructing the base image from the data is to first apply `erode` (with the default erosion element) and then to apply `dilate` (with the default dilation element). In the playground, `erode` and `dilate` are statistic factories that take the erosion and dilation elements³⁵ and return the corresponding statistics. So, using the default elements, we write our “de-noising” statistic as `denoise = erode() ^ dilate()`. First we erode and then we dilate.

³⁵Both accept instead an integer s , indicating a square element with each coordinate in $-s..s$.



Notice that this statistic keeps the resulting image the same size as the original. It eliminates the small blocks of black pixels but keeps the larger central block as is.

Let's try it on a random image.

```
pgd> data
```

An FRP with value

```
pgd> denoised = denoise(data)
```

An FRP with value

```
pgd> reconstructed = closest_image_to(denoised.value, ImageModels.model('efh'))
```

```
pgd> reconstructed
```

An FRP with value

```
pgd> image_distance(reconstructed, truth)
```

```
0
```

Here, `image_distance` counts the number of pixels in which `reconstructed` and `truth` differ. In `denoised`, we have successfully eliminated the noise pixels and only slightly degraded the base image. Our guess of the unknown base image is the image in our model that is closest to the value of `denoised`. This image, `reconstructed`, turns out to be correct in this case, so we have successfully reconstructed the unknown image.

The `denoised` image in this case loses significant structure in the F, but keep in mind the image is 32×32 and the dilation and erosion elements are 3×3 with

substantial noise. If we increase the resolution of the image and reduce the noise a bit, the denoising shows the structure of the F clearly.

```
pgd> hi_res, _ = ImageModels.observe([ImageModels.image('F#')])
```

An FRP with value



```
pgd> denoised = denoise(hi_res)
```

An FRP with value



```
pgd> denoise2 = erode() ^ dilate(2)
```

```
pgd> denoised2 = denoise2(hi_res)
```

An FRP with value



In the last image, we dilate with a larger element to “fill in the holes.”

Puzzle 36. In `denoise`, we first erode then dilate. What happens if we first dilate then erode? Try it on a few noisy images.

We can encapsulate this procedure into a single statistic. Because the procedure depends on the models and the value of p , we write this as the *statistic factory* `reconstruct_image` in Figure 2.14. We use the **factory trick** here to set up the context in which the desired statistic can be defined and returned based on the parameter (`model`) that we pass. Note that the statistic has access to all the local variables defined in the outer function. Above, we used the equivalent of the statistic `reconstruct_image('efh')`.

To evaluate how well this procedure reconstructs the unknown image, we can repeatedly clone the Data FRP, apply our statistic, and see how close our reconstruction is to the truth. We embody this procedure with a single function, `simulate_denoise` in Figure 2.15. This repeatedly simulates data from the model, applies the given denoiser, and computes two metrics of how well the denoiser performs: the proportion of incorrect reconstructions and the average distance between the reconstructed and true images. We want smaller numbers for both metrics. To use it, we specify a model and a denoising function (like `reconstruct_image`). We can also control the noise level and number of simulated data sets to use.

Now, we can evaluate our procedure with different models and different noise specifications. For instance:

```
def reconstruct_image(model_id, denoiser=erode() ^ dilate()):
    """A statistics factory for reconstructing an unknown image from noisy data.

    Parameters
    -----
    model_id: int | str - an identifier for the model holding candidate base images
    denoiser: Statistic - a denoising statistic mapping image to image

    """
    model_images = ImageModels.model(model_id)

    @statistic
    def reconstruct(observed_image):
        denoised_image = denoiser(observed_image)
        return closest_image_to(denoised_image, model_images)

    return reconstruct
```

FIGURE 2.14. Statistic factory for reconstructing an unknown image from noisy data using a model.

```
pgd> reconstruct = reconstruct_image('efh')
pgd> simulate_denoise('efh', reconstruct)
(0.1831, 8.3244)
pgd> simulate_denoise('efh', reconstruct, p='1/32')
(0.008, 0.24)
pgd> simulate_denoise('efh', reconstruct, p='1/4')
(0.5667, 43.5033)
```

In the first case, reconstruction is correct about 82% of the time and even when they are not, it differs by only about $45 \approx 8.3244/0.1831$ pixels on average from the true image.³⁶ As the noise level gets smaller or larger, the reconstruction performance gets better or worse, as we would expect. Try it yourself with several different possibilities.

Puzzle 37. Briefly explain what you take from these simulation results.

Our second statistic for reconstructing the unknown base image uses a different approach: over all candidate base images and all values of p , we choose that which *makes the image we observed as likely as possible* using the *Kind* of the Data FRP. This is called the **maximum likelihood method** and is based on the **likelihood principle**. If we look at the difference between the observed image and a particular

³⁶The distance zero is correct, so the average distance is $8.3244 = 0.1831d + 0.81790$, where d is the average distance for incorrect cases.

```

def simulate_denoise(
    model_id, denoiser,
    p='1/8', observations=10_000
):
    """Evaluates denoising statistic on repeated observations from a model.

    Parameters:
    + model_id - identifier of pre-defined model in ImageModels
    + denoiser: Statistic - a denoising statistic mapping image to image
    + p: ScalarQ - noise prevalence (0 <= p <= 1), numeric or string
    + observations: int - number of observed images to generate

    Returns a pair of numbers giving (i) the proportion of incorrect
    reconstructions over all observations, and (ii) the average
    distance between truth and reconstruction over all observations.

    """
    prop_wrong = 0
    score = 0
    for _ in range(observations):
        data, truth = ImageModels.observe(model_id, p=p)
        reconstructed = denoiser(data)
        distance = image_distance(reconstructed.value, truth)
        score += distance
        prop_wrong += (distance > 0) # 0 if correct, 1 if not
    return (prop_wrong / observations, score / observations)

```

FIGURE 2.15. Function that repeatedly simulates data from a specified model and applies a given denoiser.

candidate base image, what should we see? If the candidate image is the true base image – the one used to generate the data – what is left should look like noise (produced with some value of p). But if the candidate image differs from the true base image, we will see noise pixels *plus* excess black/1 pixels from the difference between the candidate and true base images. Combining two binary images with a pixelwise exclusive-or sets only pixels where the two images differ. If we combine a candidate base image with the observed image in this way, we will see a purely noise-like image when we have the right candidate and noise plus something more otherwise. So, we use the *Kind* of the noise FRP to assess how well each candidate base image “fits” the observed data, and pick the best fitting choice. Although these Kinds have large size, we can use mathematical reasoning to compute this efficiently.

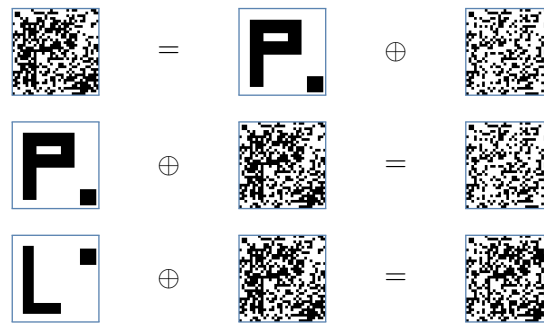


FIGURE 2.16. (Top) The observed image can be expressed as a combination (pixelwise exclusive-or) of the (unknown) base image and a noise image. (Middle) Combining the correct base image with the observed image gives us noise. (Bottom) But using the wrong base image gives us noise plus an excess of pixels where the candidate base image differs from the truth, which is visible in the rightmost image.

It will be helpful to formalize this one step further. For any base image $\mathcal{J}$, define the statistic $\psi_{\mathcal{J}}$ that maps an image $\mathcal{Z}$ (of the same size as $\mathcal{J}$) to the image that has a 1 at any pixel where $\mathcal{J}$ and $\mathcal{Z}$ disagree and a 0 at any pixel where they agree. This is just the pixelwise exclusive-or of the two images, $\mathcal{J} \oplus \mathcal{Z}$; it shows us where $\mathcal{Z}$ differs from $\mathcal{J}$.

Let Y be the Data FRP that represents the observed image and suppose that it was actually generated with base image $\mathcal{I}$ and noise parameter $0 \leq p \leq 1$. Then, the FRP $Z = \psi_{\mathcal{I}}(Y)$ represents an image of *purely noise*, as would be produced by `random_image(p)`. That is, if $\mathcal{Y}$ is the value of Y (i.e., Y ’s output image) and $\mathcal{Z}$ is the value of Z , then $\mathcal{Y} = \mathcal{I} \oplus \mathcal{Z}$. This relationship is illustrated in the top panel of Figure 2.16.

Because $\mathcal{I} \oplus \mathcal{I}$ is the empty image, it follows as well that $\mathcal{Z} = \psi_{\mathcal{I}}(\mathcal{Y})$. That is, if we take the difference of $\mathcal{Y}$ and $\mathcal{I}$, we get $\mathcal{I} \oplus \mathcal{Y} = \mathcal{Z}$, the noise image. This is illustrated in the middle panel of Figure 2.16.

However, if we instead use the *wrong* base image $\mathcal{J} \neq \mathcal{I}$ for the latter operation, we get noise plus residual parts of the base images. Specifically, the value of $\psi_{\mathcal{J}}(Y)$ is $(\mathcal{J} \oplus \mathcal{I}) \oplus \mathcal{Z}$. The $\mathcal{J} \oplus \mathcal{I}$ part is the difference between the base image we used and the true base image that generated the data. This will add “excess” pixels to the result. This is illustrated in the bottom panel of Figure 2.16.

We can use these relationships to choose a base image and a value of p . Assume that $p \leq 1/2$; this is not essential but it makes sense in practice. For candidate base image $\mathcal{J}$ to be the true base image, then $\mathcal{Z}$ would need to have the same value as $\psi_{\mathcal{J}}(Y)$. ***The weight of this value in the Kind of $\mathcal{Z}$ quantifies how likely we are to see that value.*** We call this weight the **likelihood** of the corresponding image. The Kind of $\mathcal{Z}$ depends on p , so for each $\mathcal{J}$ and each p , we get a score – the weight on the branch for value $\psi_{\mathcal{J}}(\mathcal{Y})$ in the Kind of $\mathcal{Z}$. We choose the $\mathcal{J}$ and p pair that maximizes this weight (the likelihood).

Formally, we define our reconstruction procedure as a statistic φ . Let $K_p(z)$ be the weight associated with value z in `kind(random_image(p))`, the Kind of noise generated with parameter $0 \leq p \leq 1/2$. The statistic φ takes an image $\mathcal{Y}$ and chooses a candidate base image $\mathcal{J}$ and noise parameter p to *maximize* $K_p(\varphi_{\mathcal{J}}(\mathcal{Y}))$, the weight associated with the “noise” that we see. This reconstructs the image by making the data we observed maximally likely.

In the playground module `frplib.examples.random_images`, the statistic φ is defined with `max_likelihood_image`. Like `reconstruct_image` earlier, this is a statistic factory that returns a statistic for the given model identifier (and an optional indication of whether to include the estimated p at the end of the returned tuple). Using `data` and `truth` generated earlier, we have

```
pgd> ml_image = max_likelihood_image('efh', return_p = True)
pgd> *max_liked, p_hat = ml_image(data).value
pgd> image_distance(max_liked, truth)
0
pgd> p_hat
0.131      # true value is 0.125
```

Here, `ml_image(data).value` returns the value of the feature FRP `ml_image(data)` which is a tuple containing the reconstructed image in the initial components and the estimated p in the last component. On the left side of that line, we deconstruct

this tuple into those two pieces. Remember that `ml_image` does not *know* the value of p ; it estimates it purely from the data. We can use the same type of simulations as earlier to evaluate this procedure.

```
pgd> ml_recon = max_likelihood_image('efh') # doesn't return estimated p
pgd> simulate_denoise('efh', ml_recon)
(0.0, 0.0)
pgd> simulate_denoise('efh', ml_recon, p='1/32')
(0.0, 0.0)
pgd> simulate_denoise(3, ml_recon, p='1/4')
(0.002, 0.06)
```

This performs very well, perfectly reconstructing in this simple model with low to moderate noise level p . This works because we have the excess pixels from the differences among the three candidate images are very unlikely to occur by chance. Try it yourself with several different possibilities.

Key Take Aways

The random graphs example illustrates, among other things, how we can analyze systems of real interest and how statistics embody questions about those systems that we want to predict.

The random images example illustrates similar points, and in addition, gives a practical example of learning from data. Our model for the system includes an assumption about the unknown base image, and we can use the likelihood principle (or other approaches) to identify the model that best “fits” the data (i.e., the noisy image) that we observed.

Chapter Checkpoints

After reading this Chapter you should be able to:

- Define a statistic and give several examples of useful statistics.
- Explain how to transform an FRP or Kind using a statistic.
- Explain what it means for a statistic and FRP/Kind to be compatible.
- Use the playground to construct statistics.

- Use the playground to transform an FRP or Kind using a statistic.
- Define the components of an FRP.
- Use projection statistics (via `Proj`) to find the Kind of an FRP component and to construct the FRP for a component.
- Find help and documentation on built-in statistics, factories, and combinators in the playground.
- Use built-in statistics, factories, and combinators in the playground.

frplib Overview

Most operations in `frplib` can be categorized as either [factories](#), [combinators](#), or [actions](#). Factories create things, combinators combine existing things into a new thing, and actions use things to produce an effect. The list below shows some commonly used functions; see the `frplib` documentation, help, and cheatsheet for more.

Kind Factories

- `constant` – the Kind of a constant FRP with specified value
- `uniform` – the Kind with specified values and equal weights
- `choice` – `choice(a,b,w=1)` has values `a` and `b` with weights `1` and `w`
- `weighted_as` – specified weights on arbitrary values
- `weighted_by` – weights on values determined by a general function
- `kind` – constructs a Kind from various types of objects.
- `conditional_kind` – constructs a conditional Kind from a dict or function
- `fast_join_pow` – computes `mstat(k ** n)` efficiently
- `evenly_spaced`, `integers`, `symmetric`, `linear`, `geometric` – kinds on specified values with patterns of weights
- `subsets`, `without_replacement`, `permutations_of`, `ordered_samples`
- `arbitrary` – the Kind with specified values and symbolic (unspecified) weights

FRP Factories

- `frp` – constructs an FRP from a Kind or clones another FRP.
- `conditional_frp` – constructs a conditional FRP from a dict or function
- `shuffle` – an FRP that shuffles a given sequence

Kind and FRP Combinators

- `^` operator – `a ^ stat` and `stat(a)`, transform `a` with statistic
- `*` operator – `a * b` is the independent join of `a` and `b`
- `**` operator – `a ** n` is the independent join of `a` with itself `n` times
- `>>` operator – `a >> b` is the join with source `a` and target `b`
- `|` operator – `a | c` is the conditional of `a` given the condition `c`
- `//` operator – `b // a` (read “`b` conditioning on `a`”) is equivalent to `a >> b ^ Proj[-b.dim, -b.dim+1, ..., -1]`

frplib Overview (cont'd)**Statistic Factories**

`statistic`, `condition`, `scalar_statistic` – convert a function into a statistic
`Constantly` – a statistic that always returns the same value
`Proj` – produces a projection statistic on the given indices
`Permute` – produces a permutation statistic with the given permutation

Statistics

`__`, `Scalar` – stands for the value passed in, the latter forces a scalar
`Sum`, `Product`, `Min`, `Max`, `Count` – arithmetic operations on value's components
`Mean`, `StandardDeviation` – statistical summaries of a value's components
`Abs`, `Dot` – absolute value/norm and dot product with a specified vector
`Diff` and `Diffs` – successive differences of the values components
`Exp`, `Log`, `Log2`, `Log10`, `Sin`, `Cos`, `Tan`, `Sqrt`, `Floor`, `Ceil`, `NormalCDF` – scalar mathematical functions
`top`, `bottom` – statistics that always return true and false

Statistic Combinators

`^` – `s1 ^ s2` (“`s1` then `s2`”), equivalent to `s2(s1)`
`@` – `stat @ X` is like `stat(X)` but passes `X` to a following conditional
`And`, `Or`, `Xor`, `Not` – logic operators
`Fork` - `Fork(f1,f2,...,fn)` applies each `fi` to its input
`ForEach` – apply a statistic to each component of a value
`IfThenElse` – if a condition is true, apply one statistic else another.

Actions

`symbol`, `symbols` – create symbolic quantities with given names
`clone` – create copy of an FRP or conditional FRP with its own value
`unfolded`, `unnormalized`, `raw` – alternate displays of a Kind tree
`clean` – remove branches with numerically negligible weights
`FRP.sample` – activate clones of a given FRP

Utilities

`dim`, `codim`, `size`, `values` – get properties
`irange`, `index_of` – inclusive integer ranges and index finding
`identity`, `const`, `compose` – useful functions
`frequencies` – compute frequencies of values in a sequence
`as_quantity`, `tup` – convert to quantity (numeric/symbolic) or an `frplib` tuple
`numeric_abs`, `numeric_log`, `numeric_exp`, `numeric_sqrt` – scalar ops

Help

`info` – `frplib` specific help
`help` – built-in python help

Equivalent Kinds and Canonical Forms

3

Chapter

An FRP produces a single value, fixed for all time once it is activated. So how can we predict anything about the value a fresh FRP will eventually produce? Fortunately, we have seen in our empirical investigations that we can make predictions *in the aggregate* by sampling many FRPs of the same *Kind*. The Kind represents an “ideal” version of the demo where we effectively include *all* FRPs of the that Kind. What we see in a demo with a *finite* number of FRPs will vary somewhat, but as we sample more and more FRPs of that Kind, the relative frequencies of the values we see in the demo will more closely match the weights in the Kind.

Kinds are described by weighted, complete trees, with values of the same dimension on the leaves and positive numbers for the weights. As we intuited from our earlier explorations, it is possible to have different trees that are *equivalent* in terms of the predictions they make about the value of an FRP of that Kind. In this Chapter, we take a closer look at when two Kind trees are equivalent and see how to convert among different representations of equivalent Kinds.

To illustrate equivalence, we will look at two pairs of Kinds that have different trees but produce the same demo tables up to sampling variation. We will use the `Market.compare` task to run demos for two different Kinds in parallel.

We start with a pair of simple scalar Kinds with size 2 that differ only in the scaling of their weights.

```
pgd> k1 = weighted_as(0, 1, weights=[1, 1])
pgd> k2 = weighted_as( 0, 1, weights=[1/2, 1/2])
pgd> Market.compare(1_000_000, k1, k2)

      Kind A
      ,----- 1 ---- <0>
<0> -|
      `----- 1 ---- <1>
```

```

      Kind B
    ,----- 1/2 ----- <0>
<> -|
    `----- 1/2 ----- <1>

```

```

Summary of Demo for Kind A
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| 0      | 499687 | 49.97% |
| 1      | 500313 | 50.03% |
+-----+-----+-----+

```

```

Summary of Demo for Kind B
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| 0      | 499629 | 49.96% |
| 1      | 500371 | 50.04% |
+-----+-----+-----+

```

The proportions here are not exactly the same, but the small differences can be ascribed to sampling variation – differences between the finite samples of FRPs in the two demos. Try running this comparison multiple times, and you will see that both sets of proportions fluctuate around 50-50. The more demos we run, the closer the results for the two Kinds will become. These two Kinds are equivalent.

Next we look at a pair of Kinds that differ in width but prove to be equivalent. Here, we will use the `branch` Kind combinator to build a Kind of width > 1 from other pieces.

```

pgd> k1 = branch(weighted_as((-1, -15), (-1, -5), weights=[2, 3]),
...>           constant(0, 10),
...>           weighted_as((9, 12), (9, 20), (9, 32), weights=[1, 4, 5]),
...>           weights=[1, 3, 2])
pgd> k2 = weighted_as((-1, -15), (-1, -5), (0, 10), (9, 12), (9, 20), (9, 32),
...>           weights=[1/15, 1/10, 1/2, 1/30, 2/15, 1/6])

```

Again, we use `Market.compare` to compare the demo tables with parallel samples. This also shows the Kinds.

```
pgd> Market.compare(1_000_000, k1, k2)
```

Kind A

```

,----- 2/5 ---- <-1, -15>
,----- 1 ---- <-1> -|
|                        `----- 3/5 ---- <-1, -5>
|
|
|
<> -+----- 3 ---- <0> -+----- 1 ----- <0, 10>
|
|
|
,----- 1 ----- <9, 12>
`----- 2 ---- <9> -+----- 4 ----- <9, 20>
                        `----- 5 ----- <9, 32>

```

Kind B

```

,----- 2/30 ----- <-1, -15>
|
|----- 3/30 ----- <-1, -5>
|
|----- 15/30 ----- <0, 10>
<> -|
|----- 1/30 ----- <9, 12>
|
|----- 4/30 ----- <9, 20>
|
~----- 5/30 ----- <9, 32>

```

Summary of Demo for Kind A

Values	Count	Proportion
<-1, -15>	66311	6.631%
<-1, -5>	100016	10%
<0, 10>	500824	50.08%
<9, 12>	33216	3.322%
<9, 20>	133040	13.3%
<9, 32>	166593	16.66%

Summary of Demo for Kind B		
Values	Count	Proportion
<-1, -15>	66741	6.674%
<-1, -5>	99935	9.993%
<0, 10>	499578	49.96%
<9, 12>	33170	3.317%
<9, 20>	133211	13.32%
<9, 32>	167365	16.74%

Looking to the demo tables, we again see small variations between finite samples of FRPs, but these two Kinds are also equivalent.

We will define the equivalence of two Kinds in terms of the risk-neutral prices of transformed FRPs with those Kinds. As discussed earlier, the risk-neutral price is how much we would pay – ignoring our personal aversion or attraction to risk – for the value produced by an FRP and as such represents our best prediction of that FRP’s value.¹ We use transformed FRPs here to reflect the idea that equivalent Kinds should give the same predicted answers to every valid question.

¹We will see how to *find* these prices in Chapter 6; for now, we only need the idea.

Two Kinds k and k' are equivalent if two conditions hold:

1. every scalar statistic compatible with k is compatible with k' , and vice versa;
2. for every compatible scalar statistic ψ , we are indifferent to exchanging a fresh FRP with Kind $\psi(k)$ for a fresh FRP with Kind $\psi(k')$ at no cost.

The first condition holds if and only if the two Kinds have the *same set of values on their leaves*. The second condition means that FRPs with Kinds $\psi(k)$ and $\psi(k')$ *have the same risk-neutral price*: our predictions of these FRPs’ values are the same.

To understand the role of the statistics ψ here, think of the perspective from the last section: where each statistic corresponds to a question we might ask about the unknown value of the FRP with the transformed value answering that question. A scalar statistic corresponds to a question with a numerical answer. If we have FRPs X and X' with respective Kinds k and k' , then for any scalar statistic ψ , the values of $\psi(X)$ and $\psi(X')$ are the answers to the question for the values of X and X' . **When k and k' are equivalent, it means that our predicted answers to the question are the same for any meaningful question we might ask.**

Loosely speaking, two Kinds k and k' are equivalent when, before observing any

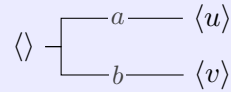
FRPs' values, we are indifferent to replacing any FRP of Kind k with an FRP of Kind k' and vice versa *no matter what transformation rule* our adversary chooses. If one Kind were easier to predict or tended to produce bigger payoffs or if we could *distinguish* the Kinds based on the values we see, then we would not be indifferent between them. Equivalent Kinds lead to FRPs that are *interchangeable*.

Definition 13. Equivalent Kinds. Two Kinds k and k' are **equivalent**, which we denote by $k \cong k'$, if both the following conditions hold:

1. k and k' have the same sets of values on their leaves.
2. For any compatible, scalar statistic ψ , FRPs with Kinds $\psi(k)$ and $\psi(k')$ have the *same risk-neutral price*.

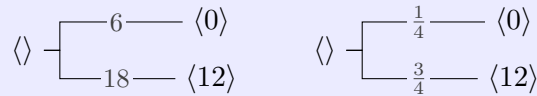
Another way to think about this is that if you have a large demo of FRPs that all have either Kind k or k' , then you cannot distinguish between equivalent Kinds by looking at the values of these FRPs, even in the aggregate. Fresh FRPs with equivalent Kinds are thus interchangeable; we are indifferent to which we have as all our predictions about their values are the same.

Puzzle 38. A Kind k of the form



with $a, b > 0$ and $u \neq v$ has risk-neutral price $\frac{au+bv}{a+b}$, as we will see in Chapter 6.

Use this and the definition of equivalence to show that the following two Kinds are equivalent:



As a first step, write the tree for the transformed Kind $\psi(k)$ for an arbitrary, compatible scalar statistic ψ . You can do this without having a concrete expression for ψ ; the answer depends only on $\psi(u)$, $\psi(v)$, a , and b .

Equivalence ($\cong$) gives a binary relation on the set of Kinds, and in fact it is what we call an *equivalence relation*:² any Kind is equivalent to itself ($k \cong k$), the relation is symmetric ($k \cong k'$ if and only if $k' \cong k$), and the relation is transitive ($k \cong k'$ and

²The notions of relation and equivalence relation are defined formally and discussed in Chapter 16 of Interlude F.

$k' \cong k''$ implies $k \cong k''$). The relation $\cong$ partitions the set of all Kinds into disjoint subsets of equivalent Kinds called *equivalence classes*. All Kinds in an equivalence class are equivalent to each other and are not equivalent to any Kind outside the equivalence class.

Condition 1 in the definition of equivalence is easy to check just by inspecting the trees for two Kinds. We look at the leaves and see if the sets of possible values are the same. Condition 2, however, seems harder to check, even with the formula for the risk-neutral price that we will derive later. How do we check it for all compatible, scalar statistics? Fortunately, because we have an equivalence relation, there is an easier way. We systematically choose one representative Kind for each equivalence class, called the **canonical form** for Kinds in the equivalence class, and convert any Kind to its canonical form. **Every Kind is equivalent to its canonical form, and thus by transitivity, any two Kinds with the same canonical form are equivalent.**

Given a Kind tree, there is a simple algorithm to produce its canonical form. To motivate this, we consider three arbitrary choices we make in specifying a Kind: the order of branches at each node, the scaling of the weighs, and the width of the tree. Different choices in these dimensions lead to equivalent Kinds, so if we make a canonical choice in each, we get the canonical form.

First, when we display a Kind, we must specify the order of branches at each node, but this choice is arbitrary. For example, in Figure 3.1, the two Kinds differ only in branch order. They have the same values and for each value, the same weights along the path from root to leaf. If we run demos of both Kinds and tabulate the values in the market, the results do not depend on the branch order.³ FRPs with these Kinds are indistinguishable in their behavior. So: *Kinds that differ only in the order of branches at a node are equivalent.*

³Try this yourself, using compare in the market.

We can choose a *canonical* branch order. Any choice will do, but it helps to be systematic. At each branching, order the nodes in increasing order of the last component in the value. Equivalently, we order the branches so that the leaf tuples are sorted from bottom to top in increasing lexicographic order (sort first by the first component, then by the second, and so forth). We are not required to use this order, but it provides a standard for comparison, as we will see below. For example, the top Kind in Figure 3.1 is in canonical branch order.

Another choice we have to make in specifying a Kind is the scaling of the weights. That is, we can multiply the weights at any branching by the same constant. Consider the Kinds in Figure 3.2. Are these distinguishable? Compare these using

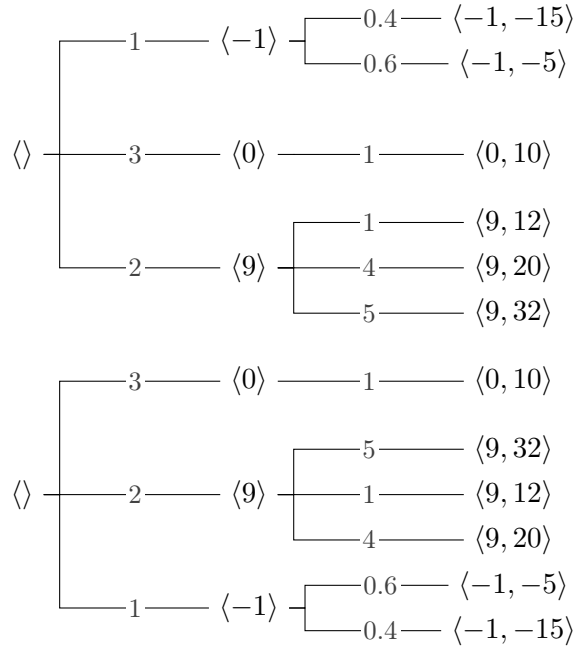


FIGURE 3.1. Two Kinds that differ only in the order of branches at some nodes.

`Market.compare`, as shown earlier. Can you tell these apart?

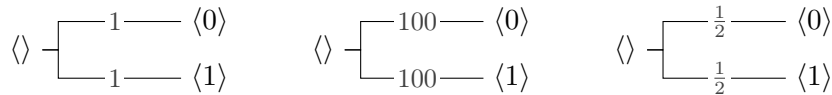


FIGURE 3.2. Three Kinds whose weights differ only by a constant multiplicative factor.

Short answer: no. If we scale all the weights branching from any node in a Kind tree by the same multiplicative factor, we get a new Kind whose demos will be indistinguishable from the original. Two Kinds are related in this way if the weights on the edges branching from some node $w_1, \dots, w_m$ and $w'_1, \dots, w'_m$, satisfy: there is a $c > 0$ where $w_i/w'_i = c$ for every i . The constant can be different at each node, but all the branches emerging from a node must be scaled alike. So: *Kinds that differ only in a constant scaling of the weights weights branching from any node are equivalent.*

We make a canonical choice of scalings: *the sum of the weights emanating from any node should equal 1*. Again, we are not required to use this scaling, and it is sometimes convenient not to, but it serves as a standard to make comparison – and some other calculations – easier.

The last choice we make in presenting Kinds is the width of the tree (i.e., the

number of edges from root to leaf) for Kinds of dimension > 1 . Having multiple levels in the tree emphasizes the sequential nature of the values generated and highlights the contingent choices made for each component of the generated value. This is often conceptually useful when building a model of a random process. At the same time, however, there is redundant information in the multi-level presentation. Figure 3.3 shows two equivalent Kinds with different widths. The Kind shown on the left has the same width as its dimension; we call this the *unfolded* form. The Kind shown on the right has the same size, dimension, and values but has width 1; we call this the *compact* form. It might not seem obvious at first but the two are equivalent, and we can easily convert between them using Algorithms UNFOLDED and COMPACT below. So: *A Kind in unfolded form is equivalent to its compact form, and vice versa.*

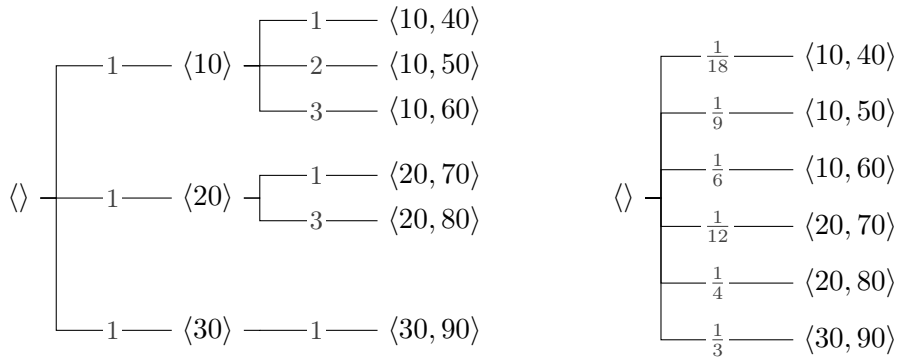


FIGURE 3.3. A Kind in two forms: unfolded and compact. Can you go from one to the other?

Our canonical choice is to show Kinds in compact form. Again, we are not required to use this choice – unfolded form can be illuminating – but it will be our default.

Combining these three transformations gives us a simple algorithm for converting any Kind to its **canonical form**. **Algorithm CANONICAL**, below, shows how to do this in general.

Algorithm CANONICAL

Given as input a Kind k , returns the canonical form of that Kind in three steps:

1. Order the leaves from top to bottom in increasing lexicographic order.
2. At each non-leaf node of the tree, normalize the weights on the edges branching from that node so that they sum to 1.
3. Reduce the tree to compact form using Algorithm COMPACT.

The result is the **canonical form** of Kind k .

The canonical form of Kinds is a simple standard that makes it easy to compare and manipulate Kinds and to identify equivalence. Other equivalent forms can also be useful, giving us insights about the random process or helping with calculations. And for some operations, a non-canonical form can make the crux of the operation clearer. We are free to use any equivalent form a Kind when appropriate.

So from here on, when we consider a Kind, we will effectively identify it with the class of Kinds that are equivalent to that tree. We treat the canonical form as a representative of this class and freely translate to other forms as needed.

Every Kind is **equivalent** to its canonical form. Two Kinds are equivalent if they have the same canonical form.

Puzzle 39. Apply Algorithm CANONICAL to the left Kind in Figure 3.3. You should get the right Kind in that Figure.

Naming Convention.

This is a good point to establish a naming convention for FRPs and Kinds, to make it easier to reference them.

For FRPs, we will name them with *capital Roman letters* (like X , Y , and Z), sometimes with subscripts to indicate FRPs that are related in some way. Thus, we can name FRPs X , Y_1 , Y_2 , R , M , D_T and so forth. We often use integer subscripts to identify scalar component FRPs of a multi-dimensional FRP. If we want to emphasize that a group of FRPs represent distinct FRPs with the same Kind, we will use the same base letter and wrap the subscripts with the index in brackets. For instance, a collection of four like-kinded FRPs $X_{[1]}$, $X_{[2]}$, $X_{[3]}$, $X_{[4]}$.

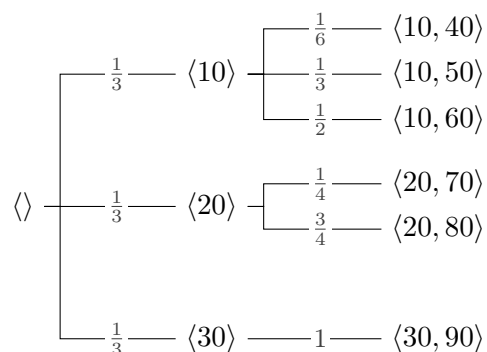
If X is an FRP, $\text{kind}(X)$ is its Kind, $\text{dim}(X)$ is its dimension, $\text{size}(X)$ is its size, and its set of values is $\text{values}(X)$.

For Kinds, we will name them with adorned letters like $k, k', k_1, \dots$, preferring to use the base letter k when convenient.

Remember that, formally, $\text{kind}(X)$ refers to an equivalence class of trees, and we can display it with any of the equivalent trees in the class, with the canonical form by default.

Finally, we turn to the algorithms for compactifying and unfolding a Kind tree. Consider first the Kind at the left of Figure 3.3. We will carry out Algorithm

COMPACT in three steps. First, we normalize the weights so that for each non-leaf node, the branches coming from that node have weights that sum to 1. The tree becomes



We work one non-leaf node at a time, including the root. Typically, we would reduce fractions to lowest terms, but that is not always necessary or clarifying. Then, for each leaf node, we multiply the normalized weights along the path from root to leaf, recording the result. For instance, for the $\langle 10, 40 \rangle$ leaf node, we get $\frac{1}{3} \cdot \frac{1}{6} = \frac{1}{18}$; for the $\langle 20, 80 \rangle$ leaf node, we get $\frac{1}{3} \cdot \frac{3}{4} = \frac{1}{4}$; and so on. Creating a one level tree with the same leaf nodes and the weights corresponding to these products yields the compact form at the bottom of the Figure.

Algorithm COMPACT

Input: a Kind as an unfolded tree

Returns: the Kind in equivalent compact form.

Step 1. At each non-leaf node of the tree, normalize the weights on the edges branching from that node so that they sum to 1.

Step 2. For each leaf node of the tree, multiply together the weights along the path from the root to that leaf. Record the resulting product for that leaf.

Step 3. Create a single level tree with the same leaf nodes and set the weight for each leaf node to be the product you computed for that node in Step 2.

The resulting Kind tree is the compact form.

In `frplib`,⁴ we can view and manipulate Kinds to understand this transformation. FRPs and Kinds can be assigned to variables for easy reference. Pick a Kind, apply the algorithms, and then use commands like the following to see both forms.

⁴When showing playground input and output, text from # to the end of a line is a comment for your benefit. You should not type or enter that.

```

pgd> k1 = weighted_as((3, 2), (3, 3), (30, 0), (30, 2),
...>                  weights=[15, 105, 44, 88])
pgd> k1                # shows the canonical form (the default)
pgd> unfolded(k1)      # shows the unfolded form, normalized at each branch
pgd> unnormalized(k1)  # shows unnormalized, width 1 form
pgd> raw(k1)           # unnormalized and unfolded

```

The different forms here only determine how they are displayed; internally, the canonical form is used for all calculations.

Now let's go the other way, starting from the right tree Figure 3.3 and producing the left. The key to making this work is that each value generated at each level must be distinct. First, we normalize the weights as in the previous two algorithms, then we build the unfolded form *from the leaves up*.

The leaf nodes in the unfolded form will be the same as in the compact form. To get the nodes at the next higher level, we remove the *last* value in the list. When we do this, we get three $\langle 10 \rangle$'s and two $\langle 20 \rangle$'s, and because values must be distinct, we need to combine each of these sets into a subtree. Let's focus on the $\langle 10 \rangle$'s and the three nodes from which they come. These nodes will be grouped in a subtree with $\langle 10 \rangle$ at the branch. Add together the weights for these three nodes, yielding $\frac{1}{18} + \frac{1}{9} + \frac{1}{6} = \frac{1}{3}$. We carry the $1/3$ forward and divide each of the nodes' weights by $\frac{1}{3}$ to renormalize the sum to 1. Our subtree weights then become $\frac{1}{6}$, $\frac{1}{3}$, and $\frac{1}{2}$ respectively. Now we repeat the process with the node $\langle 10 \rangle$. We remove the last item from the list which gives the empty node; there are no repeats here. The $\frac{1}{3}$ that we carried over becomes the weight for that edge.

For completeness, let's do the other two cases. The leaf nodes that start with 20 have weights $\frac{1}{12}$ and $\frac{1}{4}$; adding these gives $\frac{1}{3}$. Renormalizing gives weights $\frac{1}{4}$ and $\frac{3}{4}$ for the subtree with nodes $\langle 20, 70 \rangle$ and $\langle 20, 80 \rangle$, carrying $\frac{1}{3}$ forward. Repeating for $\langle 20 \rangle$ brings us to the root, so that edge has weight $\frac{1}{3}$.

The node is $\langle 30, 90 \rangle$. We remove the 90, but there no duplicates and the weight is $\frac{1}{3}$, which normalizes to 1, carrying $\frac{1}{3}$ forward. Removing 30 brings us to the root with a weight of $\frac{1}{3}$. The result is as in the earlier Figure.

We can carry out these same operations for arbitrary Kinds. The general method is described in **Algorithm UNFOLDED**.

This algorithm is rather easier to do than to precisely describe, so try it out on some examples with the following activity.

Algorithm UNFOLDED

Input: a Kind as a compact (single level) tree

Returns: the Kind in equivalent unfolded form.

Step 1. Convert the compact tree to canonical form; in particular, normalize the weights in the input Kind so that they sum to 1. Call this T_0 .

Step 2. Define two kind-valued variables S and T . Initialize both to T_0

Step 3. While Kind S has dimension > 1 , do the following:

- i. Partition the leaf nodes S into disjoint sets $\mathcal{L}_1, \dots, \mathcal{L}_m$ (for some $m \geq 1$) of leaf nodes whose values are equal excluding the last element.
- ii. For $\mathcal{L}_j$ with $j \in [1..m]$, do the following:
 - a. Let $n_1, \dots, n_k$ be the leaf nodes in $\mathcal{L}_j$, with values of the form $\langle v_1, \dots, v_{d-1}, x_i \rangle$ $d = \dim(S)$ and $i \in [1..k]$, where $v_1, \dots, v_d$ are the same for all k nodes and $x_1, \dots, x_k$ are distinct.
 - b. Modify T by removing the edges from the common parent of the nodes $n_1, \dots, n_k$ and replacing them with an edge from that common parent to a new node b_j and with edges from b_j to each node $n_1, \dots, n_k$.
 - c. Set the value for node b_j to $\langle v_1, \dots, v_{n-1} \rangle$.
 - d. If $w_1, \dots, w_k$ are the weights on the edges from $n_1, \dots, n_k$ to their original parent in T , set the new weight on the branch from b_j to each n_i to $w_i/(w_1 + \dots + w_k)$ and the weight on the branch from b_j to its parent to be $w_1 + \dots + w_k$.
- iii. Set S to the (upper) subtree of T consisting of all nodes from the root up to and including the new nodes (i.e., $b_1, \dots, b_m$) added in step ii.

Step 4. Return T .

Activity. Generate several Kinds in a join of unfolded and compact forms. Apply the algorithms to convert each to the other form. Use the playground to view each Kind in both forms and check your answers.

Key Take Aways

Kinds are described by complete trees, with values at the nodes and weights on the edges. In our empirical study of FRPs and Kinds, we saw that structurally different trees can produce the *same predictions*. Here, we explore conditions in which two distinct Kinds give equivalent descriptions of an FRP.

Two Kinds k and k' are **equivalent** when (i) they have the *same sets of values* on their leaves, and (ii) FRPs with Kind $\psi(k)$ and $\psi(k')$ have the *same risk-neutral price* for any compatible, scalar statistic ψ . Given FRPs with equivalent Kinds k and k' , one cannot distinguish whether the FRP has Kind k or k' just by observing the FRPs values, even in the aggregate. The bottom line is that FRPs with equivalent Kinds are completely interchangeable.

Kinds that differ only in the order of branches at a node are equivalent. Kinds that differ only in a constant scaling of the weights branching from any node are equivalent. And any Kind can be reduced to an equivalent Kind in *compact* form (i.e., width 1) using Algorithm COMPACT.

Every Kind tree has a **canonical form**, which can be obtained (Algorithm CANONICAL) by

1. Ordering the leaves from top to bottom in increasing lexicographic order.
2. At each non-leaf node of the tree, normalizing the weights on the edges branching from that node so that they sum to 1.
3. Reducing the tree to compact form using Algorithm COMPACT.

Every Kind is equivalent to its canonical form. Two Kinds are equivalent if they have the same canonical form.

Algorithms COMPACT and UNFOLD can be used to convert between a single-level compact form and a multi-level (in general) *unfolded* form.

Chapter Checkpoints

After reading this Chapter you should be able to:

- Explain what it means for two Kinds to be equivalent.
- Determine if two Kind trees are equivalent.
- Convert a Kind tree into canonical form via Algorithm CANONICAL
- Apply Algorithms COMPACT and UNFOLDED to convert back and forth between the compact and unfolded views of a Kind.

Building with Joins

4

Chapter

Contents

4.1	The Evolution of Random Systems	212
4.2	Independent Joins	223
4.3	Conditional FRPs and Conditional Kinds	258
4.4	General Joins	271

4.1 The Evolution of Random Systems

A common feature of many random systems is that they *evolve*: first something happens, then something else happens that depends on what happened initially, then something else happens that depends on what happened earlier, and so on. This evolution can proceed for just a few stages or for many, or it may continue indefinitely. Each stage can inject new randomness into the system. When we see the output of the random system, we get the entire history of its evolution, with data that shows what happened at each stage. For modeling and analysis of random systems, treating the entire history as a whole can be complex and challenging. It is often easier to reason about what happens at each individual stage, taking the output of all previous stages as a given. Any random system, however complex its evolution, can be built by combining such simpler pieces. The operation that combines the random output of earlier stages with a contingent source of randomness for the next stage is called a [join](#).

A join builds a new FRP from two or more other FRPs. One FRP, called the *source*, represents the output of the earlier stages of the system's evolution. Its value is used to contingently activate exactly one of the other FRPs, which are called the *targets*. The target FRPs collectively represent the random quantities that we can measure at the next stage of evolution. The value of the new FRP joins the value of the source with the value of the activated target, thus giving the whole history of the system's evolution up to that stage.

Figure 4.1 shows an example of a random system built with a join. This system evolves in two stages. First, we select one of three boxes by a random mechanism, encoded numerically as -1 for left, 0 for center, and 1 for right. Then, we withdraw from the selected box the money (in dollars) contained within, which is also random. A key feature of this system is that the random mechanisms that determine the amount of money in the three boxes are different. The output of the second stage (money withdrawn) thus depends in a fundamental way on the output of the first stage (box selected). In this join, the source FRP represents the selected box, and the target FRPs represent the amount of money drawn from each box. The target that is activated depends on the value of the source.

The top, middle, and bottom targets in the Figure correspond to the left, center, and right boxes, as seen in the connections from the source's values to the targets. The source's $\langle -1 \rangle$ output port is connected to the top target's input port, the source's $\langle 0 \rangle$ output port to the middle target's input port, and source's the $\langle 1 \rangle$ output port to

the bottom target's input port.

The differences in the random mechanisms for the amount of money in the three boxes are reflected in the different Kinds of the target FRPs. For instance, the left box may contain \$0, \$4, or \$100, but the right box may contain only \$1 or \$100. We do *require that all target FRPs have the same dimension*.

The targets' All output ports are connected to the input of a fresh, empty FRP through a new piece of hardware called a **multiplexer**. The multiplexer looks like



It accepts multiple inputs on the left and produces one output port on the right. Exactly one of its inputs will be activated, and the multiplexer forwards that input signal to its output. This connection to the empty FRP reconfigures that FRP to be the join of the source and targets, setting its Kind appropriately and its value when it is observed.

FRP Input Port Connections. When a wire is connected to the input port of an FRP and when a value is transmitted on the wire, the receiving FRP is activated, and the transmitted value is joined with the FRPs original value into higher-dimensional tuple that becomes the new value of the FRP. So, if $\langle -1 \rangle$ is transmitted to the input port of an FRP and if that FRP produces the value $\langle 4 \rangle$ on activation, the value output becomes $\langle -1, 4 \rangle$. Connecting to FRPs' input ports thus forwards values down the chain.

The behavior of joins comes from this rule on FRP input-port connections. Pushing the button on the join FRP activates the source, triggering the corresponding output port and sending that value to the corresponding target, activating it. The values of the source and activated targets are joined in a larger tuple that becomes the new output of the wired target. The value contains the entire *history* of the system up to that stage: the selected box and how much money is in the selected box. The Figure shows one sample outcome of this process: the source produces a value of -1 (left box), which triggers the top target FRP that in turn produces a value of $\langle 4 \rangle$ (dollars in the box). The joined value $\langle -1, 4 \rangle$ captures the full history of the system and is displayed when the new FRP is observed.

This construction gives us a new FRP, which we call the **join** of the source and target FRPs. The source FRP selects which of the target FRPs is activated, so the next stage of the system is dependent on the values of the previous stages. The value

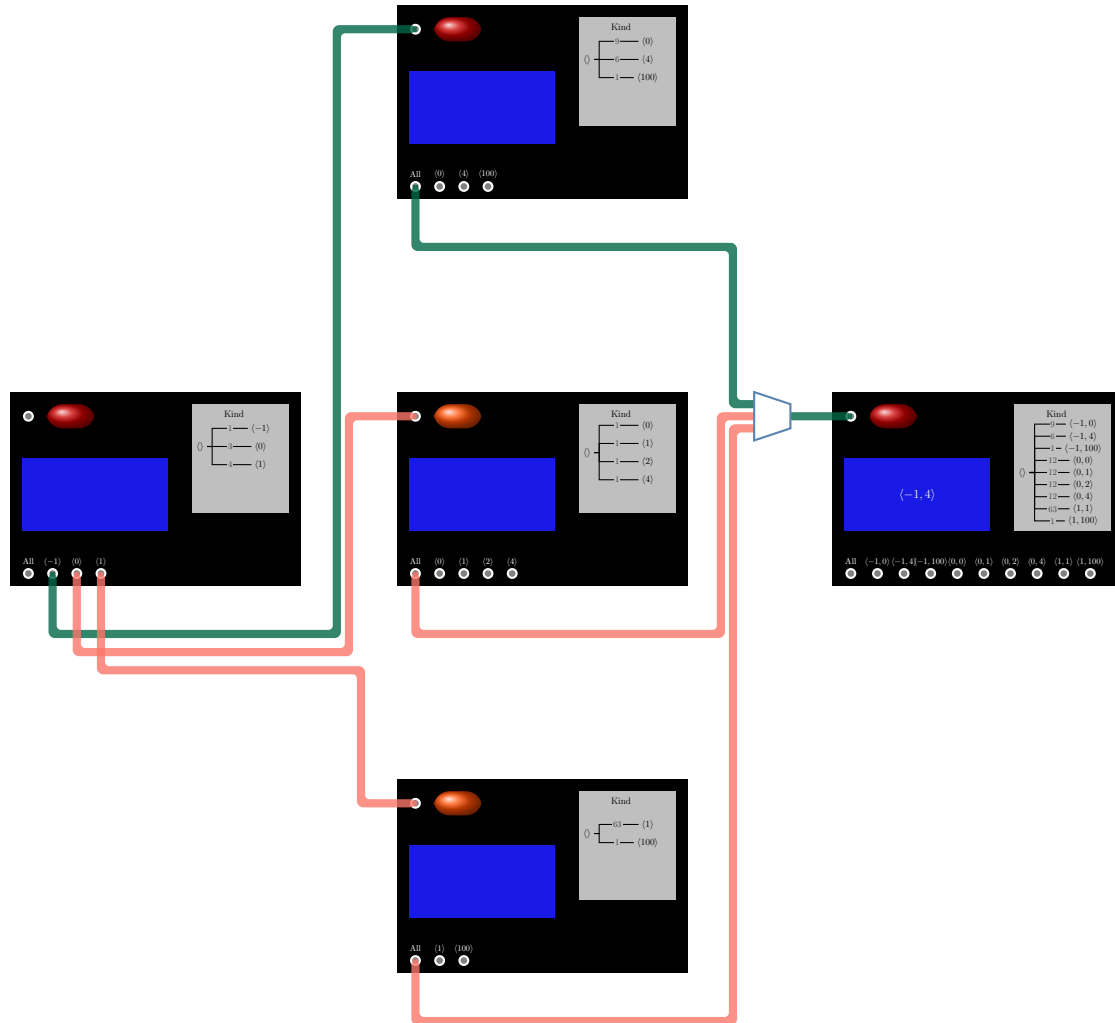


FIGURE 4.1. An FRP join, with the source on the left, the targets in the middle, and the new FRP that is produced by the join on the right. In this case, the activated source has value $\langle -1 \rangle$, which activates (green wire) the top target which has value $\langle 4 \rangle$. The joined value $\langle -1, 4 \rangle$ passes through the multiplexer to the joined FRP becoming its value. The joined FRP is observed, activating the source and one target, and its value is shown on the display.

of a join FRP reflects the systems entire evolution *history* up that stage, combining the values of the source and the activated target. Pushing the button on the join FRP causes it to be observed, first activating the source, whose signal activates one of the targets, and in turn the join FRP. With its button pushed, the join FRP is observed and displays the combined value of source and activated target.

Observing the value of a join FRP activates the source and allows us to infer the value of the source (as it is part of the combined value), but the source FRP only becomes truly observed by pushing its button. (Note that the combined value from the join FRP may not allow us to determine which target was activated; for instance, two of the targets in Figure 4.1 have 4 as a possible value.) Similarly, activating the source, activates the corresponding target and the join, but neither is observed unless its own button is pushed.

The wiring diagram in Figure 4.1 depicts the FRPs in detail, but in practice, we can use a more stylized view.¹ Figure 4.3 reproduces the wiring diagram from Figure 4.1 in simpler form. Each FRP corresponds to a box, which we can label with a name or a Kind or nothing, or whatever is most helpful. Here, we show the Kinds.

¹We will simplify the wiring diagrams joins further in the next section.

Among the Big 3+1 operations, the join operation plays a unique role: it is the only operation that can inject new randomness into a system. Transforms can change the form of a random quantity, but a transform is deterministically related to its source. Observations can constrain existing randomness but not add new random variation. And expectations give us a deterministic value that summarizes the randomness in an FRP. But with joins, the target FRPs represent new randomness introduced at that stage of the system's evolution. This is the main function of the join operation and how it allows us to build more complex random systems from simpler pieces.

Keep in Mind. The join operation is the only one of the Big 3+1 operations that can inject new randomness into a system.

As with transformation by a statistic (and in fact all of the other operations), there is a systematic relationship between the Kind of a join FRP and the Kinds of the source and target FRPs. This defines a join operation on Kinds. We will see the details shortly, but the gist is quite simple. The join operation on Kinds literally joins the Kind trees for each targets to the corresponding leaf of the source's Kind. To be consistent with the Level and Prefix rules for Kinds, we propagate the source values in each subtree into the nodes at lower levels, so the nodes reflect the history to that stage. This is illustrated in Figure 4.2 which constructs the Kind of the join

in Figure 4.1.

The Kind of a Join (informal version). From the wiring diagram for a join FRP, replace each FRP with its Kind, starting from the source and moving from there. Then, where an FRP has a connection to its input port, attach the FRP's Kind tree to the leaf node of the previous stage that corresponds to the port to which the wire is connected.

When attaching a Kind tree to a leaf node, propagate that leaf node's value to every node of the attached tree to be consistent with the Level and Prefix rules, joining the tuples with the propagated value at the beginning of a larger tuple.

This gives the Kind of the join FRP. We can, if we choose, convert that Kind to canonical form.

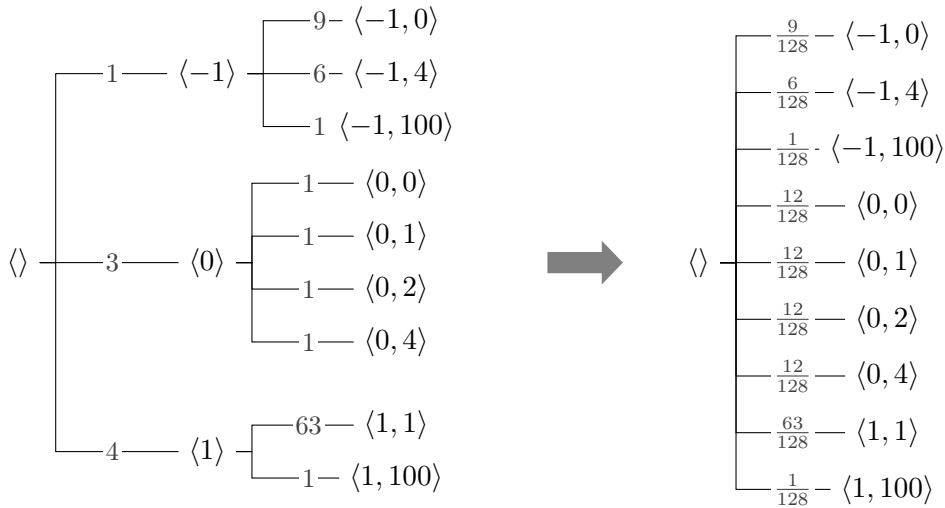


FIGURE 4.2. The join operation on the Kinds of the source and target FRPs from Figure 4.1. On the left is the original join Kind, and on the right, is its in canonical form, with width 1 and weights that sum to 1. Both have dimension 2 and size 9.

Next, we consider the random system where a balanced coin is flipped four times in succession. Each flip is one stage in the evolution of the system and injects additional randomness into the system. This system is built with three successive joins. The first flip is the initial source, and its two targets represent the following flip if the first flip is heads and if the flip is tails. Each FRP representing the second or third flips in turn acts as a source for the next flip in a join with two targets, one for heads and one for tails. A wiring diagram for these joins is shown in Figure 4.4. With repeated joins like this, we can omit the multiplexer and empty FRP at the intermediate stages

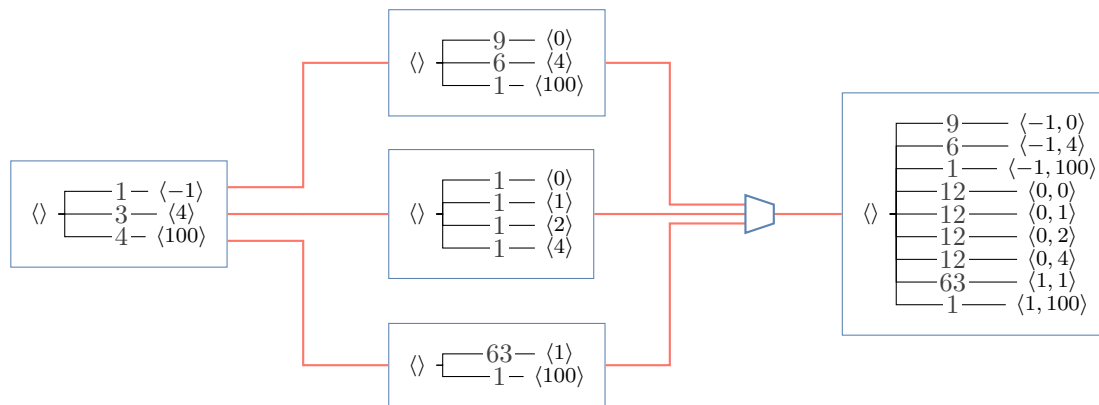


FIGURE 4.3. A leaner depiction of the join in Figure 4.1 as a wiring diagram. Each box is an FRP, labeled by its Kind. The wires connected to the output ports emerge from the right of the box in the same order as the Kind's leaves.

and instead use each target FRP as the source for the next stage *when that target is activated*. Each connection to the input port of an FRP forwards the history to that point to the next stage. The initial source FRP passes a 0 for tails or a 1 for heads to its targets. Each activated target FRP, acting as a source for the next join, passes the previous flips in order and the new flip for that stage. The final targets each have two possible values which concatenates the history of flips up to that stage and the outcome of the final flip.

A notable feature of all the joins in this example is that the outcome of the flip at any given stage is not in any way influenced by the previous flips. All the FRPs representing flips are interchangeable. The history of flips is passed through the values, but the two possible outcomes of each flip have equal weights regardless of that history. We say that each stage is *independent* of the previous stages, so we call these **independent joins**.

With this in mind, consider all the FRPs at the same level in the wiring diagram of Figure 4.4. Because these FRPs all have the same Kind, they are completely interchangeable when fresh, and due to independence (i.e., their values do not depend on the earlier stages other than passing along the history), we do not actually need different FRPs. We can instead have one FRP per level of the same Kind, wired to pass the history forward. The wiring diagram for this independent join becomes even simpler, as shown in Figure 4.5. The join FRPs produced by the wiring diagrams in Figure 4.4 and Figure 4.5 are equivalent, so we can think of an independent join in either way. Note that in the latter Figure, neither the multiplexer nor the empty FRP are necessary, but we keep them as visual and conceptual cues that distinguish

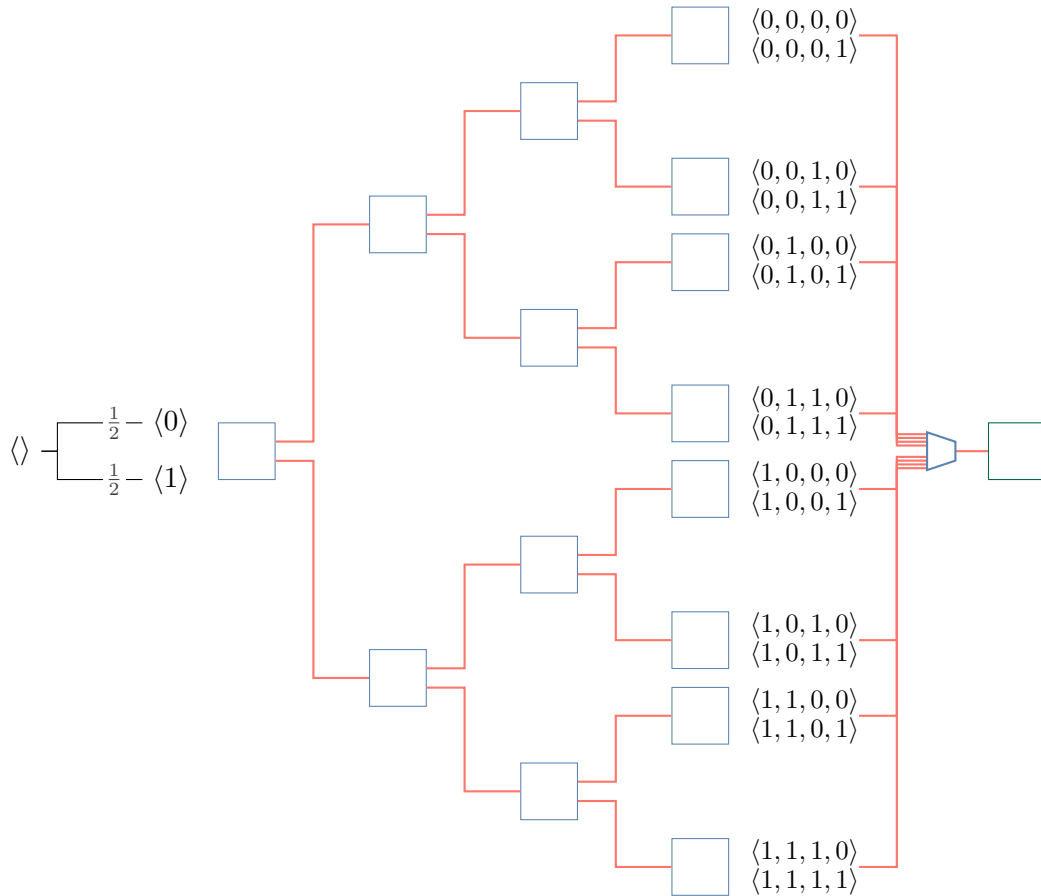


FIGURE 4.4. A join where all the source and target FRPs (blue boxes) have the same Kind, which is shown at left. When doing repeated joins like this, we can omit the intermediate multiplexer/empty FRPs except at the last stage. Instead, at each stage before the last, the targets at that stage act as the source for the next stage. The final targets are labeled with their possible values. The wire connections are in the same order as the leaves of the Kind. The join FRP, shown to the right of the multiplexer in green, has dimension 4 and size 16; its values are all 4-tuples whose components are 0 or 1, as shown.

the component and join FRPs.



FIGURE 4.5. An equivalent but simpler wiring diagram for the join FRP in Figure 4.4. Neither the multiplexer nor the empty FRP is needed in this case, but we keep them to visually and conceptually distinguish the component FRPs from the final join FRP we create.

We have focused so far on the *structure* of the system, how it is built by combining the individual flips. To make predictions about this system, we want to find the *Kind* of the join FRP. Fortunately, *the structure of the Kind derives directly from the structure of the system*. To find this Kind, we use the wiring diagram in Figure 4.1 and follow the approach described in the box “**The Kind of a Join (informal version)**” above. First, we replace each component FRP in the wiring diagram by its Kind. Starting at the source, we glue a copy of the target Kinds’ trees to the each corresponding leaf node, propagating the history forward to be consistent with the Level and Prefix rules for Kinds. We repeat this process for each join – that is, for each successive flip – moving down the tree in the wiring diagram. Since the component FRPs in these joins all the same Kind, this is easy. The result is shown in Figure 4.6.

Algorithm CANONICAL from Chapter 3, can convert this Kind to canonical form if we like. This will be a width 1 Kind with all 16 values having equal weight.

Puzzle 40. Apply **Algorithm CANONICAL** to the Kind in Figure 4.6 and confirm that its canonical form has 16 branches with equal weights.

An alternative, intuitive way to derive the Kind of the join FRP uses a symmetry argument. To set the stage, let’s consider a simple example. I give you a Kind k of width 1 and size 2 with possible values 0 and 1, compute the transform of k by the statistic $(1 - \blacksquare)$, and tell you that both k and its transform are *equal*. What does that tell you about k ? Let’s look at it in `frplib`:

```
pgd> a, b = symbols('a b')
pgd> k = unnormalized(weighted_as(0, 1, weights=[a, b]))
      ,----- a ----- 0
<> -|
      `----- b ----- 1
pgd> k ^ (1 - __)
```

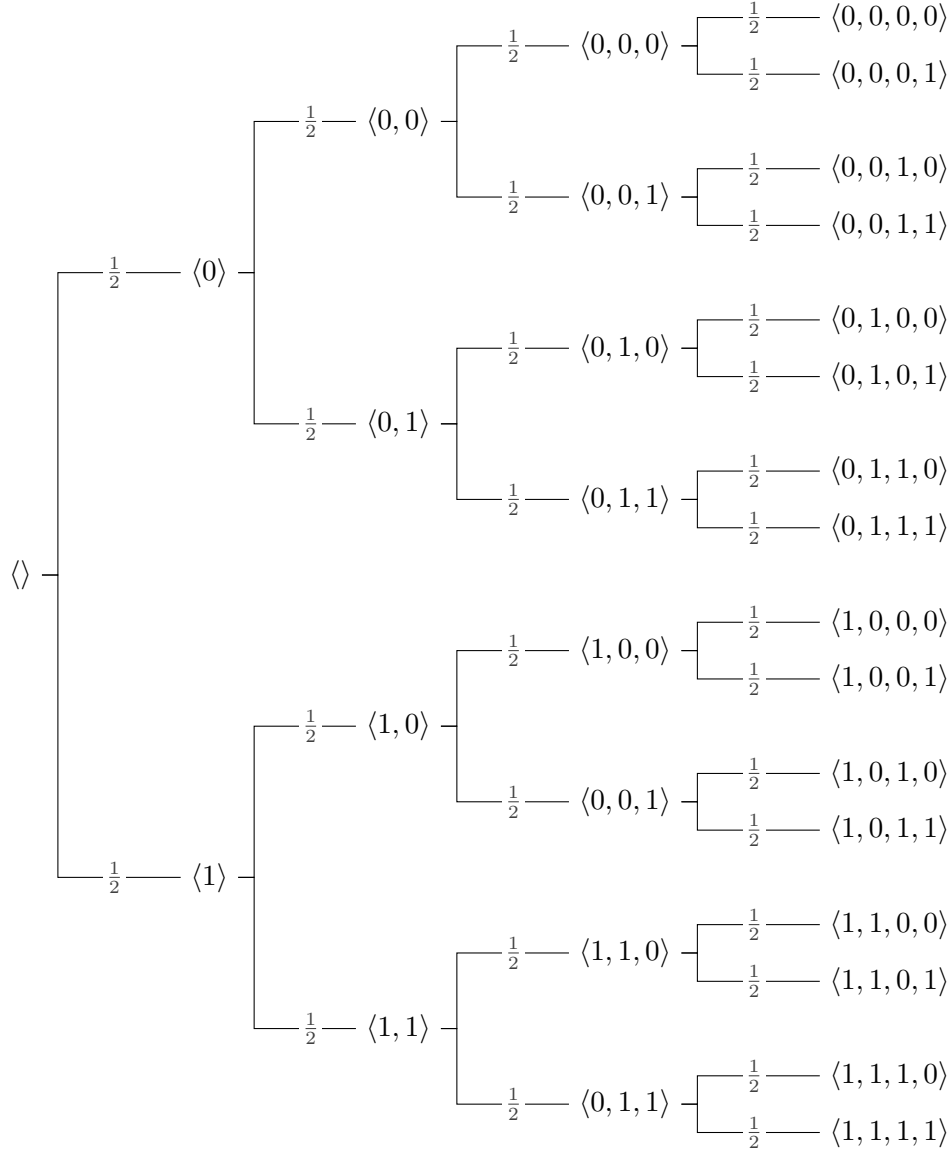


FIGURE 4.6. The Kind of the join FRP in Figure 4.6, created by glueing the Kind trees for each FRP at the corresponding leaf nodes of previous stages, and propagating the history forward at each level. Because all source and target FRPs have the same Kind, the resulting tree assigns the same weight to each path. Converting this to canonical form gives a width 1 Kind with 16 leaf nodes all with equal weight.

```

,----- b ----- 0
<> -|
`----- a ----- 1

```

The first line defines symbolic quantities for a and b ; the last two lines define k and its transform. The statistic $(1 - \blacksquare)$ maps 0 to 1 and 1 to 0,² and the rest follows from the transform operation on Kinds. We can see that if these two Kinds are equal, they must have the same weights, so $a = b$.

²In the language of Section 14.1, the statistic is a *bijection* on the possible values of k .

Puzzle 41. Explain in detail why the transform of k above has the form it has.

We will expand on this idea with the join FRP for four coin flips. We can transform the join FRP by a collection of well-chosen statistics and see that all the transformed FRPs have the same Kind. This will imply that all the values in the original FRP have the same weight.

This approach is illustrated with the wiring diagram in Figure 4.7. We compute the join for the four flips as before but then apply a transformation with a statistic that represents a *permutation*³ of the four components of the FRP's value. Each transformation is with one permutation statistic, but we will consider the transformation for all $4! = 24$ choices of permutation, including the identity permutation. For example, one permutation swaps the first and third and the second and fourth components,⁴ mapping $\langle 1, 1, 0, 0 \rangle$ to $\langle 0, 0, 1, 1 \rangle$ and $\langle 0, 1, 1, 0 \rangle$ to $\langle 1, 0, 0, 1 \rangle$. If we apply this – or any – permutation to the 16 possible values of the join FRP, we get the same 16 values but in a different order.⁵ The join FRP and its transform both have the same 16 possible values. Because the permutation merely reorders the value's components, the wiring diagram in Figure 4.7 is equivalent to the wiring diagram in Figure 4.5 after we *permute the component FRPs of the latter with the same permutation*. Yet all these component FRPs have the same Kind that gives equal weight to heads and tails, so their order does not matter. Thus, all permutations of the boxes in the wiring diagram of Figure 4.5 give an FRP with the same Kind, including the original join FRP.

³See page 674 in Interlude F. Sections 18.2 and 14.1 and the beginning of Chapter 17 are also relevant.

⁴In `frplib`, this statistic is `Permute(3, 1, 4, 2)`.

⁵Again, the permutation is a bijection on the set of 16 values.

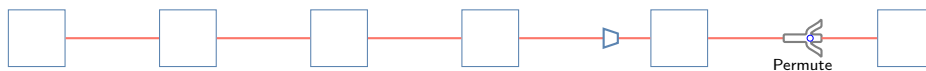


FIGURE 4.7. Transforming the FRP for four flips with a permutation statistic.

In this section, we have seen how joins build more complex random systems from simpler pieces. The joins reflect the structure of the system in describing how each

stage's evolution depends on the previous stage. There is a strong parallel between the structure of the system, as depicted in the join FRP's wiring diagram, and the structure of that FRP's Kind. We use FRPs to represent the data we obtain from the system and Kinds to represent our model and analysis. We can exploit this parallel between the structure of the system and our analysis of the system.

Our goal in this Chapter is to understand the join operation: what it means, how to use it, how to find the Kind of a join, and how to build joins in practice. We think of a join as proceeding in *stages*, with the value of the source FRP being the initial stage, the value of the activated target FRP the second stage, the value of the FRP activated as by that target the third stage, and so on through however many joins we have. The simplest joins are those where the Kinds of the targets do not depend on the value of the join, *independent joins*. We study these in the next section.

Key Take Aways

A **join** is the operation in the Big 3+1 that builds more complex systems from simpler pieces. We think of the system as evolving in stages. A join combines the output of earlier stages with a mechanism for selecting the output of the next stage depending on what the earlier output. The key feature is that the output of the current stage – and the random mechanism used to generate that output – is contingent on what happened in earlier stages. The join operation is the only one of the Big 3+1 that can inject new randomness into a system.

For a join, we start with a *source* FRP whose value represents the output of all earlier stages of the system's evolution. For each possible value of the source, we connect the corresponding output port to the input port of a distinct FRP. These latter FRPs are called the *targets*, one per possible source value, and they can (and often do) have different Kinds. The All output ports of the targets are connected through a *multiplexer* to a new FRP that becomes the join.

When the source is activated, it activates exactly one of the targets, and the combined value becomes the value of the join FRP. A key feature of joins is that they *record the entire history of the process up to that stage*.

Just as there is a join operation on FRPs, there is a compatible join operation on Kinds. We obtain the join of Kinds by joining the Kind trees of sources and targets and propagating values down the tree to form a valid Kind under the Level and Prefix rules. Specifically, replace each FRP in the join's wiring

diagram with its Kind, starting from the source. Where an FRP's input port has a connection, attach the FRP's Kind tree to the leaf node of the previous stage corresponding to that port. When attaching a Kind tree to a leaf node, propagate that leaf node's value to every node of the attached tree, joining the tuples with the propagated value at the beginning of a larger tuple.

When the targets are all “clones” of each other, we get the special case of an *independent join*.

4.2 Independent Joins

Think back to the Monty Hall game in Section 1.4. For any given strategy, the outcome of the game is determined by two stages: Monty's choice of a door and your choice of a door. However, those two choices *do not interact*: you choose a door in exactly the same way whatever Monty does. You do not know what his choice was and are completely uninfluenced by it. The game is represented by an FRP G whose value describes *both* Monty's choice and yours.

This system is built from two Element FRPs: M representing Monty's choice of door for the prize and Y representing your initial choice of door to reveal.⁶ Both components have dimension 1, and they satisfy $M = \text{proj}_1(G)$ and $Y = \text{proj}_2(G)$, so they represent distinct components of the Data FRP. To show this decomposition of the system, we write $G = M :: Y$, overloading the tuple-joining operator $::$ as the values of the components are joined in this way.⁷

⁶Note how we try to give FRPs meaningful names. M for Monty, Y for your.

⁷See Section 15.4 in Interlude F.

Definition 14. Components of an FRP. Let X be an FRP of dimension n , and suppose there are integers $m \in [1..n]$ and $0 = i_0 < i_1 < i_2 < \dots < i_{m-1} < i_m = n$, and FRPs $U_1, U_2, \dots, U_m$ such that

$$U_j = \text{proj}_{(i_{j-1}+1)..i_j}(X) \quad (4.1)$$

$$\dim(U_j) = i_j - i_{j-1}, \quad (4.2)$$

for each $j \in [1..m]$. Then we say that the U_j 's are **components** of X and write $X = U_1 :: U_2 :: \dots :: U_m$.

In general, there are many ways to decompose an FRP into components. However if $m = n$, *all* the components have dimension 1, and we call this collection

of FRPs the **scalar components** of X . The scalar components are unique, and in this special case, we write $X = \langle U_1, U_2, \dots, U_n \rangle$ to emphasize that the components are scalars.

That the two components of the system $M :: Y$ “do not interact” is an assumption about the structure of this particular system. It means that we can understand the combined system through a separate analysis of M and Y and that the output of one component does not give us any information about the other. This notion of “not interacting” is a fundamentally important one, *independence*. We will explore this in detail throughout the coming chapters, but here we start with the essential idea in informal terms. There are two perspectives on what independence means, both are equivalent and have nice expression in terms of the Big 3+1.

Definition 15. Independence (informal version).

1. The components that comprise a system are **independent** if we can fully analyze the system by combining the *separate* analyses of the components.
2. The components that comprise a system are **independent** if observing the output of any of the components *does not change* our knowledge or predictions about the output of the remaining components.

The Monty Hall game $G = M :: Y$ illustrates both these perspectives. M and Y are independent because you and Monty both make your door choices in isolation. Neither choice is influenced, implicitly or explicitly, by the other. For the first perspective, to analyze G – to express all our knowledge and predictions about its value through its Kind – we only need to analyze Monty’s choice and your choice separately. That is, we can find $\text{kind}(G)$ from $\text{kind}(M)$ and $\text{kind}(Y)$. For the second perspective, if I learned Monty’s choice, it would give me no information with which to better predict your choice, and vice versa.

In contrast, imagine a version of the game where Monty learned your choice *before* he made his choice and where the random mechanism by which he made his choice differed for each of the three choices you could make. You cannot analyze this game by considering only $\text{kind}(Y)$ and $\text{kind}(M)$, because the Kind of M in isolation can only describe Monty’s choice while accounting for your choice *in aggregate*. And knowing your choice changes our predictions about what Monty will do. In this game, the components are *not* independent. Example 4.13 shows this in detail.

Similarly, define the transformed⁸ FRP $D = M^2$ and consider the system with components $M :: M^2$. Without knowledge of M ’s value, my prediction of D ’s value

⁸Recall that the notation M^2 is a transform of M by an *inline* statistic, as described on page 131.

is uncertain, but if I *knew* M 's value, my prediction of D 's value would be perfect. Knowledge of M has changed my knowledge of D and thus my ability to predict D 's value, so M and D are not independent. And $\text{kind}(M :: M^2)$ cannot be expressed in terms of $\text{kind}(M)$ and $\text{kind}(M^2)$ alone because the two FRPs are coupled. Example 4.14 shows this in detail.

The two informal definitions of independence above have formal expressions in terms of the Big 3+1 that allow us to check whether the FRPs in any collection are independent. The first – that a system built from independent components can be analyzed by analyzing the components separately – is expressed in terms of joins. This leads to a special type of join, called an *independent join*, for explicitly combining independent components, the subject of this section. The second – that observing some components does not change our predictions about the others – is expressed in terms of observations and will be introduced in Section 5.5.

Fundamentally, independence is an *assumption* about the structure of the system, how the system's components relate, and the mechanisms that generate their values. In the Monty Hall example, we can take M and Y as independent because our description of the system specifies that those two choices are made in complete isolation. We can sometimes use data to check if an independence assumption is plausible, and we can sometimes make our analysis robust to a failure in that assumption. But in the end, it is part of our model of the system.

We take it as an axiom that we can always find independent sources of randomness. In particular, the FRP Marketplace can generate as many independent FRPs with the same Kind as we like. We call these *clones*.

Definition 16. Clones. FRPs are said to be **clones** if they are all independent and have the same Kind. Two clones, when fresh, are interchangeable with each other.

In `frplib`, we can create clones in several ways, including:

- for any FRP X , each call `clone(X)` returns a fresh clone of it; and
- for any fixed Kind k , each call `frp(k)` returns a fresh clone.

Puzzle 42. Define several FRPs as follows:

```
pgd> A = frp(uniform( (i, j) for i in irange(0, 5)
...>                               for j in irange(0, 5) ))
pgd> B = A ^ Proj[1]
pgd> C = A ^ Proj[2]
```

```

pgd> K = frp(constant(2))
pgd> U = A ~ Fork(Sum, Diff)
pgd> V = U ~ Proj[1]
pgd> W = U ~ Proj[2]
pgd> Z = frp(choice(0, 1))

```

Here, the `frplib` function `irange(s, t)` gives the inclusive range $[s..t]$. `A` has 36 possible values consisting of all points $\langle i, j \rangle$ in $[0..5] \times [0..5]$.

Using the informal definitions of independence above, especially the second, determine which of the following pairs of FRPs are independent. In each case, imagine that you know the value of one of the two FRPs (either will do fine), and check if that knowledge can change what you know (and thus can predict) about the other's value.

- (i) `A` and `B`
- (ii) `V` and `W`
- (iii) `C` and `V`
- (iv) `K` and any other FRP in the list
- (v) `Z` and any other FRP in the list

As a bonus, we can consider `B` and `C`, but that case is more subtle and we will revisit it in Chapter 5.

A solution to this puzzle is included at the end of the Chapter. You should make a strong effort at this before reading the solution.

A join that combine independent components of a system into a higher-dimensional FRP that concatenates all their values is called an **independent join**. An independent join is a special case of a join, not a distinct operation. Its importance lies what it tells us about the structure of the system and how we can analyze it. Analyzing a system, or part of a system, by analyzing its components separately is a powerful simplification that is worth highlighting.

The distinction between a general join and an independent join is based on an operation $\star$ on Kinds, defined later in this section (see equation ATTN). For the moment, we can ignore the details of this operation as our focus is on the *structure* of system implied by an independent join.

Definition 17. If an FRP $X :: Y$ with two components satisfies

$$\text{kind}(X :: Y) \cong \text{kind}(X) \star \text{kind}(Y), \quad (4.3)$$

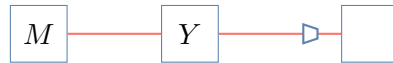
then the FRPs X and Y are independent and the original FRP is an **independent join**, written $X \star Y$.

Here, we write $\cong$ for Kind equivalence, but in practice we usually elide this distinction, write $=$, and identify equivalent Kinds. Equation (4.3) expresses the first definition of independence above. Our analysis of the combined FRP $X :: Y$ follows by combining our separate analyses of the components X and Y .

Like any join, an independent join decomposes a system into stages, but there is no contingency in the values produced at each stage. The use of an independent join is an assertion that a combined system is built from components that do not interact. For instance, the joined system $M \star Y$ for the Monty Hall game has dimension 2, with one slot for Monty's choice and one for yours. Any choice of the three doors is possible in each slot, so all nine joined values ($\langle 1, 1 \rangle$, $\langle 1, 2 \rangle$, $\langle 1, 3 \rangle$, $\langle 2, 1 \rangle$, $\langle 2, 2 \rangle$, $\langle 2, 3 \rangle$, $\langle 3, 1 \rangle$, $\langle 3, 2 \rangle$, $\langle 3, 3 \rangle$) are possible for the system as a whole. Monty makes a choice then you make a choice, each in isolation.

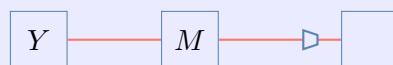
We will look at independent joins using all three interpretations discussed in subsection 1.5.2 (starting on page 71): physical, algebraic, and procedural.

The wiring diagram for the joined system embodies the physical interpretation:



M 's All output port is connected to Y 's input port, and Y 's All output port is connected to the input port of a new, empty FRP, which is automatically reconfigured to be the join FRP. When any of these FRPs are activated, so are the others. M 's value is transmitted to Y , and their combined value transmitted to the join FRP, becoming its value. (While activating any of the three FRPs activate the others, none of the FRPs move to the observed state except by pushing its own button. See Figure 1.2.) We do not really need the multiplexer here as there is only one input, but we keep it in the diagram to visually distinguish the join FRP. The key feature of this wiring diagram is that there is no contingency; the random mechanism determining Y is unaffected by the value of M . While we forward the value of M through the second stage, that value is not used at the second stage, except to pass through to join FRP.

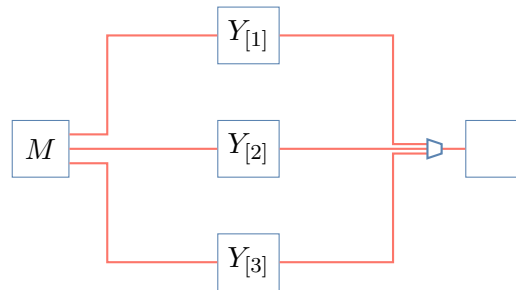
Puzzle 43. If we were to wire the system instead as



how would the resulting system differ? See Puzzle 46 below for a follow-up.

This wiring diagram for independent joins emphasizes the lack of contingency and requires a minimal number of FRPs. There is, however, an alternative wiring diagram that yields an equivalent join FRP and that shows how the independent join is a special case of the general join operation we saw earlier. We create a join in which the source is connected to multiple targets all of which are *clones* of each other.

Suppose that we have three clones of Y : $Y_{[1]}, Y_{[2]}, Y_{[3]}$, where $Y_{[i]}$ represents your choice of door when Monty has chosen door i . That these are clones tells us that your choice does not depend on Monty's choice; the same random mechanism determines your choice regardless of which of $Y_{[1]}, Y_{[2]}, Y_{[3]}$ is activated. If we connect the $\langle i \rangle$ output port of M , for each $i \in [1..3]$, to the input port of $Y_{[i]}$, we get the wiring diagram for a join FRP:



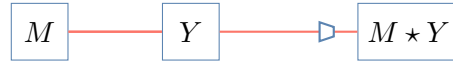
With this wiring, this is a general join, but it behaves equivalently to the independent join wiring above because all of the targets are interchangeable with Y . Whichever target activated will behave like Y because they are clones.

Both constructions of an independent join are equivalent, and we can use whichever is convenient at any point. We call the former the “flat” construction, where we connect the All port of the source to the input port of the target, and the latter the “clone” construction, where we connect every source output port to the input port of a different clone. The flat construction captures the key idea of an independent join: that independent components can be measured and analyzed separately. The clone construction shows that an independent join is just a special case of the general join operation.

Puzzle 44. Explain briefly in your own words why the flat and clone constructions are interchangeable for expressing the independent join operation.

For the algebraic interpretation of independent joins, we define an operator $\star$ that acts on two FRPs (or as we will see, two Kinds) and gives their independent join,

with the first as the source the second as the target. So, $M \star Y$ denotes the join FRP depicted by the above wiring diagrams. With labels, the wiring looks like



The possible values of $M \star Y$ are of the form $\langle m, y \rangle$ where $m, y \in [1..3]$. The first component of the tuple is Monty's door choice for the prize, and the second component is your initial door choice. The independent join operator $\star$ looks like a multiplication operator because, as we will see, it is a sort of "product." But it does *not* mean that the components are multiplied.

To construct the join $M \star Y$ for the Monty Hall game in `frplib`, we could use the code in the example module `frplib.examples.monty_hall`, but here we will build it explicitly. We create the Kinds of M and Y as described in Section 1.4. For Y , we start with an arbitrary Kind with values 1, 2, 3 and symbolic weights, but we substitute concrete weights (all equal here) for the symbols to create the FRP.

```
pgd> door_with_prize = uniform(1, 2, 3)
pgd> chosen_door = arbitrary(1, 2, 3, names=['l', 'm', 'r'])
pgd> chosen_door_concrete = substitution(chosen_door, l=1, m=1, r=1)
```

Next, we create the component FRPs from these Kinds with the `frp` function and form the independent join. The $\star$ operator is the `frplib` version of the independent join operator $\star$, and the join FRP is $M \star Y$.

```
pgd> M = frp(door_with_prize)
pgd> Y = frp(chosen_door_concrete)
pgd> M * Y
An FRP with value <3, 1>
pgd> M
An FRP with value <3>
pgd> Y
An FRP with value <1>
pgd> clone(M * Y)
An FRP with value <2, 2>
```

The value of $M \star Y$ combines the values of M and Y as shown. The `clone` function creates a clone of the given FRP, a fresh FRP with the same Kind.

Note that `frplib` requires that an independent join have independent components; otherwise, `frplib` raises an error. The reason for this is that we use independent joins

to assert that the components can be analyzed separately. Because the independent join is a special case of the general join operation, we can obtain the same combination for any FRPs, but we reserve the independent join specifically to highlight that special structure of the system.

```
pgd> M * M
This independent join could not be completed because its components
are not independent. Use a general join instead to create this FRP.
pgd> Dep_M = M ~ (__ ** 2) # Depends on M
pgd> M * Dep_M
This independent join could not be completed because its components
are not independent. Use a general join instead to create this FRP.
pgd> M * clone(M)
An FRP with value <2, 1>
```

Here, `Dep_M` and `M` are not independent. To see this, suppose we knew M 's value; then we would know `Dep_M`'s value as well and thus could *perfectly* predict it. Without that knowledge, our prediction would be more uncertain.

To get a sense of the possible values of $M * Y$, we can generate many clones and tabulate their values:

```
pgd> FRP.sample(10_000, M * Y)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| <1, 1> | 1104 | 11.04% |
| <1, 2> | 1113 | 11.13% |
| <1, 3> | 1127 | 11.27% |
| <2, 1> | 1121 | 11.21% |
| <2, 2> | 1101 | 11.01% |
| <2, 3> | 1088 | 10.88% |
| <3, 1> | 1089 | 10.89% |
| <3, 2> | 1124 | 11.24% |
| <3, 3> | 1133 | 11.33% |
+-----+-----+-----+
```

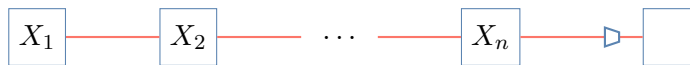
The sample includes all nine possible values, which occur with roughly the same frequency. From this, you can probably guess $\text{kind}(M * Y)$ for this case, and we will construct the Kind in detail a little later.

In general, if X and Y are independent FRPs with dimensions d_1 and d_2 and sizes s_1 and s_2 , then the FRP $X \star Y$ has dimension $d_1 + d_2$ and size $s_1 s_2$. This FRP merely collects the values of X and Y into a single value, with X 's value listed first and Y 's listed second. Every value of $X \star Y$ looks like $x :: y$ for a possible value x of X and a possible value y of Y .⁹ Each such tuple has d_1 components from x and d_2 components from y , yielding dimension $d_1 + d_2$. There are s_1 possible values of X and s_2 possible values of Y , and because this is an *independent* join – where the result of the second stage does not depend on the value of the first stage – any value of X can be paired with any possible value of Y , yielding $s_1 s_2$ possible values for $X \star Y$. In `frplib`, the join of X and Y is written analogously as $X \star Y$.

⁹The $::$ operator on tuples concatenates and flattens the tuples in order. See Section 15.4 in Interlude F.

As we have seen, we can wire joins of FRPs over any number of stages, and indeed the independent join operation can be applied to any number of independent FRPs. The $\star$ operator is *associative*:¹⁰ $(X \star Y) \star Z$ is the same FRP as $X \star (Y \star Z)$, so we can write this without parentheses as $X \star Y \star Z$ for any three (or more) FRPs. In general, $X_1 \star X_2 \star \dots \star X_n$ is an independent join of n stages. Its possible values have the form $x_1 :: x_2 :: \dots :: x_n$ for all possible values x_i of FRP X_i . Its wiring diagram looks like

¹⁰Associativity and commutativity of operations is discussed in detail in Chapter 17 of Interlude F.



Puzzle 45. If $X_1, X_2, \dots, X_n$ are FRPs where X_i has dimension d_i and size s_i for $i \in [1..n]$, what are the dimension and size of $X_1 \star X_2 \star \dots \star X_n$ and why?

Puzzle 46. The independent join operator is associative but *not* commutative, why? Give an example of two FRPs for which $X \star Y$ and $Y \star X$ are not the same.

If X and Y have dimensions d_1 and d_2 , there is an invertible statistic ψ so that $\psi(Y \star X)$ is equal to $X \star Y$ and $\psi^{-1}(X \star Y)$ is equal to $Y \star X$. What is that statistic? (For simplicity, start with the case where $d_1 = 1 = d_2$.)

Draw a wiring diagram for both $\psi(Y \star X)$ and $\psi^{-1}(X \star Y)$.

The procedural interpretation of the independent join is simple: we extract the value of M , we extract the value of Y , and we combine them to produce the value of

$M \star Y$. This FRP procedure looks like:

```
FRP  $M \star Y$ :
     $m \leftarrow M$ 
     $y \leftarrow Y$ 
    return  $m :: y$ 
```

Again, $m :: y$ makes a big tuple with the components of m and y , listing the former first. The independent join looks analogous for the independent join of any FRPs. For instance, for FRPs X_1 , X_2 , X_3 , and X_4 :

```
FRP  $X_1 \star X_2 \star X_3 \star X_4$ :
     $x_1 \leftarrow X_1$ 
     $x_2 \leftarrow X_2$ 
     $x_3 \leftarrow X_3$ 
     $x_4 \leftarrow X_4$ 
    return  $x_1 :: x_2 :: x_3 :: x_4$ 
```

We extract the value of each FRP in turn and join them. There is no interaction among them and no contingency in the construction of the returned value.

In `frplib`, we can write the independent join with procedures in almost the same way, with only minor syntactic substitutions. The head of the procedure is replaced by a decorated function definition (with no arguments), and the yield operator “ $\leftarrow$ ” is replaced by “`= yield`”.

```
@frp
def M_ijoin_Y():
    m = yield M
    y = yield Y
    return (m, y)
```

The `@frp` decorator on function definition with no parameters indicates that this is an FRP procedure that defines an FRP rather than a function. As such, this defines a new *FRP* `M_ijoin_Y`. Note the parallel structure with the procedure listed above. The `join` built-in `frplib` function¹¹ implements the `::` operation; we could return `join(m, y)` to explicitly concatenate and flatten the values `m` and `y`, but `frplib` does this for us automatically to make FRP procedures more convenient.

As an aside, we can define an FRP procedure for the clone construction of $M \star Y$:

¹¹In the `VecTuple` module but also loaded automatically in the playground

```
Y_1, Y_2, Y_3 = clone(Y), clone(Y), clone(Y)
```

```
@frp
def M_ijoin_Y_clone():
    m = yield M
    if m == 1:
        y = yield Y_1
    elif m == 2:
        y = yield Y_2
    else:
        y = yield Y_3
    return (m, y)
```

That Y_1 , Y_2 , and Y_3 are clones mean that in effect it does not matter from which we extract the value y .

Definition 18. If X and Y are independent FRPs, the FRP denoted by $X \star Y$ is called the **independent join** of X and Y . Its wiring diagram is



though it can be written equivalently as a general join.

If X has dimension d_1 and size s_1 and Y has dimension d_2 and size s_2 , then $X \star Y$ has dimension $d_1 + d_2$ and size $s_1 s_2$. The values of $X \star Y$ are all the values $x :: y$ where x is a value of X and y is a value of Y .

The independent join operator ($\star$, or $*$ in `frplib`) is associative.

The FRP procedure for this independent join is

```
FRP  $X \star Y$ :
```

```
 $x \leftarrow X$ 
```

```
 $y \leftarrow Y$ 
```

```
return  $x :: y$ 
```

which extracts the values of X and Y and joins them into a single tuple.

Example 4.1. In the classic game *Dungeons & Dragons*, each player has a character with six attributes (Strength, Intelligence, Wisdom, Constitution, Dexterity, Charisma) determined by an integer score in $[3..18]$. Each attribute's score is determined by rolling three six-side dice and summing their values.

Consider how we would represent any one attribute. We need three component FRPs that each represent the roll of a balanced, six-sided die. Call these D_1, D_2, D_3 . They are clones of each other, all having the same Kind

If D_1, D_2, D_3 are FRPs each representing a roll of a balanced six-sided die, the independent join $D_1 \star D_2 \star D_3$ represents the three rolls. Using an independent join means that the value of any individual roll does not influence the value any other roll. If we observed the first roll (D_1 's value), say, it would not help us predict the third roll (D_3 's value), and vice versa. Then, the FRP $\text{Sum}(D_1 \star D_2 \star D_3)$ represents the value of an attribute.

If S, I, W, C_o, D, C_h are independent FRPs that represent the six attributes' scores, the independent join

$$S \star I \star W \star C_o \star D \star C_h$$

represents a character. Using an independent join tells us that the processes generating the different attribute scores do not interact. That is, they are made with completely separate rolls. Whatever score we roll for Strength, for instance, does not influence (or help us predict) the score for Wisdom or Dexterity.

We can generate these in `frplib`. Rather than do this over and over, we will write FRP factories to create fresh FRPs on demand. These are just Python functions, which we can either enter directly in the playground or write in a separate Python source file. We apply the `frp_factory` decorator on these functions, which gives three benefits: its mark them with our intent, it gives nicer documentation, and it ensures that the returned value is converted to an FRP if possible. So, we can return an FRP or a Kind from the factory, and in the latter case, `frp` will be applied to the return value for us. This code is available in the module `frplib.examples.d_and_d`, which you can import into the `frplib` playground.

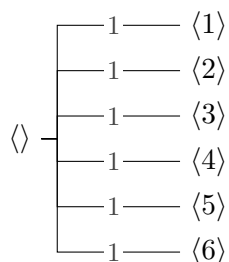
```
@frp_factory
def dice_roll():
    "the roll of a balanced 6-sided die"
    return uniform(1, 2, ..., 6)
```

```
@frp_factory
def dnd_attribute():
    "a score for a D&D character attribute"
    D_1, D_2, D_3 = dice_roll(), dice_roll(), dice_roll()
    return Sum(D_1 * D_2 * D_3)

@frp_factory
def dnd_character():
    "a D&D character's attribute scores"
    S = dnd_attribute()      # Strength
    I = dnd_attribute()      # Intelligence
    W = dnd_attribute()      # Wisdom
    Co = dnd_attribute()     # Constitution
    D = dnd_attribute()      # Dexterity
    Ch = dnd_attribute()     # Charisma

    return S * I * W * Co * D * Ch
```

The `dice_roll` factory uses the *Kind* factory `uniform` to produce the *Kind* of a single roll, which is converted to a fresh FRP by the factory. Note that the `...` in the call to `uniform` is intentional. It extends the pattern of the first two values up to and including the value after the `...`, so `uniform(1, 2, ..., 6)` is the *Kind*



In the factory `dnd_attribute`, `D_1`, `D_2`, and `D_3` represent independent rolls of balanced six-sided dice. The independent join `D_1 * D_2 * D_3` combines these into a single FRP that represents the collection of all three dice rolls. We transform this with the `Sum` statistic, producing an FRP that represents a random attribute score.

Similarly, `dnd_character` generates fresh, independent FRPs for each of the six attributes and uses an independent join to combine them into a single FRP of dimension 6. The components of this FRP are the generated attributes in the specified order.

We can use these in the playground if we typed the definitions in at the prompt or if we imported them from the example module, e.g.,

```
from frplib.examples.d_and_d import dice_roll, dnd_attribute, dnd_character
```

For example,

```
pgd> dnd_attribute()
An FRP with value <17>
pgd> dnd_character()
An FRP with value <13, 8, 9, 11, 8, 14>.
pgd> dnd_character()
An FRP with value <14, 7, 8, 11, 14, 16>.
```

If desired, we could easily define statistics that extract character's attribute scores by name.

```
# Statistics for each of the attributes by name
```

```
pgd> Strength = Proj[1]
pgd> Intelligence = Proj[2]
pgd> Wisdom = Proj[3]
pgd> Constitution = Proj[4]
pgd> Dexterity = Proj[5]
pgd> Charisma = Proj[6]
```

```
# Defining three characters
```

```
pgd> char1 = dnd_character()
pgd> char2 = dnd_character()
pgd> char3 = dnd_character()
```

```
# Applying the statistics
```

```
pgd> Strength(char1)
An FRP with value <9>
pgd> Wisdom(char2)
```

```
An FRP with value <16>
pgd> char3 ~ Fork(Intelligence, Charisma)
An FRP with value <14, 17>
```

Multi-stage joins like $D_1 \star D_2 \star D_3$ or $S \star I \star W \star D \star C_h \star C_o$, are well defined because the $\star$ operator is *associative*.¹² This means that for any independent FRPs X, Y, Z , the joins $(X \star Y) \star Z$ and $X \star (Y \star Z)$ are the same FRPs. Any way of grouping joins in a multi-stage join thus gives the same result, and we can drop the parentheses unambiguously.

¹²Associativity and commutativity of operations is discussed in detail in Chapter 17 of Interlude F.

Puzzle 47. In *Dungeons & Dragons*, a *saving throw* is a die roll made to determine if a player can resist some negative effect, like poison, a trap, or a spell. A saving throw is made by rolling a balanced 20-side die, adding an ability score, and checking if the total is $\geq$ a specified threshold determined by the nature of the threat. If so, the saving throw *succeeds*, and the character resists the threat. Otherwise, it *fails*, and bad things happen.

The ability score is based on the target character's attribute that is relevant to the particular threat (e.g., dexterity for dodging, constitution for resisting poison, charisma for resisting charm). If the relevant attribute is a , the ability score is $\lfloor \frac{a-10}{2} \rfloor$.

Write an FRP factory `saving_throw` that takes a character attribute FRP and a threshold and returns an FRP representing the event that the saving throw is successful (with 1 meaning success and 0 meaning failure).

This will look like the following, where the parameters to the function are annotated with the type for clarity. You will need to fill in the relevant objects wherever a `???` is shown.

```
@frp_factory
def saving_throw(character: FRP, attribute: Statistic, thresh: int):
    """the outcome of a saving throw for a specific character,
       attribute, and threshold
    """
    roll_d20 = frp(uniform( ??? ))
    ability = character ~ attribute ~ ???
    data = ??? * ???

    return data ~ ( ??? )
```

We are injecting randomness – the die roll – into the system represented by the character, and for this, we use an independent join. It is also necessary to transform two of the FRPs here to get them into the desired form.

We can use this as follows, referring to variables defined in Example 4.1.

```
pgd> saving_throw(char1, Constitution, 10)
An FRP with value <1>
pgd> saving_throw(char2, Dexterity, 13)
An FRP with value <0>
```

A solution to this puzzle is given at the end of the Chapter. But try this in the playground and experiment a bit before looking at the solution.

So far, we have been focusing on the structure of the system: the nature of the components, how they relate, and how they are combined. But to analyze such systems, we need more. We need to understand how to compute the *Kind* of an independent join from the Kinds of the components.

In the Monty Hall game discussed earlier, $\text{kind}(M \star Y)$ is shown in Figure 1.18. This comes from equation (4.3), which tells us that salientwe can find the Kind of an independent join from the Kinds of the constituent FRPs.

The independent-join operation on *Kinds*, denoted with the same operator $\star$, makes the following identity hold for independent FRPs:

$$\text{kind}(X \star Y) = \text{kind}(X) \star \text{kind}(Y). \quad (4.4)$$

In words: the Kind of an independent join FRP is the independent join of the two Kinds.

This identity is powerful: it tells us that in a system built from independent components, we can analyze the system by analyzing all of the components *separately*. The Kind embodies all of our knowledge about an FRP’s value, and it is the basis of all our analysis and predictions about the FRP. So, $\text{kind}(M \star Y)$ describes our knowledge about the system’s output, and the right-hand side of the equation tells us that we can fully describe the system’s output by separately analyzing each of the components and combining those analyses.

The $\star$ operation on Kinds is based on the wiring diagram for the “clone” construction in two steps:

1. attach a copy of the $\text{kind}(Y)$ tree to each leaf node in $\text{kind}(X)$
2. propagate the values at X ’s leaf nodes into the front of each tuple in the nodes below, to follow the Level and Prefix rules for Kinds.

Figure 4.8 illustrates this process for the Monty Hall example $M \star Y$. On the left side of the figure, we associate a copy of $\text{kind}(Y)$ at each leaf node of $\text{kind}(M)$. We join those trees, giving new leaf nodes for each copy of $\text{kind}(Y)$. We then rewrite the values of those leaf nodes to hold the sequence of values seen as we move from the root to that leaf. For example, top-most node is on the $\langle 1 \rangle$ branch for M and the $\langle 1 \rangle$ branch for Y ; its value becomes $\langle 1, 1 \rangle$. The fourth leaf node down is on the $\langle 2 \rangle$ branch for M and the $\langle 1 \rangle$ branch for Y ; its value becomes $\langle 2, 1 \rangle$. The eighth leaf node down is on the $\langle 3 \rangle$ branch for M and the $\langle 2 \rangle$ branch for Y ; its value becomes $\langle 3, 2 \rangle$. And so on.

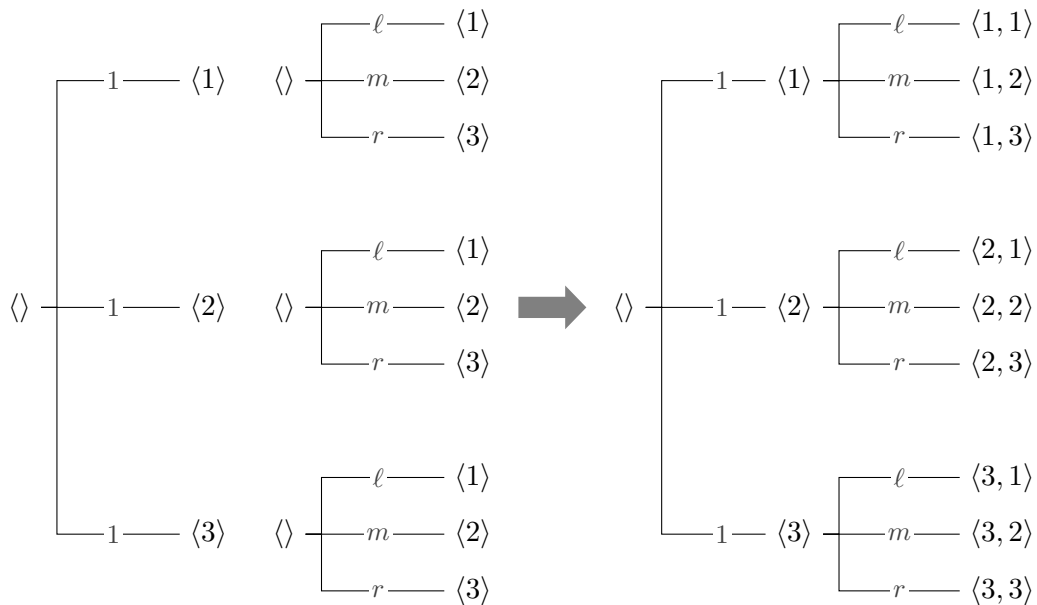


FIGURE 4.8. Constructing the Kind join $\text{kind}(M) \star \text{kind}(Y)$. For each leaf of $\text{kind}(M)$, we take a copy of $\text{kind}(Y)$ and attach it, forming the Kind tree on the right.

Puzzle 48. Convince yourself that with this definition of independent join of Kinds, $\text{kind}(M \star Y) = \text{kind}(M) \star \text{kind}(Y)$. Notice that after M generates a value, the next stage looks the same no matter what that value is.

We can examine these Kinds in the `frplib`. Using the Kinds `door_with_prize` and `chosen_door` defined earlier (and in `frplib.examples.monty_hall`) and the FRPs M and Y , do

```
pgd> door_with_prize
pgd> chosen_door
```



```

pgd> outcome = door_with_prize * chosen_door
pgd> outcome

pgd> outcome1 = substitution(outcome, l=1, m=1, r=1)
pgd> outcome1
pgd> kind(M * Y)
pgd> Kind.equal(outcome1, kind(M * Y))

pgd> unfolded(substitution(door_with_prize * chosen_door, l=1, m=1, r=1))
pgd> unfolded(kind(M * Y))

```

The output is omitted here, but it should show the corresponding Kinds equal and match the Figures up to the scaling of the weights. Try it yourself.

Definition 19. The Independent Join of Kinds.

If k_1 and k_2 are Kinds, their **is** a Kind $k_1 \star k_2$, defined by the procedure:

1. attach a copy of k_2 to each leaf node of k_1 , and
2. replace each leaf node of the combined tree from step 1 with a concatenation of the tuples on the path from the root to the leaf.

Equivalently, in terms of Kind procedures we have

Kind $k_1 \star k_2$:

$x \leftarrow k_1$

$y \leftarrow k_2$

return $x :: y$

Over all possible values from k_1 and all possible values from k_2 , we join the values and account properly for the weights.

As with FRPs, we take independent joins of Kinds over any number of stages, e.g., $k_1 \star k_2 \star \dots \star k_n$. Again $\star$ is an *associative* operation, so $(k_1 \star k_2) \star k_3 = k_1 \star (k_2 \star k_3)$, and we can write both without parentheses as $k_1 \star k_2 \star k_3$.

The independent join operations satisfy a key identity: a system built as the join of independent components can be analyzed by analyzing the components separately and combining those analyses.

For independent FRPs $X_1, X_2, \dots, X_n$,

$$\text{kind}(X_1 \star X_2 \star \dots \star X_n) = \text{kind}(X_1) \star \text{kind}(X_2) \star \dots \star \text{kind}(X_n). \quad (4.5)$$

Thus, Kinds and FRPs combine in parallel ways, and the Kinds of independent joins are thus completely determined by the Kinds of their constituent FRPs. If a system is built with independent components, our knowledge of the system's output can be described by combining our knowledge about each component separately.

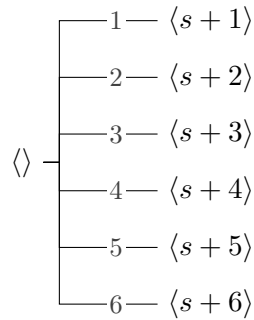


FIGURE 4.9. The Kind for the choice of player matchups in Example 4.2, with $s = 0$ for the first player and $s = 6$ for the second player.

Example 4.2. In a round-robin tournament, twelve players are ranked from 1 to 12, with lower rank for stronger players. The first match pits two players against each other, with one chosen at random from the ranks 1–6 and the other chosen at random from ranks 7–12. The two choices are made independently. Assume that players are weighted by rank in this choice, with less weight on stronger players, as shown in Figure 4.9. Specifically, the Figure shows the Kind for an FRP representing the rank of the chosen player, with $s = 0$ corresponding to the choice of first player (ranks 1 through 6) and $s = 6$ corresponding to the choice of the second player (ranks 7 through 12). Let F be the FRP representing the pair of players engaged in the first match. We want to find $\text{kind}(F)$.

Let `first_player` and `second_player` be the Kinds for the first and second player chosen, from Figure 4.9.

```
pgd> first_player = weighted_by(1, 2, ..., 6, weight_by=scalar_fn(Id))
pgd> second_player = weighted_by(7, 8, ..., 12, weight_by=scalar_fn(Id - 6))
pgd> first_player
pgd> second_player
```

The Kind factory `weighted_by` returns a Kind with the given values where the weights are computed by a function of the value. Here, `scalar_fn` converts a scalar statistic into a simple function. Look at these Kinds and compare to the display above.

We have that `F = frp(first_player * second_player)` and `kind(F)` equals `first_player * second_player`. Before we compute this Kind in the playground, it is worth seeing how we would compute it by hand.

Let's consider the leaf node of the independent join Kind with value $\langle 3, 10 \rangle$. The $\langle 3 \rangle$ node in `first_player` has weight $\frac{3}{21}$, and the $\langle 10 \rangle$ node in `second_player` has weight $\frac{4}{21}$. So the canonical weight on node $\langle 3, 10 \rangle$ is $\frac{4}{147} \approx 0.027211$. Similarly, for the leaf node $\langle 6, 12 \rangle$, the `first_player` weight for $\langle 6 \rangle$ is $\frac{6}{21}$ as is the `second_player` weight for $\langle 12 \rangle$. The weight on the leaf node $\langle 6, 12 \rangle$ is $\frac{4}{49} \approx 0.081633$.

With this in mind, we can look at the Kind, in both unfolded and canonical form. The output is omitted here, but see Figure 4.10 for the unfolded form.

```
pgd> match_up = first_player * second_player
pgd> unfolded(match_up)
pgd> match_up
```

Notice that

```
pgd> Kind.equal(Proj[1](match_up), first_player)
True
pgd> Kind.equal(Proj[2](match_up), second_player)
True
```

So the Kind `match_up` combines the component Kinds as expected.

Puzzle 49. Convince yourself that the “flat” and “clone” constructions of $M \star Y$ give FRPs with the same Kind.

Puzzle 50. In Example 4.1, sketch the Kind of

```
dice_roll() * dice_roll() * dice_roll()
```

by hand. How would you find the Kind of `dnd_attribute()` by hand?

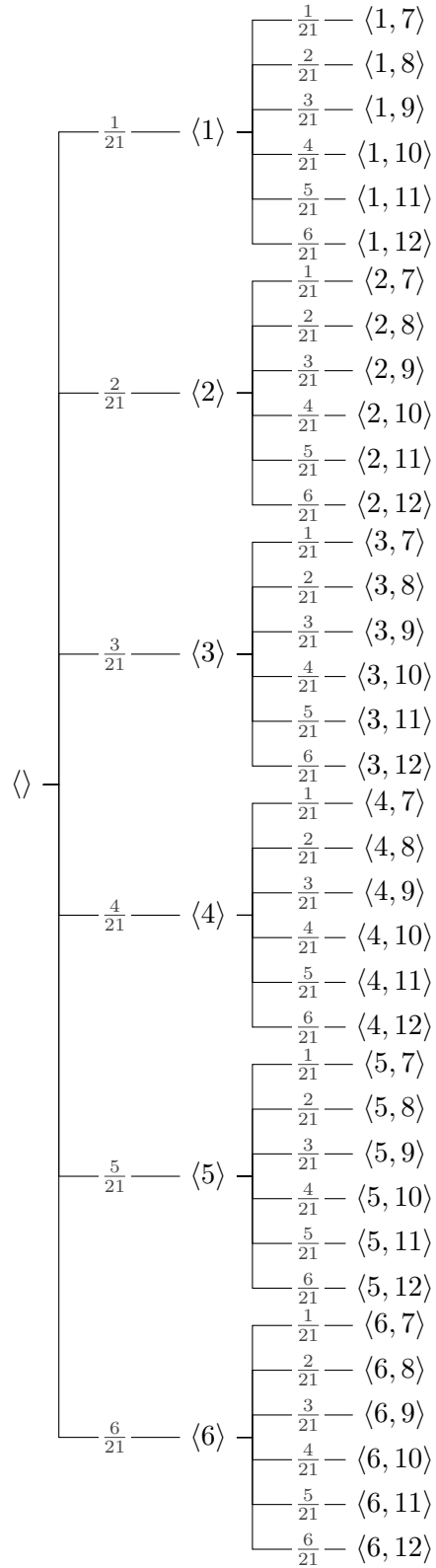


FIGURE 4.10. Unfolded Kind of the first tournament matchup in Example 4.2.

Find both these Kinds in the playground to check your work.

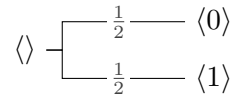
The word **independent** in “independent join” has meaning. It tells us that the distinct stages of the process represented by the FRPs in the join evolve without interaction or influence on each other. So if you observe the value of some FRPs in the join, that information *does not help you predict the value of any other FRPs* in the join. Independence is at the heart of equation (4.5). It is independence that gives the “flat” and “clone” constructions equivalent outputs. When the parts of a system are independent, we can analyze the whole system by analyzing the parts separately. Lack of independence – *dependence* – means that knowing the values of some FRPs in the join *changes our predictions* of the other FRPs’ values. A system with dependent parts is coupled and cannot be as easily decomposed. Independence, when we have it, is powerful.

Figure 4.10 from the previous example illustrates these ideas. Without any information, what can we say about the second player in the match-up? Our knowledge (and all the predictions we might make about the second player) is embodied in the Kind `second_player`, which is just `Proj [2] (match_up)`. Now suppose I tell you (truthfully) that I have activated and observed the display of the first player rank FRP and its value is $\langle 4 \rangle$. How does this change your knowledge of the second player? Picture yourself at the $\langle 4 \rangle$ node in the Kind tree, having just obtained the information about the first player. What will happen in the next stage is represented by the subtree at that node. The *definition* of the independent join of Kinds means that for any value $\langle v \rangle$ of the first player FRP, the Kind we see *looking down the tree from the $\langle v \rangle$ node* is essentially¹³ a copy of `second_player`. The knowledge of the first player gives us no useful information about the second player. Similarly, if I observe that the second player is ranked 9 and tell you this. You know that you must be at one of the leaf nodes that look like $\langle \blacksquare, 9 \rangle$; you do not know which one. But there is only one such node at each subtree, so looking up the tree from these nodes, we see a copy of `first_player`. The knowledge of the second player thus gives us no useful information about the first player. The FRPs representing choices of the two players are independent.

The choice to use an independent join when building a system is an assumption that the constituent FRPs are independent in this sense. The next example shows this idea as well, and we will return formally to the idea of independence when we discuss observations in Chapter 5. When there is dependence between stages, we use general joins instead (Section 4.4).

¹³The only difference is that the tuple $\langle v \rangle$ is prepended to every node in the tree, but this does not change the point.

Example 4.3. We want to build an FRP representing three flips of a coin where heads and tails are equally likely and each flip is independent of the other flips. The result will be an independent join $F = F_1 \star F_2 \star F_3$ where the FRPs F_i all have Kind



where we use 0 to represent tails and 1 to represent heads.

In `frplib`, this becomes

```
pgd> flip = choice(0, 1)
pgd> F_1, F_2, F_3 = frp(flip), frp(flip), frp(flip)
pgd> F = F_1 * F_2 * F_3
pgd> F
An FRP with value <1, 0, 1>
pgd> flips = flip * flip * flip
pgd> Kind.equal( kind(F), flips )
True
```

Look at the Kind `unfolded(flips)` in the playground. (Really!) Imagine that you have learned the value of F_1 and want to assess what this information tells you about F_2 and F_3 . If move in the kind tree to the node representing the observed value of F_1 , what you see looking down the tree is the Kind of the other two flips, but for either value of F_1 , you see the same tree: `flip * flip`. Your predictions about F_2 and F_3 have not changed. If instead you have learned the value v of F_3 , then you know that you are at a node for which F_3 takes that value (i.e., $\langle \blacksquare, \blacksquare, v \rangle$), but you cannot tell which one. Looking up from all such nodes gives you a tree that again is just `flip * flip`, and again your predictions of the other flips have not changed. Although it looks more complicated, the same story holds if you learn the value of F_2 . This puts you at a node like $\langle \blacksquare, v, \blacksquare \rangle$ and what you can see from all such nodes is again just the tree `flip * flip`.

Another way to see the latter is to transform the Kind with a permutation so that the third flip is listed first:

```
pgd> flips_p = flips ^ Permute(2,1)
```

Now, if we have observed F_3 , our analysis is like that of F_1 earlier, looking down the tree from the nodes at the first level. In fact, for any permutation of the

three components, the Kind transformed by the permutation is equal to **flips**.

The above analysis works with various Kinds of information. For instance, observing the values of F_1 and F_2 , puts us at a node in the second level, and looking down the tree, we see **flip**.

Puzzle 51. If k_1 and k_2 are Kinds, is $k_1 \star k_2$ the same as $k_2 \star k_1$? If so, how do you know? If not, how are they related?

INDEPENDENT JOINS IN TERMS OF KERNELS. For Kinds $k_1, k_2, \dots, k_m$ of dimensions $d_1, d_2, \dots, d_m$, their independent join $k = k_1 \star k_2 \star \dots \star k_m$ is Kind of dimension $d_1 + d_2 + \dots + d_m$ with possible values $v_1 :: v_2 :: \dots :: v_m$ over all possible values v_i of k_i for $i \in [1..m]$.

Imagine that the Kinds $k_1, k_2, \dots, k_m$ are in canonical form, and consider one possible value $v_1 :: v_2 :: \dots :: v_m$ at a leaf node of the independent join. The independent join operation on Kinds builds k by successively attaching copies of the Kind tree k_i to the tree built from $k_1, \dots, k_{i-1}$. The value at a leaf node thus describes a *path through the tree k* from root to leaf: take the v_1 branch at the root, then from that subtree take the v_2 branch, then from that subtree take the v_3 branch, and so on until taking the v_m branch at the final subtree. See Figure 4.11 for an illustration. The edges along that path are the canonical weights of v_i in Kind k_i for each i . From ALGORITHM CANONICAL in Chapter 3, **the weight on $v_1 :: v_2 :: \dots :: v_m$ in the canonical form of k is the product of the weights along the path from root to leaf.**

We can express this in terms of the *kernels* of the Kinds involved, whether or not the Kinds are in canonical form, because the kernels give the canonical weights for a given value, or 0 if that value is not in the tree.

The kernel of an independent join of Kinds $k_1, k_2, \dots, k_m$ is given by:

$$(k_1 \star k_2 \star \dots \star k_m)(v_1 :: v_2 :: \dots :: v_m) = \prod_{i=1}^m k_i(v_i). \quad (4.6)$$

In terms of the tensor product¹⁴ described in Section 15.5 of Interlude F, this means that the kernels of the Kind $k_1 \star k_2 \star \dots \star k_m$ is the tensor product function $k_1 \otimes k_2 \otimes \dots \otimes k_m$. Written in terms of independent FRPs $X_1, X_2, \dots, X_m$ this

¹⁴ $(f \otimes g)(u :: v) = f(u)g(v)$, where u and v match the codimensions of f and g .

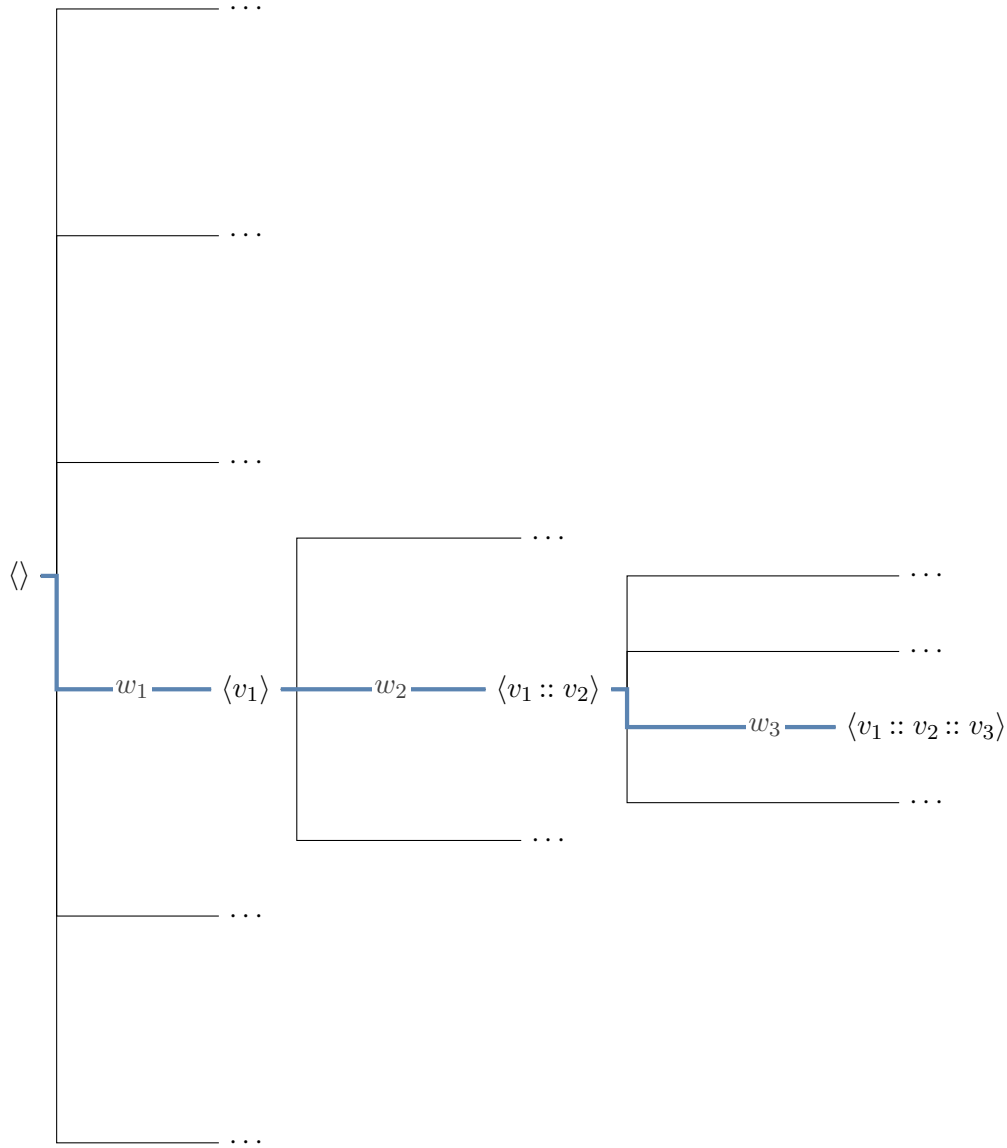


FIGURE 4.11. An illustration of the path through an independent join Kind tree to reach value $v_1 :: v_2 :: v_3$. The canonical weight for this value is the product along the path $w_1 w_2 w_3$.

becomes

$$k_{X_1 \star X_2 \star \dots \star X_m} = k_{X_1} \otimes k_{X_2} \otimes \dots \otimes k_{X_m}. \quad (4.7)$$

Aside: The Algebra of Independent Joins. The independent join operation $\star$ has a few notable properties. First, let X be an FRP. Recall that **empty** is the trivial FRP with Kind $\langle \rangle$, just a root node with an empty list.

If we connect the All output port of **empty** to the input port of X , then when we push the button on **empty**, we get just the output of X because the empty list does not add anything to the value. So, $\mathbf{empty} \star X$ is the same as X .

Similarly if we connect the All output port of X to **empty** (or the individual output ports of $\text{size}(X)$ copies of **empty**), the result is again the same as X . So, $X \star \mathbf{empty}$ is the same as X . That is:

$$X \star \mathbf{empty} = X = \mathbf{empty} \star X.$$

This applies to the Kinds as well:

$$\text{kind}(X) \star \langle \rangle = \text{kind}(X) = \langle \rangle \star \text{kind}(X),$$

where $\langle \rangle$ denotes the empty Kind ($\text{kind}(\mathbf{empty})$). For FRPs and Kinds, respectively, **empty** and $\langle \rangle$ are “identity elements” for the independent join operation.

Second, with three FRPs X , Y , and Z , we can take the independent join of the three in two different ways: $X \star (Y \star Z)$ or $(X \star Y) \star Z$. As we are just connecting output and input ports, it does not matter which pair we join first, and similarly with Kinds:

$$\begin{aligned} X \star (Y \star Z) &= (X \star Y) \star Z \\ \text{kind}(X) \star (\text{kind}(Y) \star \text{kind}(Z)) &= (\text{kind}(X) \star \text{kind}(Y)) \star \text{kind}(Z). \end{aligned}$$

Thus, $\star$ is “associative” for FRPs and Kinds, and we can write the join without parentheses, $X \star Y \star Z$ and $\text{kind}(X) \star \text{kind}(Y) \star \text{kind}(Z)$.

Note that the $\star$ operator is not *quite* commutative, but it is close in an important sense. $X \star Y$ and $Y \star X$ are not equal in general because they combine their constituent values in different orders within the value, and similarly for Kinds $k_1 \star k_2$ and $k_2 \star k_1$. However, we can transform $X \star Y$ to $Y \star X$ and vice versa by permuting the tuples, so they have effectively the same information. The

See Chapters 13 and 14 in Interlude F for details on invertible functions.

same goes for Kinds $k_1 \star k_2$ and $k_2 \star k_1$. Formally, this transform is an invertible permutation statistic ψ such that $Y \star X = \psi(X \star Y)$ and $X \star Y = \psi^{-1}(Y \star X)$ and $k_2 \star k_1 = \psi(k_1 \star k_2)$ and $k_1 \star k_2 = \psi^{-1}(k_2 \star k_1)$. Thus, the two FRPs $X \star Y$ and $Y \star X$ and the two Kinds $k_1 \star k_2$ and $k_2 \star k_1$ are the same up to ordering of the components. While not *equal*, we say that they are *isomorphic*, which is the next best thing.

INDEPENDENT JOIN POWERS. A common pattern is to build an independent join from multiple clones of an FRP or from the same Kind used multiple times. We saw this in Example 4.1 with

```
D_1 * D_2 * D_3
dice_roll() * dice_roll() * dice_roll(),
```

the first giving an FRP for the roll of three dice and the second giving its Kind.

The independent join acts like a type of “product,” so this repeated product looks and acts like a “power.” This pattern is so common, we have a shorthand for it that evokes that connection.

Definition 20. Independent Join Powers. For any Kind k and any natural number m , we define

$$k \star\star m = \overbrace{k \star \cdots \star k}^{m \text{ times}}, \quad (4.8)$$

where $k \star\star 0 = \langle \rangle$, the empty Kind.

For any FRP X and any natural number m , we define

$$X \star\star m = \overbrace{\text{clone}(X) \star \text{clone}(X) \star \cdots \star \text{clone}(X)}^{m \text{ times}}, \quad (4.9)$$

where $X \star\star 0 = \text{empty}$. The clones are here to make the shorthand more useful; simply repeating the exact value of X is not what we want. This ensures that $X \star\star m$ is an independent join of m FRPs with the same Kind as X .

In `frplib`, we use the `**` operator for this, so these joins look like `X ** m` and `k ** m`.

The operators $\star\star$ and `**` are reminiscent of the operator for powers in several programming language, which is not a coincidence. Indeed, $k \star\star m$ and $X \star\star m$ are essentially *powers* of the independent join operator.

The set of possible values for an independent join m th-power contains all m -tuples of possible values of the component Kind or FRP. Each such tuple specifies a unique path down the Kind tree from root to leaf. Taking one component at a time, we follow the branch for that component and at the end, we are the bottom of the tree. If k is in canonical form, then the canonical form of an independent join power $k \star \star m$ assigns weight to a value $v_1 :: v_2 :: \dots :: v_m$ that is **the product of the canonical weights for k along the path in the tree from root to leaf**. Written in terms of kernels, we have

$$(k \star \star m)(v_1 :: v_2 :: \dots :: v_m) = \prod_{i=1}^m k(v_i), \quad (4.10)$$

following equation (4.6). Written in terms of an FRP X this becomes

$$k_{X \star \star m}(v_1 :: v_2 :: \dots :: v_m) = \prod_{i=1}^m k_X(v_i), \quad (4.11)$$

Example 4.4 Gambler's Fortune

A gambler places a series of identical bets of \$1 on a casino game, e.g., roulette (Examples 2.1, 2.2, 2.5). Let $B_1, B_2, \dots, B_n$ be the FRPs representing the outcomes of the individual bets. We assume that all bets are independent and have Kind

$$\langle \rangle \begin{cases} \text{---} 1-p \text{---} \langle -1 \rangle \\ \text{---} p \text{---} \langle 1 \rangle \end{cases}$$

For concreteness, we will use $p = 4/9$ in this example, putting a lower weight on winning than losing. We might call this an “unfair” game.

```
pgd> p = symbol('p')
pgd> bet_kind = weighted_as(-1, 1, weights=[1 - p, p])
pgd> bet_kind_c = substitution(bet_kind, p=as_quantity('4/9'))
pgd> bet_kind ** 3
,---- 1 + -3 p + 3 p^2 + -1 p^3 ---- <-1,-1,-1>
|---- p + -2 p^2 + p^3 ----- <-1,-1, 1>
|---- p + -2 p^2 + p^3 ----- <-1, 1,-1>
|---- p^2 + -1 p^3 ----- <-1, 1, 1>
```

```

<> -|
|---- p + -2 p^2 + p^3 ----- <1,-1,-1>
|---- p^2 + -1 p^3 ----- <1,-1, 1>
|---- p^2 + -1 p^3 ----- <1, 1,-1>
`---- p^3 ----- <1, 1, 1>
pgd> bet_kind_c ** 3
,---- 0.17147 ----- <-1,-1,-1>
|---- 0.13717 ----- <-1,-1, 1>
|---- 0.13717 ----- <-1, 1,-1>
|---- 0.10974 ----- <-1, 1, 1>
<> -|
|---- 0.13717 ----- <1,-1,-1>
|---- 0.10974 ----- <1,-1, 1>
|---- 0.10974 ----- <1, 1,-1>
`---- 0.087791 ----- <1, 1, 1>
pgd> B_c = frp(bet_kind_c)
An FRP with value <1>
pgd> B_c ** 10
An FRP with value <1, 1, -1, -1, -1, -1, -1, -1, -1, -1>

```

Here, `bet_kind ** 3` is the Kind for a series of three bets with general p , and `B_c ** 10` is an FRP representing a series of 10 bets with the concrete setting $p = 4/9$. Notice that `bet_kind ** n` has size 2^n which grows quickly with n . Look at the Kind for `bet_kind ** 5` up to, say, `bet_kind ** 10`; beyond 20 or so, the Kind tree gets rather overwhelming and slow to compute.

However, as usual, we are in practice less interested in the sequence of bets itself and more interested in statistics that answer questions about that sequence. For instance, what is the gambler's net gain or loss? Or: does the gambler end up ahead or behind? We can define statistics to embody these questions:

```

pgd> net_gain = Sum
pgd> end_ahead = (Sum >= 0)
pgd> net_gain(B_c ** 10)
An FRP with value <-6>
pgd> end_ahead(bet_kind_c ** 5)
,---- 0.60331 ----- 0
<> -|

```

```
`---- 0.39669 ---- 1
```

For large n , a direct computation of the Kind `bet_kind_c ** n` will be slow because the Kind has size that is exponential in n . But `frplib` has a few tricks that can make this efficient in some cases. (Details are in Section 7.1; see `fast_join_pow` in the `frplib` documentation.) In particular, the following are quite fast:

```
pgd> ng_100 = kind(net_gain(B_c ** 100))
pgd> ea_100 = ng_100 ^ (__ >= 0)
```

The Kind `ng_100` equals `net_gain(bet_kind_c ** 100)` and `ea_100` equals `end_ahead(bet_kind_c ** 100)`, but both are computable very quickly.

With this in hand, we are in a position to answer some interesting questions. If you could choose n , what should you choose? What do we predict for your winnings and for whether you will at least break even?

So, suppose you have $\$w$ in your wallet and will place n equal-sized bets of $\$w/n$, on a game like the above. First, we will write a function `winnings` to do our analysis. It will take an FRP (like `B_c`) that represents a single \$1 bet along with n and w and *return the Kind of the FRP that represents our net winnings after n bets*. The statistic `in_the_black` takes the winnings and indicates whether we at least broken even, which we document below.

```
def winnings(BaseBet, n, w=1):
    """Returns the Kind of an FRP for net winnings after n bets of $w/n.

    BaseBet is an FRP representing the change in the gambler's wealth from
    a single $1 bet on the game.

    """

    stakes = w / n
    Bet = BaseBet ^ (__ * stakes)
    return kind(Sum(Bet ** n))      # or fast_join_pow(Sum, kind(Bet), n)

in_the_black = (__ >= 0)
in_the_black.doc = 'asks: given our winnings, have we at least broken even?'
```

Now, we can look at some results assuming an initial wealth of \$4200. We loop over selected values of n , compute the two Kinds, and store them. This may take a moment to run, though we could make this much faster with a little more effort.

```
pgd> win_kinds = {} # Dictionary keyed by n
pgd> num_bets = [*irange(2, 16, step=2), *irange(20,100,step=5),
...>             *irange(110, 500, step=10)]
pgd> for n in num_bets:
    win_kinds[n - 1] = winnings(B_c, n - 1, 4200)
    win_kinds[n] = winnings(B_c, n, 4200)
pgd> in_the_black(win_kinds[1])
    ,---- 5/9 ---- 0
    <> -|
    `---- 4/9 ---- 1
pgd> in_the_black(win_kinds[2])
    ,---- 0.30864 ---- 0
    <> -|
    `---- 0.69136 ---- 1
pgd> in_the_black(win_kinds[10])
    ,---- 0.51886 ---- 0
    <> -|
    `---- 0.48114 ---- 1
pgd> in_the_black(win_kinds[100])
    ,---- 0.84550 ---- 0
    <> -|
    `---- 0.15450 ---- 1
pgd> in_the_black(win_kinds[500])
    ,---- 0.99283 ----- 0
    <> -|
    `---- 0.0071681 ---- 1
```

We expect there to be a difference between even and odd n here because with even n , there is an extra way to break even. We collect these results using `E` to compute our predictions about winnings and breaking even:

```
pgd> win_pred = { n : as_float(E(k)) for n, k in win_kinds.items() }
```

```
pgd> break_even = { n : as_float(E(in_the_black(k)))
...>               for n, k in win_kinds.items() }
pgd> break_even[0] = 1
```

We use the `as_float` function to convert the expectations to floating point numbers from the high-precision decimals used by `frplib` natively.

Looking at `win_pred`, which gives our predicted winnings for each n , we see that all the values are the same $-466\frac{2}{3}$. This happens to be $4200 \cdot -\frac{1}{9}$. The game B_c is after all against us in that we are more likely to lose than win, and note that $E(B_c)$ is $-1/9$ as you can confirm in the playground. Our best prediction is that we will lose on average $-\frac{1}{9} \cdot \frac{w}{n}$ per play.

On the other hand, `break_even` shows a pattern plotted in Figure 4.12. Even and odd n are indeed different as predicted, and within each group, our chance of breaking even is *strictly decreasing*. (Notice that the Kind

`in_the_black(winnings(B, n, w))`

does not depend on w at all.) In an “unfair” game like B_c , our best choice is not to play, and if we play, we minimize our chance of an overall loss by “bold” play – putting it all on a few bets.

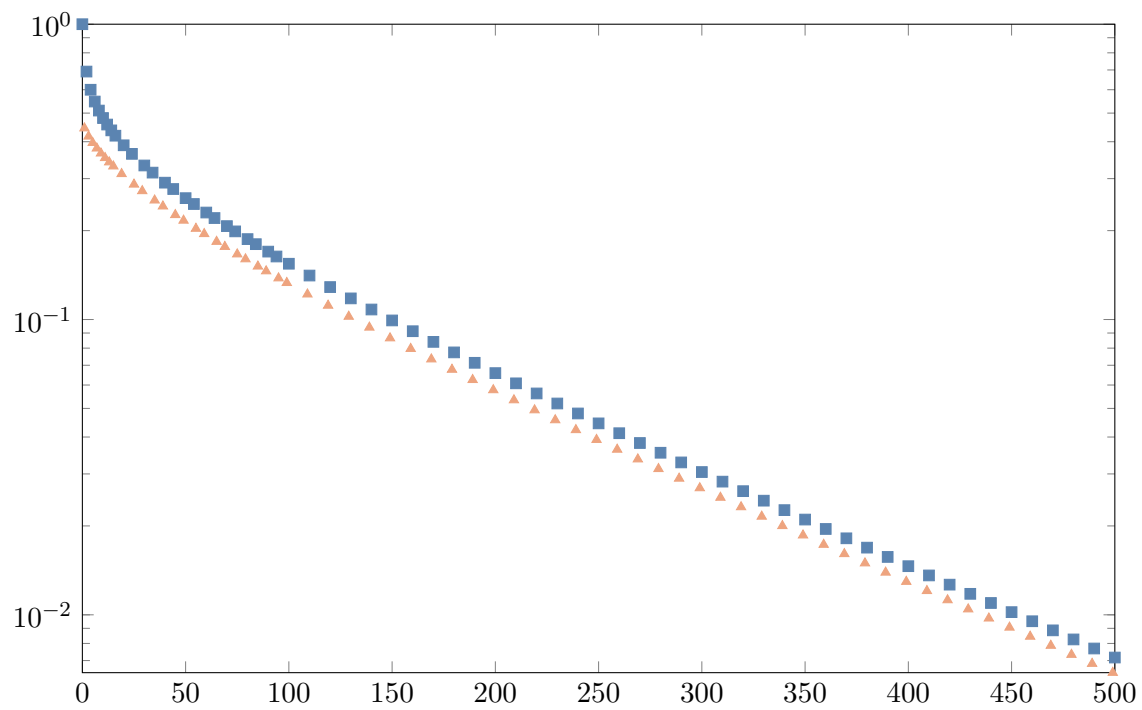


FIGURE 4.12. Predicted chance to break even in n bets in Example 4.4. Results for even n are plotted as squares and for odd n triangles.

Key Take Aways

An FRP representing the output of a random system can be decomposed into **components** that correspond to disjoint segments of the FRPs value. We write $X = U_1 :: U_2 :: \dots :: U_m$ to show a decomposition of an FRP X of dimension n into m components. (When the U_i 's all have dimension 1, these are the **scalar components** of X , and we write $X = \langle U_1, U_2, \dots, U_n \rangle$.)

The components that comprise a system are **independent** if

1. we can fully analyze the system by analyzing the components *separately*, or
2. observing the output of any of the components *does not change* our knowledge or predictions about the output of the remaining components.

These two conditions are equivalent. Two FRPs are **clones** if they are independent and have the same Kind.

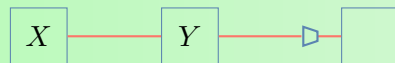
If FRP $X :: Y$ satisfies

$$\text{kind}(X :: Y) = \text{kind}(X) \star \text{kind}(Y), \quad (4.12)$$

where $\star$ is the independent join operation on Kinds, then we say that the FRP is an **independent join**, written $X \star Y$. This implies that X and Y are **independent**.

An **independent join** is the special case of a join in which we combine independent components. As with a general join, we connect the output of one stage into the input of the next, but with independent joins, those connections serve merely to forward the values of all the components into the final join FRP. The individual components evolve independently.

The physical interpretation of an independent join is the wiring diagram:



The algebraic interpretation defines an associative operator $\star$ ($\star$ in `frplib`) with $X \star Y$ the independent join of X and Y . The possible values of $X \star Y$ have the form $x :: y$ over all possible values x of X and y of Y . The procedural interpretation extracts the value from each component and joins them into a larger tuple in order:

FRP $X \star Y$:

$x \leftarrow X$

$y \leftarrow Y$

return $x :: y$

The independent join of Kinds is analogous. We form the tree $k_1 \star k_2$ by attaching a copy of k_2 to each leaf node of k_1 and then propagating the values of those leaf nodes down the tree, prepending them to every lower node. The $\star$ operator is associative on Kinds and $k \star \langle \rangle = k = \langle \rangle \star k$.

The procedural interpretation of independent joins for Kinds parallels that for FRPs. We select in parallel every possible value from k_1 and pair that with every possible value from k_2 , joining the values and multiplying the weights. We have

Kind $k_1 \star k_2$:

$x \leftarrow k_1$

$y \leftarrow k_2$

return $x :: y$

For independent FRPs $X_1, X_2, \dots, X_n$,

$$\text{kind}(X_1 \star X_2 \star \dots \star X_n) = \text{kind}(X_1) \star \text{kind}(X_2) \star \dots \star \text{kind}(X_n). \quad (4.13)$$

Thus, a system built with independent components can be analyzed by analyzing the components separately.

In terms of kernels, we get the canonical weights of a value in an independent join by taking the product of canonical weights along each path from root to leaf:

$$(k_1 \star k_2 \star \dots \star k_m)(v_1 :: v_2 :: \dots :: v_m) = \prod_{i=1}^m k_i(v_i). \quad (4.14)$$

A common pattern is the independent join of a Kind with itself multiple times or with multiple clones of the same FRP. This acts like a “power” operation,

which gives us the **independent join power**. For Kind k and FRP X :

$$k \star\star m = \overbrace{k \star \cdots \star k}^{m \text{ times}}$$

$$X \star\star m = \overbrace{\text{clone}(X) \star \text{clone}(X) \star \cdots \star \text{clone}(X)}^{m \text{ times}},$$

with $k \star\star 0 = \langle \rangle$ and $X \star\star 0 = \text{empty}$. In `frplib`, this operator is `**`.

The possible values of an independent join m th-power are all m -tuples of the possible component values. Each such tuple specifies a unique path down the Kind tree from root to leaf, so we can view these values as an address in the tree.

4.3 Conditional FRPs and Conditional Kinds

The wiring diagram for general joins, like in Figures 4.3, 4.1, and 4.5 can easily become complicated. So as we move to consider general joins, it will be helpful to introduce some new hardware that make joins easier to express and work with.

Figure 4.13 shows an example of a *selector switch*. On the input side, the switch has a single port that can accept values from an FRP. On the output side, the switch has a port for each of several *specific* values, which can be arbitrary tuples of the *same dimension*. When a signal is passed through the input port, the switch checks the value against its allowed outputs and passes the signal through the corresponding output port. An input signal must represent one of the possible output values; if not, the input port glows red to indicate the error. (Overheating or other bad things may occur when this happens.)

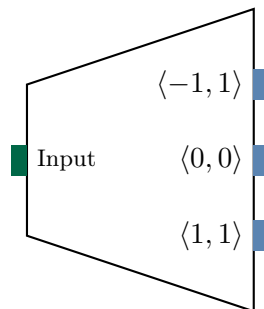


FIGURE 4.13. An example selector switch with input port on the left and labeled output ports on the right. This switch accepts only the listed values at its input.

We will use selector switches to isolate the targets from the source, creating a new type of object called a **conditional FRP**. The following example illustrates building a general join and then splitting it into more modular pieces.

Example 4.5.

Some number of people in a population have a particular disease. When an individual is tested for the disease, random errors in the process give an uncertain result that depends on whether the person has the disease. Roughly speaking:

- Roughly 1 out of a 1000 people have the disease.
- When 1000 people *with* the disease are tested, roughly 950 will test positive.
- When 1000 people *without* the disease are tested, roughly 10 will test positive.

Our goal in this situation is make a good prediction on whether a patient has the disease given their test result. For this example, however, we will focus on building an FRP to describe this system.

The system evolves in two stages: the patient's disease status is determined, and then, contingent on that outcome, the patient is tested for the disease. As usual, we associate a number with each outcome:

$$\begin{array}{ll} 0 \leftrightarrow \text{No Disease} & 0 \leftrightarrow \text{Test Negative} \\ 1 \leftrightarrow \text{Disease} & 1 \leftrightarrow \text{Test Positive} \end{array}$$

The first stage (disease status) is represented by an FRP D with Kind

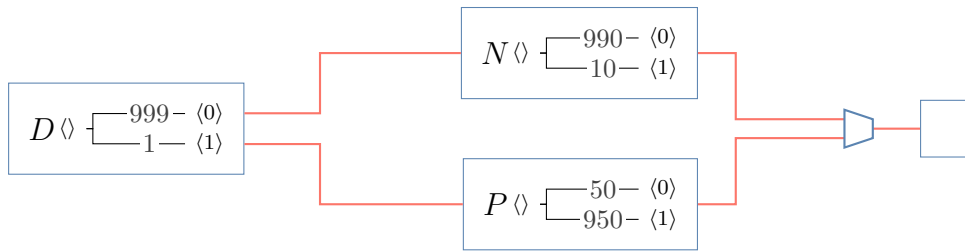
$$\langle \rangle \begin{cases} \text{---} 999 \text{---} \langle 0 \rangle \\ \text{---} 1 \text{---} \langle 1 \rangle \end{cases}$$

which reflects our understanding of the disease prevalence in the population. The second stage is represented by two FRPs, call them N and P for “negative” and “positive,” that represent the patient's test results in the situations where they do not and they do have the disease. These FRPs have Kinds:

$$\langle \rangle \begin{cases} \text{---} 990 \text{---} \langle 0 \rangle \\ \text{---} 10 \text{---} \langle 1 \rangle \end{cases} \quad \langle \rangle \begin{cases} \text{---} 50 \text{---} \langle 0 \rangle \\ \text{---} 950 \text{---} \langle 1 \rangle \end{cases}$$

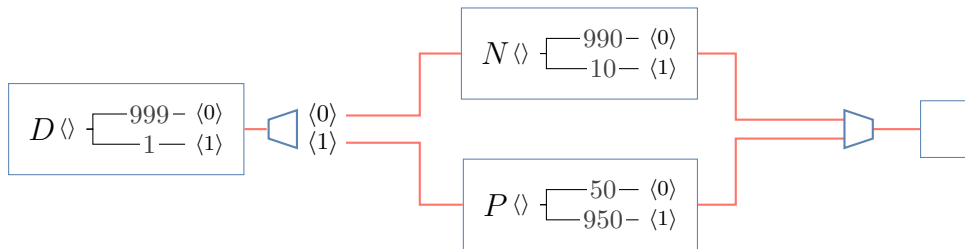
reflecting our understanding of the test's performance.

The full system is described by a join with the following wiring diagram:



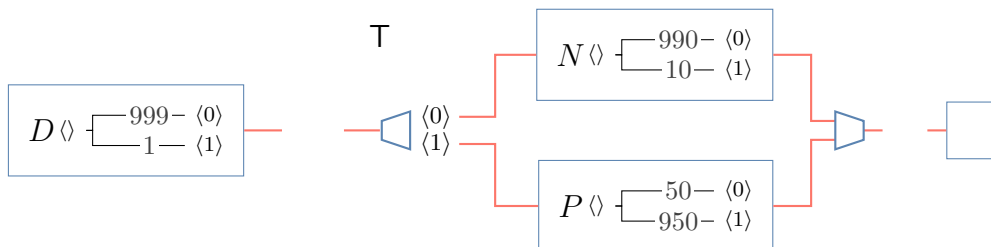
Note the contingency here: D 's value determines which of N and P is activated. Unlike in an independent join, N and P have different Kinds. When we push the button on the join FRP, it activates D then in turn one of N and P , and the join FRP shows the *combined* value on its display, a tuple that concatenates the value of D and the value of the activated target (N or P).

Now, let's make one trivial change by inserting a selector switch:



Here, we have connected D 's All output port to the input of the switch and the switch's output ports to each of the target FRPs. Despite the addition of the switch, the two wirings are completely equivalent. The switch plays the same role as the internal circuitry in the FRP that controls its output ports. We have redundantly reproduced that function in the switch.

If we cut the wires in the second wiring before the switch and after the multiplexer, we decompose the join system into three pieces:



On the left, we have D , the source, as before. On the right, we have the empty FRP that will be reconfigured to the join FRP. And in the middle, we have an object, which we have named T , of a new type that associates each input in some pre-specified set with an FRP. When connected to a source FRP producing only valid inputs, like D , it acts as if we had wired the target FRPs directly to the source, as in the previous two diagrams. We call T a **conditional FRP**. It represents the patient's test outcome *conditional* on the patient's disease status.

The conditional FRP T acts like N when given input $\langle 0 \rangle$ and like P when given input $\langle 1 \rangle$. Connecting the All output port of any FRP with values $\{\langle 0 \rangle, \langle 1 \rangle\}$ to the conditional FRP's input port and connecting T 's output to an empty FRP produces a valid join FRP.

The conditional FRP in the previous example is of the DIY¹⁵ variety. We built it from existing FRPs N and P , wires, multiplexer, and a selector switch tailored to the chosen input values. And that's fine. Fortunately, conditional FRPs have proved so useful in practice, that the FRP Marketplace offers conditional FRPs *integrated into a single device that looks like an ordinary FRP*. It receives its input value through its input port, which is labeled with the valid input values, and transmits its output value through its output port. We depict conditional FRPs in wiring diagrams as:

¹⁵"Do it yourself"



which puts the wiring for a device like T into a single package.

Before looking at conditional FRPs more formally, let us play with Example 4.5 in the `frplib`. Here, `choice(u, v, weight_v)` is a Kind factory producing Kinds with two values u and v and respective weights 1 and `weight_v`.

```
pgd> D = frp(choice(0, 1, 1/999))
pgd> N = frp(choice(0, 1, 1/99))
pgd> P = frp(choice(0, 1, 19))
```

In `frplib`, we can define a conditional FRP in several ways: as a dictionary mapping input values¹⁶ to FRPs,

```
pgd> T = conditional_frp({ 0: N, 1: P })
```

as a named function that returns FRPs,

```
pgd> @conditional_frp(codim=1, target_dim=1)
```

¹⁶In the scalar (codim=1) case, the dictionary keys can be scalars or 1-dimensional tuples, written $(0,)$ in Python.

```
...> def T(value):
...>     if value == 0: # Inputs are scalars when codim=1
...>         return N
...>     return P
```

or as an anonymous function¹⁷ that returns FRPs:

```
pgd> T = conditional_frp(lambda value: N if value == 0 else P, codim=1)
```

`conditional_frp` used as a function or a decorator wraps the given mapping into a conditional FRP object. When the codimension is 1, a wrapped function must take a *scalar* argument; in all other cases, the function's argument will be a tuple. When passing a dictionary with codimension set to 1, you may use scalar keys.

In practice, rather than use pre-defined variables like `N` and `P`, we more often define the target FRPs directly in the conditional FRP. For example, with for the above dictionary, we would typically write

```
pgd> T = conditional_frp({
...>     0: frp(choice(0, 1, 1/99)), # Like N
...>     1: frp(choice(0, 1, 19))    # Like P
...> })
```

The conditional FRP holds the specified FRPs and remembers their state and values over all uses. However, if the `auto_clone` optional argument is set to `True`, every use of the conditional FRP gives a fresh clone, which is useful when reusing a single conditional FRP in simulations.

The `conditional_frp` function can deduce the valid inputs of a conditional FRP from a dictionary but not from a function. You can specify such properties of the conditional FRP with optional arguments. It is good practice to give the codimension (via `codim`) and dimension (via `dim` or `target_dim`),¹⁸ where possible and appropriate. You can also use `domain` to specify the valid inputs, which causes an error to be raised for an invalid value. The domain can be an explicit list¹⁹ of valid values, of common dimension, or a predicate that returns true for valid values.²⁰ For instance, to indicate that the function `T` above accepts only values 0 and 1:

```
pgd> @conditional_frp(domain=[0, 1], target_dim=1)
...> def T(value):
...>     if value == 0:
...>         return frp(choice(0, 1, 1/99))
...>     return frp(choice(0, 1, 19))
```

¹⁷In Python, anonymous functions are written with the `lambda` keyword as `lambda args: expr` taking arguments `args` and returning the value of expression `expr`.

¹⁸The `dim` includes the length of the input vector; `target_dim` does not.

¹⁹...or a set or a generator/iterator

²⁰When `codim=1`, a domain predicate should accept scalar inputs.

With an explicit domain, `conditional_frp` can deduce the codimension for a function, so it need not be supplied. Alternatively, you can handle invalid inputs in the function:

```
pgd> @conditional_frp(codim=1)
...> def T(value):
...>     if value == 0:
...>         return frp(choice(0, 1, 1/99))
...>     if value == 1:
...>         return frp(choice(0, 1, 19))
...>     raise MismatchedDomain(f'Value {value} for T must be 0 or 1')
```

When we examine the conditional FRP `T` in the playground, it shows us the mapping from input values to FRPs just like the wiring diagram does.

```
pgd> T
A conditional FRP with wiring:
<0>: An FRP with value 0
<1>: An FRP with value 1
```

We can also evaluate a conditional FRP like a function. It returns the target FRP corresponding to the given value, or raises an error if the input value is invalid.²¹

```
pgd> T(0)
An FRP with value <0>
pgd> T(1)
An FRP with value <1>
pgd> T(2)
Value <2> not in the domain of this conditional FRP.
```

(Recall that we treat $T(\langle x \rangle)$ and $T(x)$ as equivalent. See Chapter 15 of Interlude F.)

Every conditional FRP like `T` has an associated function `T.joined` that takes an input value and returns the corresponding target FRP with the input value prepended to the tuple, as you get in the join operation.

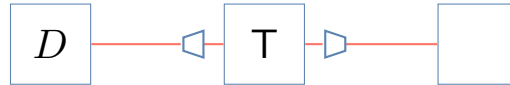
```
pgd> T.joined(0)
An FRP with value <0, 0>
pgd> T.joined(1)
An FRP with value <1, 1>
pgd> T.joined(2)
Value <2> not in the domain of this conditional FRP.
```

²¹`frplib` also accepts a value in braces `[]` to get the target, e.g., `T[0]`. We prefer the function style, but the option is there.

Let prepend_c denote the “prepend c ” statistic for a tuple c , where $\text{prepend}_c(v) = c :: v$. Then, we can write

$$\begin{aligned} T(0) &= N \\ T(1) &= P \\ T.\text{joined}(0) &= \text{prepend}_0(N) \\ T.\text{joined}(1) &= \text{prepend}_1(P). \end{aligned}$$

We form the *join* depicted in the previous example by connecting the output of D to the input of the conditional FRP T and the output of T to a new, empty FRP. The wiring diagram for the join thus becomes:



Algebraically, we define a join operator $\triangleright$ (or $\gg$ in `frplib`). We can give this an FRP on the left and a compatible conditional FRP on the right.²² The join FRP is given by $D \triangleright T$, or $D \gg T$ in `frplib`.

²²This operator is more flexible than this, as we will see.

```
pgd> D >> T
An FRP with value <0, 0>
```

In this run, D has value 0 which is passed to T , giving $T.\text{joined}(0)$ as a result. (Of course, your values may differ for these FRPs.)

Having gotten a feel for conditional FRPs, their wiring and evaluation, we can now formalize the idea.

Definition 21. A **conditional FRP** is a function R that maps values in a finite set $\mathcal{V}$ to FRPs. We call $\mathcal{V}$ the *domain* of R and the corresponding FRPs the *targets*.

For an input value $v \in \mathcal{V}$, $R(v)$ returns the corresponding target FRP, and the associated function $R.\text{joined}(v)$ returns the target FRP transformed with the statistic prepend_v . That is, if w is the value of the target $R(v)$, then $v :: w$ is the value of $R.\text{joined}(v)$, and vice versa.

We require that the target FRPs returned for inputs of a input dimension all have the same dimension. If a conditional FRP accepts values of dimension m and the target FRPs for such inputs have dimension n , we say that the conditional FRP has

- **codimension** m ,

- **target dimension** n ,
- **dimension** $m + n$, and
- **type** $m \rightarrow m + n$.

Every FRP of dimension n is also a conditional FRP of type $0 \rightarrow n$.

For example, the conditional FRP **T** from Example 4.5 has codimension 1, target dimension 1, dimension 2, and type $1 \rightarrow 2$. This type tells us that the output of **T** can be connected to the input of another conditional FRP that accepts pairs of 0s and 1s.

The reason we distinguish between the target dimension (n) and the dimension ($m + n$) is that the latter is the dimension of the FRP that results when we use a conditional FRP in a join. This will be convenient for assessing which joins are possible.

We can think of a regular FRP as a conditional FRP of codimension 0, meaning that it accepts the trivial input $\langle \rangle$, or put another way, it accepts no arguments.²³ (An ordinary FRP still accepts a connection at its input port from an output port of another FRP for transforming with statistics and building joins, but this is a distinct phenomenon.)

The words “conditional” and “condition” arise in several contexts throughout this material, which can be confusing. In this case, the word conditional is meant as in programming: “if we get a 0 then use N , else use P .” A conditional FRP (or conditional Kind as we will soon see) is choosing an FRP (or Kind) contingently on some value. We will often express such contingency with the word “given”: given an input v , the result looks like this; given an input w , the result looks like that; and so on.

Remember that a conditional FRP stands in for the target layer in the wiring of a join. We get an FRP that is selected based on some input, and the different target FRPs can have different properties.

Puzzle 52. In the playground, build a conditional FRP whose domain contains three values and that returns FRPs of at least two different Kinds.

A conditional FRP **R** maps each input in its domain with an FRP. By taking the Kind of these FRPs, we get a function that maps each input in the domain to a Kind. Specifically, we can define the function **r** to map v to $\text{kind}(\mathbf{R}(v))$ for each input value v in the domain of **R**. We call this a *conditional Kind*.

In Example 4.5, the conditional FRP **T** activates N given input $\langle 0 \rangle$ and P given input $\langle 1 \rangle$. The conditional Kind **t** associated with **T** is illustrated in Figure 4.14.

²³See Example 14.3 in Interlude F. A function f from $\{\langle \rangle\}$ to any set $\mathcal{A}$ is isomorphic to an element of $\mathcal{A}$, the element “picked out” by $f(\langle \rangle)$.

This shows the Kind of the activated FRP given each valid input. Notice the direct correspondence between the wiring diagrams for t and T here and in Example 4.5.

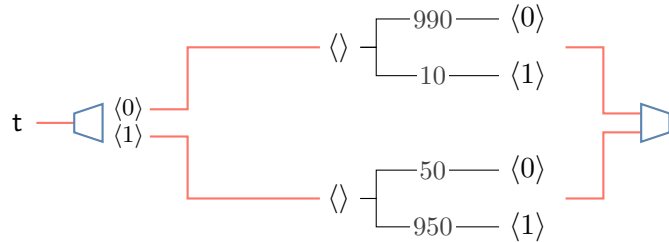


FIGURE 4.14. The conditional Kind t corresponds to the conditional FRP T in Example 4.5.

In `frplib`, we can also construct conditional Kinds directly in ways exactly analogous to how we constructed conditional FRPs, using `conditional_kind` as a function or decorator. For instance, following up on our earlier example, we can use a dictionary mapping values to Kinds

```
pgd> t = conditional_kind({
...>     0: choice(0, 1, 1/99),
...>     1: choice(0, 1, 19)
...> })
```

or a named function

```
pgd> @conditional_kind(domain=[0, 1], target_dim=1)
...> def t(value):
...>     if value == 0:
...>         return choice(0, 1, 1/99)
...>     return choice(0, 1, 19)
```

or an anonymous function

```
pgd> t = conditional_kind(
...>     lambda value:
...>         choice(0, 1, 1/99) if value == 0 else choice(0, 1, 19),
...>     domain=[0, 1], target_dim=1
...> )
```

Try these and look at each t in the playground. The optional arguments `codim`, `dim`, `target_dim`, and `domain` are used as for `conditional_frp`. It is good practice to specify these (especially `codim` and either `dim` or `target_dim`) when building a

conditional Kind from a function so that errors can be raised on invalid input or operations.

Note that just as for a conditional FRP, when you look at `t` in the playground, it shows you the mapping of values to Kinds, like the wiring diagram in Figure 4.14.

```
pgd> t
A conditional Kind with wiring:
      ,---- 0.99000  ----- 0
<0>:  <> -|
      `---- 0.010000 ---- 1

      ,---- 0.050000 ---- 0
<1>:  <> -|
      `---- 0.95000  ----- 1
```

This shows the Kind of the FRP attached to each input wire. You can see these Kinds individually when evaluating `t` as a function:

```
pgd> t(0)
      ,---- 0.99000  ----- <0>
<> -|
      `---- 0.010000 ---- <1>
pgd> t(1)
      ,---- 0.050000 ---- <0>
<> -|
      `---- 0.95000  ----- <1>
```

It returns the Kind of the target associated with each input, or an error for an invalid input.²⁴ The associated function `t.joined` returns the Kind of the target FRP with the input prepended to its value, as produced by a join operation.

```
pgd> t.joined(0)
      ,---- 0.99000  ----- <0, 0>
<> -|
      `---- 0.010000 ---- <0, 1>
pgd> t.joined(1)
      ,---- 0.050000 ---- <1, 0>
<> -|
      `---- 0.95000  ----- <1, 1>
```

Observe that $t(0) = \text{kind}(T(0))$ and $t(1) = \text{kind}(T(1))$.

²⁴Again, we can use either `t(value)` or `t[value]` to get the *target* Kind.

Definition 22. A **conditional Kind** is a function r that maps values in a finite set $\mathcal{V}$ to Kinds. We call $\mathcal{V}$ the *domain* of r and the corresponding Kinds the *targets*.

When given an input value $v \in \mathcal{V}$, $r(v)$ returns the Kind of the target associated with v . The associated function $r.\text{joined}$ returns the Kind of the target transformed by the statistic prepend_v that prepends v to each leaf node of the tree. That is, if $r(v) = w$, $r.\text{joined}(v) = v :: w$.

We require that for inputs of a given dimension, the target Kinds all have the same dimension. If a conditional Kind accepts values of dimension m and if the returned Kinds have dimension n , we say that the conditional Kind has

- **codimension** m ,
- **target dimension** n ,
- **dimension** $m + n$, and
- **type** $m \rightarrow m + n$.

Every Kind of dimension n is also a conditional Kind of type $0 \rightarrow n$.

Every conditional FRP R has an associated conditional Kind r with the same domain defined by

$$r(v) = \text{kind}(R(v)) \quad (4.15)$$

for every v in the domain of R . Thus, $r = \text{kind} \circ R$ is a composition of functions, “kind after R ,” where kind maps FRPs to their Kinds. This is good, but it is useful to extend the kind function to *also* accept conditional FRPs and return their corresponding conditional Kind. So, we can write $r = \text{kind}(R)$ to express the relationship, and in `frplib`, we can use `kind(R)` to find the corresponding conditional Kind.

```
pgd> kind(T)
A conditional Kind with wiring:
      ,---- 0.99000  ----- 0          ,---- 0.050000 ---- 0
<0>:  <> -|                                <1>:  <> -|
      `----- 0.010000 ---- 1          `----- 0.95000  ----- 1

pgd> conditional_kind({ 0: kind(N), 1: kind(P) })
```

Both of these are the same object as what we got for `t` earlier.

As with FRPs, we can view regular Kinds as a special case of conditional Kinds with codimension 0, meaning they take no input.

Puzzle 53. In the playground, build a conditional Kind to describe the following system: flip three coins and take the sum of h balanced, six-sided dice, where h is the number of heads flipped.

It will likely be easiest to define this with a named function decorated by `@conditional_kind`. If you want to specify a domain, you can pass one of the following as the `domain` argument:

```
((i,j,k) for i in [0,1] for j in [0,1] for k in [0,1])
```

including the parentheses or, after first entering, `import itertools`,

```
itertools.product([0,1], repeat=3)
```

THE KERNEL OF A CONDITIONAL KIND. Recall that the *kernel* of a Kind k is a function that takes candidate values and returns the canonical weight in that Kind for that value, or 0 if the value is not possible. When working mathematically with Kinds, we overload the symbol for the Kind by using it for the kernel function as well, e.g., $k(v)$. (In `frplib`, we instead use the `.kernel` attribute, so we would write `k.kernel(v)`.)

A conditional Kind is, in essence, a function that takes input values and returns a Kind. The returned Kind has a kernel, like any Kind. If c is a conditional Kind and u a valid input value, then the target $c(u)$ is a Kind, so it would follow that $c(u)(v)$ is the kernel of that Kind at the value v . **But:** that becomes hard to parse and use, so we adopt a more convenient and evocative notation for the kernels derived from conditional Kinds.

Kernels from a Conditional Kind.

If c is a conditional Kind and u is a valid input value, then $c(u)$ is a target Kind. We denote its kernel evaluated at value v by

$$c(v \mid u) \tag{4.16}$$

The $\mid$ is read as “given,” as it is throughout the book. That language helps remember which argument is the given input value and which is the candidate value whose weight we are finding. The value after the given bar is the input value and must be a valid input to the conditional Kind. The value before the given bar is the value at which we are evaluating the target’s Kernel.

For each valid u , the function $\langle v \rangle \mapsto \mathbf{c}(v \mid u)$ is the kernel of the Kind $\mathbf{c}(u)$. This function is not well-defined for invalid inputs u .

If $\mathbf{c}$ is an ordinary Kind – that is, it has codimension 0 – then the only valid input is the empty tuple $\langle \rangle$, so we drop the given bar and input value. This gives us our ordinary notation for the kernel at v , $\mathbf{c}(v)$.

We are using the conditional Kind $\mathbf{c}$ as a function in two different ways. Without the given bar, we get a mapping $\langle u \rangle \mapsto \mathbf{c}(u)$ from values to target Kinds. With a valid value after the given bar, we get a mapping $\langle v \rangle \mapsto \mathbf{c}(v \mid u)$ from values to their weights *in the target Kind*. The presence of the given bar makes the context clear. In the case of an ordinary Kind, only the second type of function is relevant anyway.

For example, in Figure 4.14, the conditional Kind $\mathbf{t}$ has the following kernel:

$$\begin{array}{ll} \mathbf{t}(0 \mid 0) = 0.990 & \mathbf{t}(1 \mid 0) = 0.010 \\ \mathbf{t}(0 \mid 1) = 0.050 & \mathbf{t}(1 \mid 1) = 0.950. \end{array}$$

Again, the input value to the conditional Kind is given *after* the given bar. The first row in the previous display gives the weights $\mathbf{t}(0 \mid 0)$ and $\mathbf{t}(1 \mid 0)$ for the Kind $\mathbf{t}(0)$. The second row gives the weights $\mathbf{t}(0 \mid 1)$ and $\mathbf{t}(1 \mid 1)$ for the Kind $\mathbf{t}(1)$.

Key Take Aways

In wiring a join, we connect the source FRP to all of the targets and the targets to the new FRP. With the addition new hardware – a selector switch and a multiplexer – we can separate the source from the targets and package the targets and their wiring all in a separate, integrated device with one output. All targets must have the same dimension.

A conditional FRP represents such a device. It accepts values in a fixed set of valid inputs, and the input determines which target FRP will activate. The values of all targets are wired through the multiplexer to a single output.

A **conditional FRP** is a function from the set of valid inputs to FRPs, with one distinct target FRP per possible input value. A **conditional Kind** is an analogous function that maps a set of valid inputs to Kinds. The Kind returned for an input value v is the Kind of the target FRP at v .

There are two ways to access a conditional FRP or Kind when we pass it an input value. The “raw” value is from the activated target only. The “joined”

value combines the input value and the activated target value into one larger tuple. We access the former by evaluating the function and the latter with the associated `.joined` method.

The properties of a conditional FRP or Kind are

- its **codimension**, which is the dimension of the valid input tuples;
- its **target dimension**, which is the dimension of the target FRPs or Kinds;
- its **dimension**, which is the sum of the codimension and target dimension;
- its **type**, of the form $c \rightarrow d$ for codimension c and dimension d .

Every regular FRP and regular Kind of dimension d can be considered a conditional FRP and conditional Kind of type $0 \rightarrow d$. They take no input and return a single FRP or Kind (themselves).

If c is a conditional Kind and u is a valid input, then $c(u)$ is the target Kind. We write its kernel with special notation to make it easier to parse: $c(v \mid u)$ is the kernel evaluated at v . The value after the given bar is the input (which is “given” to the conditional Kind), and the value before the given bar is where we evaluate the kernel.

4.4 General Joins

For many random systems, the system’s evolution at later stages is contingent on what happens at earlier stages. To define such systems and build FRPs that represent them, it is generally easier to take a modular approach: describe the individual stages and the mechanisms that connect them to describe and represent the system as a whole. This is purpose of joins.

As we have seen, joins allow us to build systems in stages where the value produced at earlier stages determines the random mechanism used to produce the value at the next stage. When we combine FRPs this way, the values from earlier stages pass through and are combined with the values produced at later stages, so the values of a join FRP produces records the whole history of the system’s evolution up to that point. The join operation is the only one in the Big 3+1 in which we can add random variation when building a system.

General joins allow the targets to be FRPs with different Kinds, so the random mechanism at the later stage *depends on the output of the earlier stages*. Independent joins are a special case²⁵ where the stages do not interact and there is no contingency. Independent joins can be viewed as wiring each source to targets with *the same Kind*.

²⁵ An important special case.

In Section 4.1, we saw the physical interpretation of a general join, as illustrated in Figures 4.1, 4.3, and 4.4. The *source* FRP represents the history of the system's evolution so far, its value being the output of all earlier stages. For each possible value of the source, we connect the corresponding output port to the input port of a *target* FRP. Each target represents the value produced at the next stage given that the corresponding value was produced at the earlier stages. The targets output ports are connected through a multiplexer to a new, empty FRP that is reconfigured to become the join FRP. When the source is activated, exactly one target will be activated, and the join's value is the concatenation of the source's and activated target's value, giving the entire history of the system to that point.

At its heart, this is a simple idea: one thing happens, then another thing happens based on the first, and we collect the whole history. Before formalizing the join operation, let's look at another simple example.

Example 4.6. You play a game where you flip a balanced coin. If the coin comes up heads, you receive \$2.50. If the coin comes up tails, you roll a balanced 6-sided die and receive in dollars the value of that roll.

The source, the first thing that happens, is the coin flip. Encode tails and heads as 0 and 1, we have

```
pgd> Flip = frp(choice(0, 1))
```

Now, the next thing that happens is either a die roll or a constant (2.5). A conditional FRP describes both possibilities for the later stage:

```
pgd> Payoff = conditional_frp({
...>    0: frp(uniform(1, 2, ..., 6)) # Saw tails, roll the dice
...>    1: frp(constant(2.5))         # Saw heads, get $2.50
...> })
```

Based on what happens in the earlier stage (*Flips*), we choose a random quantity that gives the output of the later stage.

A join puts wires these together and records the values of both early and later stages. In *frplib*, the join operator is `>>`, which takes an FRP (or Kind) on the left and a conditional FRP (or conditional Kind) on the right.

```
pgd> Flip >> Payoff
An FRP with value <1, 2.5>
pgd> FRP.sample(10_000, Flip >> Payoff)
```

Summary of Output Values		
Values	Count	Proportion
<0, 1>	832	8.32%
<0, 2>	844	8.44%
<0, 3>	844	8.44%
<0, 4>	830	8.3%
<0, 5>	823	8.23%
<0, 6>	853	8.53%
<1, 5/2>	4974	49.74%

The summary table shows us the dependence of the later stage on the output of the early stage.

Conditional FRPs (and their conditional Kinds) package all the join targets in a single device. A conditional FRP only accepts inputs in a designated set, and each valid input value corresponds to one target FRP. It is a convenient abstraction to simplify our wiring diagrams for complex joins. We depict a conditional FRP as



in wiring diagrams, but we drop the selector switch and multiplexer for ordinary FRPs, which have codimension 0. We then write the wiring diagram for a join as



with the source FRP on the left, a conditional FRP containing the targets in the middle, and the final join FRP on the right.

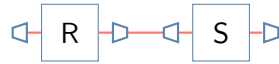
In fact, general joins are somewhat more versatile than that. We have seen that an FRP of dimension d can be considered as a conditional FRP of type $0 \rightarrow d$. The logic of joins works with any two conditional FRPs that are *compatible*.

Definition 23. We say that two conditional FRPs R and S (or two conditional Kinds r and s) are **compatible** if all possible values of R .joined (r .joined) across all its inputs belong to the domain of S (s). That is, all possible values of the former are valid inputs to the latter. The dimension of R (r) equals the codimension of S

(s).

So, if S has type $p \rightarrow n$, then R must have type $m \rightarrow p$ for some m .

This tells us that the possible outputs of the first stage are valid inputs to the second stage. It also implies that the types of the two stages match, $m \rightarrow p$ and $p \rightarrow n$, producing a result of type $m \rightarrow n$. A join of two compatible conditional FRPs connecting the output of one to the input of the other, and the result is another conditional FRP. The wiring diagram for this new conditional FRP is



which can then be used in subsequent joins. Notice that when R here has type $0 \rightarrow p$, it is a regular FRP, so the join will be as well, with type $0 \rightarrow n$.

The algebraic interpretation of joins is expressed through an operator, $\triangleright$ in mathematics and `>>` in `frplib`, that acts on two conditional FRPs (or two conditional Kinds) to produce a new conditional FRP (or conditional Kind). If R has type $m \rightarrow p$ and S has type $p \rightarrow n$, then $R \triangleright S$ has type $m \rightarrow n$. The most common case is when we use an FRP (or Kind) on the left to produce a new FRP (or Kind). If R is an FRP of type $0 \rightarrow p$ and S is a conditional FRP of type $p \rightarrow n$, then $R \triangleright S$ is an FRP of dimension n (type $0 \rightarrow n$).

The procedural interpretation of a general join extracts the value of the source and the value of the activated target to produce the combined output value. An FRP procedure applies when the left side of the join is an FRP (type $0 \rightarrow p$):

```
FRP  $R \triangleright S$ :
   $w \leftarrow R$ 
   $x \leftarrow S(w)$ 
  return  $w :: x$ 
```

and in `frplib`, we can do this in exactly the same way or use the `.joined` method if we prefer

<pre>@frp def R_join_S(): w = yield R x = yield S(w) return (w, x)</pre>	<pre>@frp def R_join_S(): w = yield R wx = yield S.joined(w) return wx</pre>

Remember that the `S.joined` function returns the FRP that represents the input prepended to the target, whereas `S` returns just the target. Note that we cannot just

return `S.joined(w)` in the definition on the right. The yield operation must be done separately to extract the value before we can use it; put another way, the return statement must give a value not an FRP. For a join of two non-trivial conditional FRPs, we use the same shape procedure but label our procedure differently and provide input parameters.

conditional FRP $(R \triangleright S)(v)$:

```

 $w \leftarrow R(v)$ 
 $x \leftarrow S(v :: w)$ 
return  $v :: w :: x$ 

```

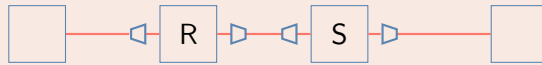
and in `frplib`:

<pre> @conditional_frp def R_join_S(v): w = yield R(v) x = yield S(join(v, w)) return (v, w, x) </pre>	<pre> @conditional_frp def R_join_S(v): vw = yield R.joined(v) vwx = yield S.joined(vw) return vwx </pre>

which are equivalent in result. Note that for all the `frplib` procedures, we cannot just return `S.joined(w)` or `S.joined(vw)`. Again, we cannot just return `S.joined(vw)` here; the yield operation must be done separately.

Definition 24. If R and S are compatible conditional FRPs of respective types $m \rightarrow p$ and $p \rightarrow n$, then their **join** is the conditional FRP $R \triangleright S$ of type $m \rightarrow n$. This is obtained by connecting the output of R to the input of S . In particular, if R is an ordinary FRP of dimension p (i.e., has type $0 \rightarrow p$) then the join $R \triangleright S$ is an ordinary FRP of dimension n (i.e., has type $0 \rightarrow n$).

If we connect an m -dimensional source to this and a new, empty FRP at the end, we get the wiring diagram for a join that yields an n -dimensional FRP:



The value of a join FRP records the entire history, including the initial input and the values produced by target FRPs at every stage. Given input v in the domain of R , $R \triangleright S$ produces an FRP with value $v :: w :: x$ where w is the value of the target FRP $R(v)$ and x is the value of the target FRP $S(v :: w)$. The domain of $R \triangleright S$

equals the domain of R .

The procedural form for a join with an FRP source is

```
FRP  $R \triangleright S$ :
   $w \leftarrow R$ 
   $x \leftarrow S(w)$ 
  return  $w :: x$ 
```

and for the more general case

```
conditional FRP  $(R \triangleright S)(v)$ :
   $w \leftarrow R(v)$ 
   $x \leftarrow S(v :: w)$ 
  return  $v :: w :: x$ 
```

Keep in mind that in the join operation, the input values to a conditional FRP are passed forward into the output, prepended to the activated target's value. This is the value that is passed on to the next stage of a join. So given input v , the value of $R \triangleright S$ is $v :: w :: x$ when

1. w is the value of the target $R(v)$, and
2. x is the value of the target $S(v :: w)$.

Let's trace this through. Given input v , R activates the corresponding target $R(v)$ and produces the FRP $\text{prepend}_v(R(v))$.²⁶ When the activated target FRP has value w , this has value $v :: w$. This value is then passed as input to S , activates the target $S(v :: w)$ and produces the FRP $\text{prepend}_{v :: w}(S(v :: w))$. This FRP has value $v :: w :: x$ when the target FRP $S(v :: w)$ has value x . The join FRP thus has value $v :: w :: x$, which includes the source input and the value produced by each constituent FRP activated in the join.

If R is an ordinary FRP (i.e., has codimension $m = 0$), then $R \triangleright S$ is also an ordinary FRP. Its value concatenates the value of R and the contingent value of S given the value of R . In Example 4.5, for instance, we defined an FRP D that represents whether the patient has the disease and a conditional FRP T that represents the outcome of the test contingent on whether the patient has the disease. The join $D \triangleright T$ is an FRP that represents both the patient's disease status and test outcome. When we activated this FRP in the playground on page 264, this FRP had value $\langle 0, 0 \rangle$, indicating that the patient does not have the disease and tested negative.

²⁶Recall that $\text{prepend}_v(w) = v :: w$. This is the builtin statistic `Prepend(v)` in `frplib`.

Example 4.7. Waiting at the airport, my flight has been delayed several times, and I worry that the delays will get worse. I can switch to a later flight to my destination, hoping that the later flight leaves before my original (delayed) flight. Or I can stick with my original flight hoping that whatever problem is causing the delay is resolved before the later flight is scheduled to leave.

Let A be the conditional FRP that represents the difference between my actual and scheduled arrival times at my destination, contingent on my choice of flight. This returns a different FRP depending on whether I choose the later or original flight.

Lacking any useful information, I choose the flight by a coin flip, an FRP F with Kind

$$\langle \rangle \text{ --- } \begin{cases} 1 \text{ --- } \langle 0 \rangle \\ 1 \text{ --- } \langle 1 \rangle \end{cases}$$

The join $F \triangleright A$ is the Data FRP, with dimension 2. Its first component represents the flight I chose, with $F = \text{proj}_1(F \triangleright A)$, and the second component $D = \text{proj}_2(F \triangleright A)$ is the FRP that represents the delay in my arrival time with my chosen flight.

Puzzle 54. In the `frplib` playground, create an appropriate conditional FRP A for the previous example. Construct and study the resulting Data FRP $F \triangleright A$ and delay FRP D and their Kinds.

Example 4.8. One piece of pizza remains at the FRP Marketplace office party, and three friends each want it for themselves. Unwilling to divide it, they devise a scheme to select fairly among the three at random using only the large supply of FRPs with Kind

$$\langle \rangle \text{ --- } \begin{cases} 1 \text{ --- } \langle 0 \rangle \\ 1 \text{ --- } \langle 1 \rangle \end{cases}$$

that are piled in a corner of the room.

Let values 1, 2, and 3 label the three friends, and value 0 indicate that a decision has not yet been made. They start with an FRP C_0 that is a constant with value 0 and an FRP representing a balanced coin flip.

```
pgd> C_0 = frp(constant(0))
pgd> Flip = frp(choice(0, 1))
```

They then construct a conditional FRP:

```
pgd> F_1 = conditional_frp({
    0: clone(Flip) * clone(Flip) ^ (2 * Proj[2] + Proj[1]),
    1: frp(constant(1)),
    2: frp(constant(2)),
    3: frp(constant(3))
})
```

The input to `F_1` indicates their decision up to the previous stage. If they have already decided who gets the pizza (1, 2, or 3), `F_1` returns an FRP whose value is that choice. But if they have not decided (0), `F_1` returns an FRP that represents a transform of two coin flips giving value 0 for $\langle 0, 0 \rangle$, 1 for $\langle 0, 1 \rangle$, 2 for $\langle 1, 0 \rangle$, and 3 for $\langle 1, 1 \rangle$. This value represents the choice of pizza winner. Because all four possibilities for two balanced coin flips have the same weight, the three friends are fairly selected, *except* there is still a possibility (value 0) wherein the choice is not resolved. Will the pizza go to waste? They find

```
pgd> F_1
A conditional FRP with wiring:
<0>          An FRP with value 0
<1>          An FRP with value 1
<2>          An FRP with value 2
<3>          An FRP with value 3
pgd> C_0 >> F_1
An FRP with value <0, 0>
pgd> C_1 = (C_0 >> F_1) ^ Proj[2]
An FRP with value <0>
```

The join selects from `F_1` the FRP corresponding to `C_0`'s value (which we know to be 0) and concatenates the value of `C_0` with the value of that FRP. Had that FRP had value 1, 2, or 3, then the FRP `C_1` would give the choice among the three friends for who gets the pizza. But alas, it has value 0, which means that the choice is not yet resolved. So they do it again!

Because they want a conditional FRP “F_2” that looks just like F_1 but with different clones, they can either define a *factory* for the conditional FRP

```
pgd> def F_n():
    return clone(F_1)
```

which returns a conditional FRP with the same structure as F_1 and fresh targets, or they can pass `auto_clone=True` to `conditional_frp`

```
pgd> F_n = conditional_frp({
    0: clone(Flip) * clone(Flip) ^ (2 * Proj[2] + Proj[1]),
    1: frp(constant(1)),
    2: frp(constant(2)),
    3: frp(constant(3))
}, auto_clone=True)
```

which will use fresh clones of the targets every time it is used in a join. We will do the latter here. Now,

```
pgd> C_2 = (C_1 >> F_n) ^ Proj[2]
An FRP with value <0>
```

The drama continues.

```
pgd> C_3 = (C_2 >> F_n) ^ Proj[2]
An FRP with value <2>
```

So friend 2 eats the pizza. But was this really a fair process?

```
pgd> kind(C_0)
<> ----- 1 ----- 0
pgd> kind(C_1)
,----- 1/4 ----- 0
|----- 1/4 ----- 1
<> -|
|----- 1/4 ----- 2
`----- 1/4 ----- 3
pgd> kind(C_2)
,----- 1/16 ----- 0
|----- 5/16 ----- 1
```



```

<> -|
    |---- 5/16 ---- 2
    `---- 5/16 ---- 3
pgd> kind(C_3)
    ,---- 0.015625 ---- 0
    |---- 0.32813 ----- 1
<> -|
    |---- 0.32813 ----- 2
    `---- 0.32813 ----- 3

```

The weights on 1, 2, and 3 are equal at every stage, and after repeating the process n times, the weight on 0 is 4^{-n} , which gets small quickly, so after at most a few iterations, they are likely to resolve the choice. Friends 1 and 3 may be glum, but they cannot feel mistreated.

Puzzle 55. You have two independent FRPs X and Y , and you want to construct their independent join $X \star Y$ using the join operator $\triangleright$.

Describe a conditional FRP U that makes $X \triangleright U = X \star Y$.

Puzzle 56. Characters in *Dungeons & Dragons* who specialize in combat (called “fighters”) and who roll an 18 for their Strength attribute score can make an additional roll of a balanced 100-sided die to determine their “exceptional strength” sub-score in the range $[1..100]$. All other characters have exceptional strength 0.

Modify the FRP factory `dnd_character` in Example 4.1 to include an exceptional strength sub-score in the value. The FRP returned by this factory will have dimension 7. The factory should be defined with an optional argument that indicates if the character is a “fighter,” `False` by default (e.g., `def dnd_character(fighter=False)`).

Notice that there are different Kinds of joins at play here because the exceptional strength sub-score depends on the character’s Strength attribute.

A roll for exceptional strength is an FRP whose Kind can be generated in the playground with `uniform(1, 2, ..., 100)`.

Puzzle 57. If S, T, U are scalar FRPs and $W = S \star T \star U$, what are the components of W ? (The components of an FRP are described in Definition 12 on page 158.)

Puzzle 58. If X has components $\langle X_1, X_2, \dots, X_n \rangle$, is it always true that we can write $X = X_1 \star X_2 \star \dots \star X_n$? If so why? If not, can you find an example where this relationship does not hold?

The strong correspondence that we have seen several times so far between operations on FRPs and operations on Kinds holds also for joins. Just as we can build joins of conditional FRPs, we can build joins of conditional Kinds. The join operation on Kinds proceeds as described previously: for each value of the source, we attach a copy of target Kind at the corresponding leaf node of the source Kind, propagating the source values down the tree at the beginning of each node's tuple.

Definition 25. If r and s are compatible conditional Kinds of respective types $m \rightarrow n$ and $n \rightarrow p$, then their **join** is the conditional Kind $r \triangleright s$ of type $m \rightarrow p$.

We form the join as follows: For each valid input v to r and for each leaf node w in $r(v)$, attach the tree $s(w)$ to that leaf node, and propagate input values down the tree to form a valid Kind under the Level and Prefix rules. This gives a Kind tree for every v .

In particular, if r is an ordinary Kind of dimension n (i.e., has type $0 \rightarrow n$) then the join $r \triangleright s$ is an ordinary Kind of dimension p (i.e., has type $0 \rightarrow p$).

We would like to relate the Kind of a join to the join of the Kinds, but first, we need to make a key distinction. A join injects new randomness into the system based on the output of earlier stages. This is easiest to understand, when the new randomness depends on the earlier stages *only through the choice of target*. We call this a *clean join*.

Definition 26. If R is an FRP and S is a compatible conditional FRP, then $R \triangleright S$ is called a **clean join** if R is independent of $S(v)$ for all possible values v of R .

More generally, if R and S are compatible conditional FRPs, then $R \triangleright S$ is a **clean join** if $R(v)$ is independent of $S(v :: w)$ for all v in the domain of R and all possible values w of $R(v)$.

If R, S , and T are compatible conditional FRPs and if $R \triangleright S$ and $S \triangleright T$ are clean joins, then so is $R \triangleright S \triangleright T$.

The targets of a conditional FRP represents the randomness injected at the next

stage given the output of the earlier stages. A join is **clean** when the dependence between the earlier stages and the new randomness is captured entirely through the conditional FRP's input. In practice, one has to work to make a join not clean, so we will treat clean joins as the presumptive default, especially when building and describing systems stagewise.

Example 4.9. If X and Y are independent FRPs, then $X \star Y$ is a clean join. To see this, note that $X \star Y = X \triangleright F$ where $F = \text{const}(Y)$, the function that returns Y for every input. X and every target in F are independent by assumption, so the join is clean.

Example 4.10. Let X be a scalar FRP and ψ a compatible, non-constant statistic. Define a conditional FRP S with target $S(v) = \psi(X)$. That is, every target is the transformed FRP $\psi(X)$.

Then $X \triangleright S$ is *not* clean because each target $S(v)$ for possible value v of X is just $\psi(X)$ and thus dependent on X . (Knowing X 's value makes your predictions about $\psi(X)$ perfect.)

We can illustrate this further with a concrete case.

```
pgd> X = frp(uniform(1, 2, 3))
pgd> psi = (__ ** 2)
pgd> kind(X >> conditional_frp(lambda _: psi(X))
,---- 1/3 ---- <1, 1>
<> -+----- 1/3 ---- <2, 4>
      `----- 1/3 ---- <3, 9>
```

This leads us to a critical identity that expresses the compatibility of join operations on conditional FRPs and on conditional Kinds. If $R \triangleright S$ is a clean join, then

$$\text{kind}(R) \triangleright \text{kind}(S) = \text{kind}(R \triangleright S). \quad (4.17)$$

Put loosely: the join of Kinds equals the Kind of the join

In Example 4.5, let $d = \text{kind}(D)$, the Kind of the FRP D representing whether the patient has the disease, and let t be the conditional Kind defined earlier. Figure 4.15 shows the process for building the join Kind $d \triangleright t$. For each value v of d , we take the Kind $t(v)$ and attach it at the corresponding leaf node of d . On the left, the Figure shows the component Kinds; on the right, it shows join Kind, which we can convert to canonical form.

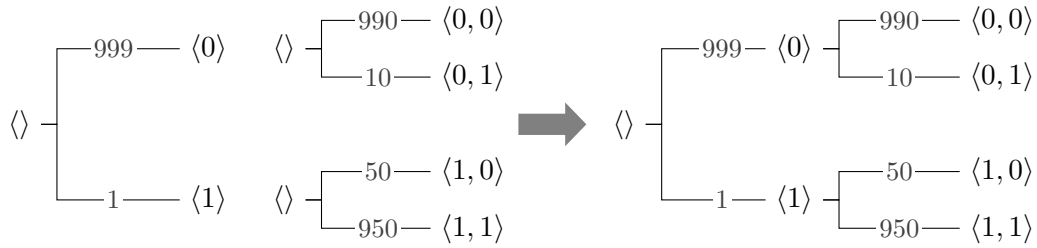


FIGURE 4.15. Constructing the Kind join $d \bowtie t$. For each value v of d , we take the Kind $t(v)$ and attach it at the corresponding leaf node of d , forming the Kind tree at right.

```
pgd> kind(D) >> kind(T)
,---- 0.98901 ----- <0, 0>
|---- 0.0099900 ----- <0, 1>
<> -|
|---- 0.000050000 ---- <1, 0>
`---- 0.00095000 ----- <1, 1>
pgd> FRP.sample( 1_000_000, D )
Summary of output values:
<0,0>    988786 (98.88%)
<0,1>    10176 ( 1.02%)
<1,0>     44 ( 0.00%)
<1,1>     994 ( 0.10%)
```

We are close here to answering the original question in the example, and will see a nice way to do it exactly in the next Chapter. But for now, we can see an approximate answer here; 994/10176 is the proportion of positive tests among those with the disease.

As another example of taking joins of Kinds, suppose we flip two balanced coins (equal weight on heads and tails) and given h heads, take the maximum of h rolls of a balanced six-sided die. (If $h = 0$, we take the maximum to be 0.) The FRPs representing each of the coin flips has Kind $\langle \rangle - \begin{cases} \frac{1}{2} & \langle 0 \rangle \\ \frac{1}{2} & \langle 1 \rangle \end{cases}$ where 0 means tails and 1 means heads. The FRPs representing the three possible rolls have Kinds

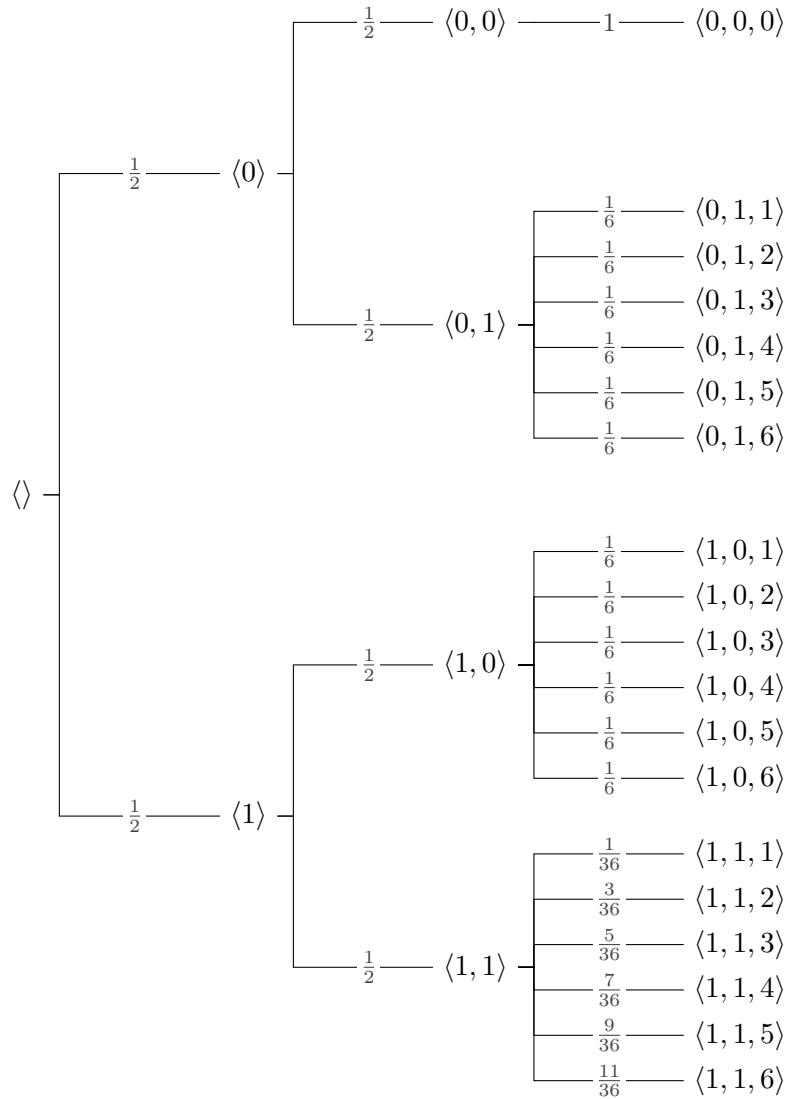
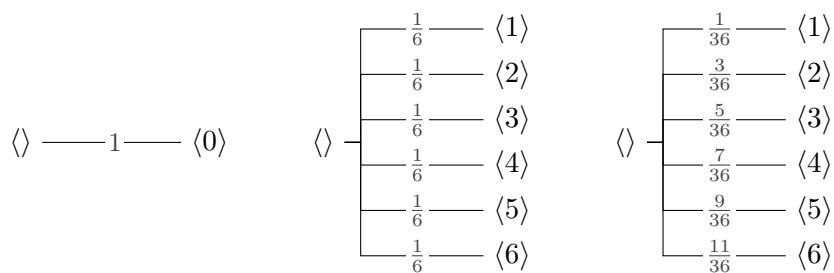


FIGURE 4.16. The unfolded join Kind representing two coin flips and a contingent dice roll.



Puzzle 59. Before proceeding, show how to derive these three Kinds in `frplib`. Explain your steps briefly.

The Kind describing the outcome of the system is a join: we take an independent join of the coin flip Kind with itself and then join with a conditional Kind that gives the Kind of the roll for each outcome of the coin flips. This is easier to see in `frplib`:

```
pgd> flip = choice(0, 1)           # Kind of a coin flip
pgd> roll0 = constant(0)           # Kind of no rolls
pgd> roll1 = uniform(1, 2, ..., 6) # Kind of one roll
pgd> roll2 = Max(uniform(1, 2, ..., 6) ** 2) # Kind of max of two rolls
```

Notice that in `roll2` we use an independent join of the single-roll Kinds before transforming with the statistic. Look at these Kinds in the playground and compare to the pictures above. The contingent roll is described by a conditional Kind:

```
pgd> roll = conditional_kind({
...>      (0, 0): roll0,
...>      (0, 1): roll1,
...>      (1, 0): roll1,
...>      (1, 1): roll2
...> })
```

Now we can form the join of two independent flips and a contingent roll:

```
pgd> outcome = flip ** 2 >> roll
pgd> unfolded(outcome)
```

Look at the canonical and unfolded trees in the playground; the latter reveals the steps of the join. First, a copy of `flip` is attached to each leaf node of `flip`, which is the independent join. Then, a copy of `roll0`, `roll1`, or `roll2` is attached to the corresponding leaf of the tree produced by the first stage, depending on how many heads occurred. Notice how the history of the process is recorded in the values at each node. The unfolded Kind is shown in Figure 4.16.

Keep in mind that the independent join `flip ** 2` forms the first two stages of the whole join. This is in fact equivalent to the general join `flip >> conditional_kind(flip, codim=1)` because for Kind `k`, `conditional_kind(k)` gives a conditional Kind with target `k` for any input. So we could write `outcome` with just the `>>` operator doing successive joins by

```
flip >> conditional_kind(flip, codim=1) >> roll
```

$$\frac{1}{3} \cdot \langle \rangle \begin{cases} 0 & \langle 1 \rangle \\ \frac{1}{2} & \langle 2 \rangle \\ \frac{1}{2} & \langle 3 \rangle \end{cases} + \frac{1}{3} \cdot \langle \rangle \begin{cases} 0 & \langle 1 \rangle \\ 0 & \langle 2 \rangle \\ 1 & \langle 3 \rangle \end{cases} + \frac{1}{3} \cdot \langle \rangle \begin{cases} \frac{3}{10} & \langle 1 \rangle \\ \frac{6}{10} & \langle 2 \rangle \\ \frac{1}{10} & \langle 4 \rangle \end{cases} = \langle \rangle \begin{cases} \frac{3}{30} & \langle 1 \rangle \\ \frac{11}{30} & \langle 2 \rangle \\ \frac{16}{30} & \langle 3 \rangle \end{cases}$$

FIGURE 4.17. The Kind of Monty's door choice in Example 4.13 expressed as a weighted average. Missing values in the Kinds are shown as edges with 0 weight.

Puzzle 60. Rewrite the conditional Kind `roll` in terms of a named function

```
@conditional_kind(codim=2)
def roll(flips):
    ...
```

that returns a Kind. You can enforce a domain constraint by using

```
@conditional_kind(codim=2, domain=lambda v: all(x in {0,1} for x in v))
```

as the decorator above to test that the input tuple consists of only 0s and 1s.

Next, we consider several examples that illustrate these ideas. These examples nicely illustrates the contingent evolution that joins capture and how we can use joins to describe complicated systems.

Example 4.11 Random Point in a Circle We want to generate a point with integer coordinates at random from within a circle of radius 5, and we would like all the points to be chosen with equal weight. We will build an FRP to do this in two stages: X will be an FRP that represents the x -coordinate, and Y will be a conditional FRP that chooses the y -coordinate contingent on the x -coordinate. The wiring diagram for this system is



We will build this `frplib`. Let's start with what *seems* like the obvious solution, but which fails: choosing x - and y -coordinates with equal weights subject to the constraint that the join value lies within the circle.

```
pgd> X = frp(uniform(-5, -4, ..., 5))
pgd> @conditional_frp(domain=irange(-5, 5), target_dim=1)
...> def Y(x):
```

```

...>     y_kind = uniform(y for y in irange(-5, 5)
...>                               if y * y <= 5 * 5 - x * x)
...>     return frp(y_kind)
pgd> kind(X >> Y)

```

You will notice that this Kind does *not* put equal weight on all the values. Why not? Consider the two values $\langle 0, 0 \rangle$ and $\langle 5, 0 \rangle$. Look at the unfolded Kind tree with `unfolded(kind(X >> Y))` and reconstruct the weights in the canonical form. Does that suggest a better solution?

Let's define two small helper functions:

```

pgd> def y_points(x, r=5):
...>     "y-coordinates of points in a circle of radius r at given x"
...>     return [y for y in irange(-r, r) if y * y <= r * r - x * x]
pgd> num_y_points = compose(len, y_points) # len `after` y_points

```

Now, we can adjust our first approach slightly to solve our problem.

```

pgd> X = frp(weighted_by(-5, -4, ..., 5, weight_by=num_y_points))
pgd> @conditional_frp(domain=irange(-5, 5), target_dim=1)
...> def Y(x):
...>     return frp(uniform(y_points(x)))
pgd> XY_join = X >> Y
pgd> kind(XY_join)

```

Here, we give weight to x -coordinates in proportion to the number of points at that x -coordinate that lie inside the circle. The Kind (of size 81) has weights on all random points equal to $1/81$.

The join $X \triangleright Y$ is an FRP that represents a random choice of integer points inside a circle (of radius 5), where all points are chosen with equal weight.

We could construct this in another way:

```

pgd> points_inside_5 = [(x, y) for x in irange(-5, 5)
...>                               for y in irange(-5, 5)
...>                               if x * x + y * y <= 25]
pgd> XY_all = frp(uniform(points_inside_5))
pgd> Kind.equal( kind(XY_join), kind(XY_all) )
True

```


The two constructions – building with a join (`XY_join`) and specifying all components jointly (`XY_all`) – give the same results, but each has practical advantages in some circumstances. Joins break a problem into smaller pieces that are easier to describe, and they often scale well computationally. These advantages make joins our most common way to build more complex systems. However, modeling components jointly allows direct specification of the weights, which can be useful. For instance, we can easily give points weight that decreases with their distance from the origin with

```
pgd> point_wgts = [numeric_exp(-(x*x + y*y)) for x,y in points_inside_5]
pgd> XY_dist = frp(weighted_as(points_inside_5, weights=point_weights_5))
pgd> kind(XY_dist)
```

This can be done with joins but takes a bit more effort.

With either construction, we can predict the answers to questions about the random point. For instance: How far away is the point from the origin? (The statistic `Norm` computes the root sum of squares of the components.)

```
pgd> kind(Norm(XY_m))
,---- 0.012346 ---- 0
|---- 0.049383 ---- 1
|---- 0.049383 ---- 1.4142
|---- 0.049383 ---- 2
|---- 0.098765 ---- 2.2361
|---- 0.049383 ---- 2.8284
|---- 0.049383 ---- 3
<> -|
|---- 0.098765 ---- 3.1623
|---- 0.098765 ---- 3.6056
|---- 0.049383 ---- 4
|---- 0.098765 ---- 4.1231
|---- 0.049383 ---- 4.2426
|---- 0.098765 ---- 4.4721
`---- 0.14815 ----- 5
pgd> E(Norm(XY_m))
3.391780659838882
```

Or: Is X bigger than Y ?

```

pgd> compareXY = IfThenElse(Proj[1] > Proj[2], 1,
...>                               IfThenElse(Proj[1] < Proj[2], -1, 0))
pgd> kind(compareXY(XY_join))
      ,---- 0.45679 ----- -1
<> -+----- 0.086420 ---- 0
      `----- 0.45679 ----- 1

```

Observe that, as we might expect, X and Y are equally likely to be biggest.

Notice also that the operator form of joins is concise and convenient, but if we want to see the connections more explicitly, we could use a procedure instead, for instance

```

pgd> @frp
...> def XY_join():
...>     x = yield X
...>     y = yield Y(x)
...>     return (x, y)

```

You can load FRP factories for this example with

```
from frplib.examples.circle_points import circle_points
```

Then `circle_points()` returns a clone of `XY_join` (or of `XY_all` with the `method` argument), and also accepts a different circle radius. (The helpers `y_points`, `num_y_points`, and `points_inside`, which takes an optional radius, are also available.) We will use `circle_points` in ensuing examples.

Example 4.12 Random Lines

We now want to generate a random *line* formed by two random points within the circle of radius 5 generated as in the previous example. Our only constraint is that the two points not be the same, so the line is well defined.

```

pgd> First_Point = circle_points()
pgd> point_kind = kind(First_Point)
pgd> @conditional_frp(domain=points_inside_5, target_dim=2)
...> def Second_Point(first_point):
...>     not_same_point = (__ != first_point) # a statistic
...>     return frp(point_kind | not_same_point)
pgd> Line = First_Point >> Second_Point

```

```
pgd> Line
An FRP with value <1, -4, -4, 3>
```

The conditional FRP `Second_Point` takes a point as input and generates a random point with equal weight inside the circle *excluding the input point*. To do this, it defines a Boolean statistic (aka. a condition) that tests whether a given point is equal to the input point. It then uses the `|` operator, which is read as “given,” to impose a constraint: `point_kind | not_same_point` is a Kind like `point_kind` that excludes the input point. (We will discuss this in detail in the next Chapter.) The join generates the first point, passes it to the conditional FRP to generate the second point, and the result is the concatenation of the two points: a line from $\langle 1, -4 \rangle$ to $\langle -4, 3 \rangle$.

What can we say about how long the generated line is?

```
pgd> line_length = Norm(Proj[1,2] - Proj[3,4])
pgd> kind(line_length(Line))
,---- 0.043210 ----- 1
|---- 0.040741 ----- 1.4142
|---- 0.037654 ----- 2
|      ...           ...
<> -|
|      ...           ...
|---- 0.0024691 ---- 9.4868
|---- 0.0012346 ---- 9.8995
`---- 0.0018519 ---- 10
pgd> E(line_length(Line))
4.667555258539501
```

where some output has been omitted. (Look at it in the playground.) Notice how we build the statistic `line_length`. We take the first two components as a vector tuple and the last two as a vector tuple, subtract them as vectors (componentwise), and then take the length of the result. The resulting Kind has properties we would expect: a minimum length of 1, for adjacent points, and a maximum length of 10 along a diameter of the circle. Roughly speaking, longer lines are less likely because there are fewer ways to obtain them, and our best prediction of the line length ≈ 4.67 is slightly less than half the maximum.

Example 4.13 When Monty Cheats A Little

In section 4.2, we discussed the idea of independence using the Monty Hall game as an example. The original game was structured so that Monty's and your choices occur in isolation, independent by design. In this example, we investigate the idea of independence by seeing one way it can fail, particularly equation (4.3), via a version of the Monty Hall game in which Monty cheats a bit.

Suppose that Monty has arranged the game so that he can learn your choice of doors *before* he commits to his. Monty still makes a random choice, but the random mechanism he uses depends on the door you have chosen.

```
pgd> your_choice = uniform(1, 2, 3)
pgd> montys_choice = conditional_kind({
...>   1: choice(2, 3),
...>   2: constant(3),
...>   3: weighted_as(1, 2, 3, weights=[3, 6, 1])
...> })
pgd> game_outcome = (your_choice >> montys_choice) ^ Permute(2, 1)
```

Monty is a canny student of human behavior and tries to select doors that you are less likely to switch to based on the numbers, while making the DON'T SWITCH strategy a poor one overall.

The Kind of the game outcome joins the two stages, your choice followed by Monty's choice that depends on yours. The `Permute(2, 1)` transform swaps the order of the components so that it matches the order we used in the earlier examples.

```
pgd> G = frp(game_outcome)
pgd> M = G ^ Proj[1]
pgd> Y = G ^ Proj[2]
```

So Y and M are components of the Data FRP G , which we can write as $G = M :: Y$.

Notice that

```
pgd> m = kind(M)
pgd> m
,---- 3/30 ----- 1
<> -+---- 11/30 ---- 2
```

```

`---- 16/30 ---- 3

```

Tracking through the operations of join and transform, we can see that this is a weighted average of the Kinds in `montys_choice` over all three doors, with the weights of `your_choice` on the corresponding terms. This is illustrated in Figure 4.17, where “missing” values have been included in all the component Kinds with weight 0.

To see this, start with `kind(G)`:

```

pgd> game_outcome
,---- 5/30 ----- <1, 2>
|---- 5/30 ----- <1, 3>
|---- 10/30 ----- <2, 3>
<> -|
|---- 3/30 ----- <3, 1>
|---- 6/30 ----- <3, 2>
`---- 1/30 ----- <3, 3>

```

The projection transformation `game_outcome ^ Proj[1]` gives us `m`. For each choice d of Monty’s door, the weight in `m` is the sum of the weights on $\langle d, y \rangle$ over $y \in [1..3]$. But the weight on $\langle d, y \rangle$ equals the weight on y in `your_choice` times the weight on d in `montys_choice(y)`. In terms of Kernels, this is

```

sum(your_choice.kernel(y) * montys_choice(y).kernel(d) for y in [1, 2, 3])

```

We can do this weighted average simultaneously on all branches of the conditional Kind’s targets, which gives us Figure 4.17.

So, the Kind of M *aggregates* the possibilities over your choice, but once aggregated it cannot distinguish them. Compare `kind(G)` to `kind(M) ★ kind(Y)`:

```

pgd> kind(M) * kind(Y)
,---- 0.033333 ---- <1, 1>
|---- 0.12222 ---- <1, 2>
|---- 0.17778 ---- <1, 3>
|---- 0.033333 ---- <2, 1>
<> -+---- 0.12222 ---- <2, 2>
|---- 0.17778 ---- <2, 3>
|---- 0.033333 ---- <3, 1>

```

```
|---- 0.12222 ----- <3, 2>
^---- 0.17778 ----- <3, 3>
```

This Kind includes values that are *not possible* in this version of the game because your and Monty's choices are *coupled*. For example, if I know that you choose door 2, our prediction of Monty's choice is easy: he will always choose door 3. The two stages are not independent.

Puzzle 61. In the previous example, we defined the Kind of Y with equal weights on all three doors. Pick a Kind for Y with unequal weights and compute $\mathbf{m}$ as above. Show how $\mathbf{m}$ is a weighted average with the weights you specified analogously to Figure 4.17.

Puzzle 62. In the previous example, define

```
pgd> Y_c = frp(your_choice)           # A clone of Y above
pgd> C = conditional_frp(montys_choice)
```

Show how to define (a clone of) G with Y_c and using the Big 3+1 as appropriate. You can write Permute_{21} for the permutation statistic.

Example 4.14 Dependent Components

As another look at the difference between independence and dependence, define FRP M with Kind `uniform(1, 2, 3)` and consider the (inline) transformed FRP M^2 whose values are the squares of M 's value.

By our informal notions of independence, these two FRPs cannot be independent. Once you know M 's value, M^2 's value is uniquely determined. But without that knowledge M^2 has Kind

$$\langle \rangle \begin{cases} \frac{1}{3} & \langle 1 \rangle \\ \frac{1}{3} & \langle 4 \rangle \\ \frac{1}{3} & \langle 9 \rangle \end{cases}$$

Consider the system $M :: M^2$. We can express this as a join:

```
pgd> M = frp(uniform(1, 2, 3))
pgd> Sq = conditional_frp({
```

```

...>    1: frp(constant(1)),
...>    2: frp(constant(4)),
...>    3: frp(constant(9)),
...> })
pgd> Msq = M >> Sq ~ Proj[2]

```

We can also test independence with equation (4.3) by comparing $\text{kind}(M \triangleright \text{Sq})$ with $\text{kind}(M) \star \text{kind}(M^2)$.

```

pgd> kind(M >> Sq)
,---- 1/3 ---- <1, 1>
<> -+---- 1/3 ---- <2, 4>
    `---- 1/3 ---- <3, 9>
pgd> kind(M) * kind(Msq)
,---- 1/9 ---- <1, 1>
|---- 1/9 ---- <1, 4>
|---- 1/9 ---- <1, 9>
|---- 1/9 ---- <2, 1>
<> -+---- 1/9 ---- <2, 4>
    |---- 1/9 ---- <2, 9>
    |---- 1/9 ---- <3, 1>
    |---- 1/9 ---- <3, 4>
    `---- 1/9 ---- <3, 9>

```

The first Kind reflects the correct coupling between M and M^2 ; the second component is determined by the first. The second Kind cannot capture that coupling by the nature of the independent join operation. All 9 possibilities are represented. So, we cannot analyze the system $M :: M^2$ by analyzing the components separately; their relationship is essential to understanding the system's evolution and output.

Puzzle 63. Let X be an arbitrary FRP and C be a *constant* FRP with $\text{Kind } \text{constant}(v)$ for some value v .

Explain why, using either or both of our informal notions of independence, X and C are independent.

Thus, the join $M \gg \text{Sq}$ in the previous example is a clean join.

Hint: If you learn the value of X , what does that tell you about C ?

Example 4.15 Headphone Interface

I like my wireless headphones well enough, but they have only a single control button: push to play, push to stop. If I want to advance or rewind a track or episode, I need to double-click or triple-click that button. (If at the beginning of a track, a triple click moves to the beginning of the previous track; otherwise, it moves to the beginning of the current track.) Unfortunately, double or triple clicking a button in one's ear is highly unreliable because the earbud squishes around as the button is pressed.

I am listening to a track and want to repeat it from the beginning, so I attempt to triple click the control button. The clicks register *randomly* with the headphones as either a triple click, a double click and a click, a click and a double click, or three single clicks. This leaves me in one of several places: at the beginning of the target track, as desired, or paused at the beginning of the following track, or paused where I was. If paused, I immediately hit the button to continue. What is the chance that I will be able to rewind the target track to the beginning within four attempts? (After four attempts, I give up and grudgingly open my phone to restart the track directly.)

Let's build a system to model this situation and answer my question. We need to specify how we will represent (i) the random outcome after an attempted triple click, and (ii) the state of the system at each stage. For (i), we will use an encoding in 3-tuples where a 3 component means a triple click, a 2 means a double click, a 1 means a single click, and 0 is a placeholder that maintains a constant dimension for the tuples. This gives four possibilities: $\langle 3, 0, 0 \rangle$ for a triple click, $\langle 2, 1, 0 \rangle$ for a double click then a single click, $\langle 1, 2, 0 \rangle$ for a single click then a double click, and $\langle 1, 1, 1 \rangle$ for three single clicks.

For (ii), we will keep track of a tuple of the form $\langle t, a \rangle$ at each stage of the system's evolution. The first component t describes the current track can be any number in the set $\mathcal{T} = \{0, \frac{1}{2}, 1, 2, 3, 4\}$. The values 0-4 denote the beginning of successive tracks, with 0 meaning the target track I am listening to initially. The value $1/2$ means that the target track is in progress, but not at the very beginning. The second component a counts the number of triple-click attempts so far. It can be any value in $[0..4]$ Taken together, this tuple $\langle t, a \rangle$ comprises the *state* of the system, and each time we attempt a triple click, the state is randomly updated. We *start* in state $\langle \frac{1}{2}, 0 \rangle$, with the target track in progress but not at the beginning and we end as soon as either $t = 0$ or $a = 4$. So, our goal for the system to end in one of the states $\langle 0, a \rangle$ for some $a \in [0..4]$.

For example, if we make an attempt that registers as three single clicks, we move from state $\langle \frac{1}{2}, 0 \rangle$ to $\langle \frac{1}{2}, 1 \rangle$, requiring another click to unpause. If on the next attempt, we register a triple click, we move from state $\langle \frac{1}{2}, 1 \rangle$ to $\langle 0, 2 \rangle$, and the process ends successfully.

Each attempted click generates a random outcome that determines how the state changes. We will model this two operations: a conditional FRP that takes the current state as input and generates an outcome for the click attempt and a statistic that transforms the joined state-outcome into a new state.

Based on my experience, assume that the outcome of an attempted triple click has Kind k_1 given by

$$\begin{array}{l} \langle \rangle \text{ --- } 0.10 \text{ --- } \langle 1, 1, 1 \rangle \\ \quad \quad \quad \text{--- } 0.35 \text{ --- } \langle 1, 2, 0 \rangle \\ \quad \quad \quad \text{--- } 0.35 \text{ --- } \langle 2, 1, 0 \rangle \\ \quad \quad \quad \text{--- } 0.20 \text{ --- } \langle 3, 0, 0 \rangle \end{array}$$

We will also use the constant Kind k_0

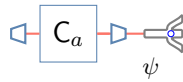
$$\langle \rangle \text{ --- } 1 \text{ --- } \langle 0, 0, 0 \rangle$$

for FRPs that always produce the value $\langle 0, 0, 0 \rangle$.

Define the FRP S_0 to represent the initial state of the system. Because the initial state is fixed, $\text{kind}(S_0) = \text{constant}(\langle \frac{1}{2}, 0 \rangle)$.

For each $a \in [1..4]$, we define a conditional FRP C_a to represent the a th click attempt. For attempt $a \in [1..4]$, C_a has type $2 \rightarrow 5$. It accepts inputs $\langle 0, b \rangle$ for $0 \leq b < a$ and $\langle t, a-1 \rangle$ for $t \in \mathcal{T}$ and $0 < t < a$. The former occurs when we have successfully rewound on the b th attempt, and the latter occurs in other possible states after $a-1$ attempts. For inputs $\langle t, a \rangle$ where $t = 0$, the target of C_a has Kind k_0 . For inputs $\langle t, a \rangle$ where $t > 0$, the target of C_a has Kind k_1 .

For $a > 0$, we then transform the output of S_a with the statistic ψ whose job is to take the current state and the output of the attempt and return the next state. The wiring for the a th stage looks like:

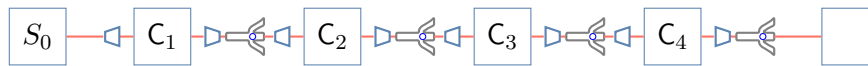


Here, ψ is defined for valid $\langle t, a \rangle$ with $t > 0$ by

$$\begin{aligned}\psi(0, a, x, y, z) &= \langle 0, a \rangle \\ \psi(t, a, 3, 0, 0) &= \langle \lceil t \rceil - 1, a + 1 \rangle \\ \psi(t, a, 2, 1, 0) &= \langle \lfloor t \rfloor + 1, a + 1 \rangle \\ \psi(t, a, 1, 2, 0) &= \langle \lfloor t \rfloor + 1, a + 1 \rangle \\ \psi(t, a, 1, 1, 1) &= \langle t, a + 1 \rangle,\end{aligned}$$

where ceiling $\lceil t \rceil$ is the smallest integer $\geq t$ and floor $\lfloor t \rfloor$ is the greatest integer $\leq t$. The first two components are the current state, and the last three components are the outcome of the attempt. If we reach $t = 0$, we stop counting attempts, so the system is in the proper state at the end.

The transformed conditional FRPs, the pieces shown in the wiring above, have type $2 \rightarrow 2$. They take in the current state and output the random next state. Taken together, our system is wired as follows:



The output values $\langle t, a \rangle$ either have $t = 0$ if I am able to rewind successfully or $t > 0$ and $a = 4$ otherwise.

Now, let's build this in the `frplib`. We start by defining with the Kinds k_0 and k_1 and the initial state FRP S_0 :

```
pgd> k_0 = constant(0, 0, 0)
pgd> k_1 = weighted_as((3, 0, 0), (2, 1, 0), (1, 2, 0), (1, 1, 1),
...>                  weights=[0.2, 0.35, 0.35, 0.1])
pgd> S_0 = frp(constant('1/2', 0))
```

Here, `constant` is the factory for the constant Kind with only the given value, and `weighted_as` creates a Kind with arbitrary values and specified weights. Note that we can give fractional quantities as strings; for some fractions (though not $1/2$), this gives a more accurate numerical representation internally.

We define the conditional FRPs C_a for $a \in [1..4]$ in two steps: the base mapping common to all four conditional FRPs and a factory function that enforces the requirements on the inputs.

```
pgd> @conditional_frp(codim=2, target_dim=3)
```

```

...> def C_base(v):
...>     t, a = v
...>     if t == 0:
...>         return frp(k_0)
...>     return frp(k_1)

pgd> def C_(a):
...>     "Returns the conditional FRP C_a for a in 1..4."
...>     assert a in set([1, 2, 3, 4])
...>     domain_C = ( [(0, b) for b in range(a)] +
...>                  [(t, a - 1) for t in [0.5, 1, 2, 3, 4] if t < a] )
...>
...>     return conditional_frp(C_base, domain=domain_C)
pgd> C_(1)
A conditional FRP as a function with domain={(0.5, 0), (0, 0)}
pgd> C_(2)
A conditional FRP as a function with domain={(0, 1), (0.5, 1), (1, 1), (0, 0)}

```

We can get away without the factory, but the domain constraints do help us avoid mistakes. And we need *distinct* conditional FRPs for each of the four stages.

Next, we define the statistic ψ :

```

pgd> @statistic(codim=5, dim=2)
...> def psi(v):
...>     match v:
...>         case (0, a, _, _, _):
...>             return (0, a)
...>         case (t, a, 3, 0, 0):
...>             return (numeric_ceil(t) - 1, a + 1)
...>         case (t, a, 2, 1, 0) | (t, a, 1, 2, 0):
...>             return (numeric_floor(t) - 1, a + 1)
...>         case (t, a, 1, 1, 1):
...>             return (t, a + 1)
...>         raise MismatchedDomain(f'Improper input {v} to psi')

```

For Python versions before 3.10, `match` is unsupported, so you can use explicit if-then-else statements.

Finally, we put this all together into an FRP that represents the entire system.

We write this as a simple factory to return a fresh FRP:

```
pgd> @frp_factory
...> def try_rewind():
...>     "trying to rewind the current track."
...>     R = S_0
...>     for a in irange(1, 4):
...>         R = R >> C_(a) ^ psi
...>     return S
pgd> try_rewind()
An FRP with value <3, 4>
pgd> try_rewind()
An FRP with value <0, 2>
```

The loop in `try_rewind` exactly expresses our wiring diagram above. We feed S_0 into C_1 and through ψ , and the value of the resulting FRP becomes the input for the next stage. On the first try here, we give up; on the second, we succeed on the second attempt.

Note that we need not write a factory like `try_rewind`, as there are several alternatives. For instance, we could express our wiring diagram in a single expression, with a few extra parentheses to avoid ambiguity:

```
pgd> (((S_0 >> C_(1) ^ psi) >> C_(2) ^ psi) >> C_(3) ^ psi) >> C_(4) ^ psi
An FRP with value <0, 4>
```

The parentheses are needed because the $\wedge$ operator has lower precedence than $\gg$; without them, `psi` would group improperly with the following `C_(a)`. Or we could use a procedure to combine all these steps, which is even clearer. We can write the procedure in either of two equivalent ways, as we prefer, that combines all the join and transform steps:

<pre>@frp def Rewind_Attempted(): t_0 = yield S_0 s_1 = yield C_(1)(t_0) s_2 = yield C_(2)(psi(s_1)) s_3 = yield C_(3)(psi(s_2)) s_4 = yield C_(4)(psi(s_3))</pre>	<pre>@frp def Rewind_Attempted(): t_0 = yield S_0 s_n = yield C_(1)(t_0) for n in irange(2, 4): t_n = psi(s_n) s_n = yield C_(n)(t_n)</pre>

```
return psi(s_4)
```

```
return psi(s_n)
```

We can compute the Kind of this FRP to assess our chances:

```
pgd> Succeed = (Proj[1] == 0)
pgd> kind(try_rewind())
,---- 0.2 ----- <0, 1>
|---- 0.16 ----- <0, 2>
|---- 0.03 ----- <0, 3>
|---- 0.0240 ----- <0, 4>
<> -+---- 0.0001 ----- <1/2, 4>
|---- 0.04200 ----- <1, 4>
|---- 0.16660 ----- <2, 4>
|---- 0.13720 ----- <3, 4>
`---- 0.24010 ----- <4, 4>
pgd> kind(try_rewind() ^ Succeed)
,---- 0.5860 ---- 0
<> -|
`---- 0.414 ----- 1
```

The statistic `Proj[1] == 0` tests whether we end up at the beginning of the target track, so the second Kind tells us we have roughly 41% chance of succeeding. Put another way, computing our prediction:

```
pgd> E(try_rewind() ^ (Proj[1] == 0))
0.414
```

We can also ask where we will end up after this try.

```
pgd> kind(Proj[1](try_rewind()))
,---- 0.414 ----- 0
|---- 0.0001 ----- 1/2
|---- 0.042 ----- 1
<> -|
|---- 0.1666 ----- 2
|---- 0.1372 ----- 3
`---- 0.2401 ----- 4
```

So, we are likely to end up several tracks ahead.

Because these are all clean joins – we have independent click attempt outcomes at every stage – we can apply equation (4.17) to see the connection between the *structure* of the system and our *analysis* of it. For example, if $s_0 = \text{kind}(S_0)$ and $c_ = \text{compose}(\text{kind}, C_)$ are the Kind and conditional Kind of S_0 and the C_a ’s respectively, then we have FRP and Kind given by exactly parallel expressions

```
((S_0 >> C_(1) ^ psi) >> C_(2) ^ psi) >> C_(3) ^ psi >> C_(4) ^ psi
((s_0 >> c_(1) ^ psi) >> c_(2) ^ psi) >> c_(3) ^ psi >> c_(4) ^ psi
```

and procedures

<pre>@frp def Rewind_Attempted(): t_0 = yield S_0 s_1 = yield C_(1)(t_0) s_2 = yield C_(2)(psi(s_1)) s_3 = yield C_(3)(psi(s_2)) s_4 = yield C_(4)(psi(s_3)) return psi(s_4)</pre>	<pre>@kind def rewind_attempted(): t_0 = yield s_0 s_1 = yield c_(1)(t_0) s_2 = yield c_(2)(psi(s_1)) s_3 = yield c_(3)(psi(s_2)) s_4 = yield c_(4)(psi(s_3)) return psi(s_4)</pre>

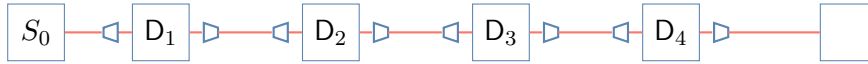
We use the same operations in the same order to build the system and to analyze it, operating on FRPs in the former case and Kinds in the latter. So all our thinking about how to structure the system serves to structure our analysis as well.

It is worth noting that we can transform conditional FRPs and conditional Kinds by statistics as we can with regular FRPs and Kinds. If C and k are a conditional FRP and conditional Kind with compatible statistic ψ , then we write $\psi(C)$ or $C \wedge \psi$ and $\psi(k)$ or $k \wedge \psi$ for the transformation of the joined targets (i.e., includes the input). This works in `frplib` as well, and you can also use the `.transform_targets` method to transform by a statistic that only takes the target values. So for instance, in the previous example, we could define

```
def D_(a):
    "Returns the transformed conditional FRP psi(C_a) for a in 1..4."
    assert a in set([1, 2, 3, 4])
    domain_C = ( [(0, b) for b in range(a)] +
                 [(t, a - 1) for t in [0.5, 1, 2, 3, 4] if t < a] )
```

```
return conditional_frp(C_base, domain=domain_C) ^ psi
```

The resulting conditional FRPs take in the 2-dimensional state and the targets have the next state as their value (codimension 2, target dimension 2). This absorbs the statistic adapter into the conditional FRPs giving an adjusted wiring diagram



With this transformation, both the join expression and the procedure become simpler, with the algebraic expression

```
S_0 >> D_(1) >> D_(2) >> D_(3) >> D_(4)
```

and the procedure

```
@frp
def Rewind_Attempted():
    t_0 = yield S_0
    t_1 = yield D_(1)(t_0)
    t_2 = yield D_(2)(t_1)
    t_3 = yield D_(3)(t_2)
    t_4 = yield D_(4)(t_3)
    return t_4
```

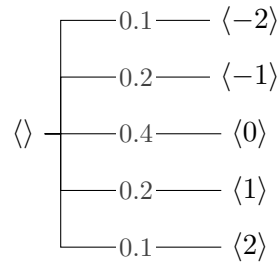
where we extract the target values (the next state) from each conditional FRP target for the current state.

Example 4.16 The Drunken Sailor

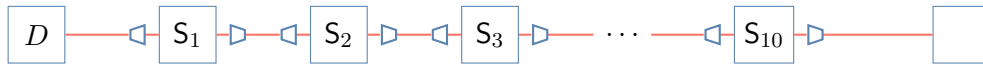
A drunken sailor stands on a dock facing the gangway taking him to his ship. If he moves five steps forward, he makes it onto the ramp and, however unsteadily, to safety on the ship. If he moves five steps backward, he falls off the dock into the ocean and ends up sleeping on the beach as his ship leaves without him. Assume that he moves forward (1) or backward (-1) at random and independently at each step, where his direction has Kind

$$\langle \rangle \begin{cases} \frac{3}{5} & \text{---} \langle -1 \rangle \\ \frac{2}{5} & \text{---} \langle 1 \rangle \end{cases}$$

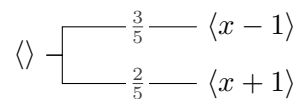
Assume the dock is 9 steps long and that we label his positions on the dock by integers in $[-4..4]$. When the sailor stumbles down to the dock, he initially moves onto a starting position randomly, with Kind



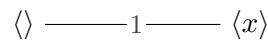
What is the chance that he makes it to the ship within 10 lurching steps? To reasonable approximation, what is the chance that he eventually makes it to the ship without falling into the ocean? We can visualize this system with the wiring diagram



where we can extend with more conditional FRPs as desired. Here, D is an FRP representing the sailor's starting position on the dock, with the Kind above. The conditional FRPs S_i represent the sailor's steps and all have the same structure. They map a position x in $[-4..4]$ to an FRP with Kind



and a position x in $\{-5, 5\}$ to a constant FRP with Kind



Let's compute the answers to our question in `frplib`.

```
pgd> move_kind = choice(-1, 1, '2/3')
pgd> D = frp(weighted_as(-2, -1, ..., 2, weights=[0.1, 0.2, 0.4, 0.2, 0.1]))
```

We define a single conditional FRP S and set `auto_clone` to make the copies $S_1, S_2, \dots, S_{10}$. We first define a Boolean function that tests for valid inputs; giving this function as the `domain` argument to `conditional_frp` raises an error with any input for which this function returns False.


```

pgd> def is_valid_path(v):
...>     valid_positions = set(irange(-5, 5))
...>     return all(x in valid_positions for x in v)

pgd> @conditional_frp(domain=is_valid_path, target_dim=1, auto_clone=True)
...> def S(path):
...>     x = path[-1]                                # most recent position
...>     if x == -5 or x == 5:                        # boat or ocean...
...>         return frp(constant(x))                 # ...stay there
...>     return frp(move_kind ^ (x + 1))             # move left or right

```

Following the wiring diagram, we start with D and successively join with a copy of S in a loop. The final value of W is an FRP representing the sailor's path over 10 steps.

```

pgd> W = D
pgd> for _ in range(10):
...>     W = W >> S
pgd> W
An FRP with value <0, 1, 0, -1, -2, -3, -4, -5, -5, -5>
pgd> clone(W)
An FRP with value <0, -1, -2, -3, -4, -5, -5, -5, -5, -5>
pgd> clone(W)
An FRP with value <2, 1, 0, 1, 2, 3, 4, 3, 4, 5>
pgd> clone(W)
An FRP with value <1, 0, -1, 0, -1, 0, -1, 0, 1, 0>
pgd> ten_steps = kind(W ^ Proj[-1])

```

We can see on running the system anew several times that sometimes the sailor falls in the ocean, sometimes reaches the boat, and sometimes neither within 10 steps. We can view the Kind of W easily, but since it has size 3902, you should look at it on your terminal. Instead, we define the Kind `ten_steps`, which looks at the sailor's state at the end only, which tells us what we need to know.

More interestingly, we can predict the answer our various questions about the sailor's trajectory. Does the sailor reach the boat or the ocean in 10 steps?

```

pgd> ten_steps ^ Or(Proj[-1] == 5, Proj[-1] == -5)
,---- 0.65354 ---- 0

```

```

<> -|
    `---- 0.34646 ---- 1
pgd> which_end_point = Cases({-5: -5, 5: 5}, default=0)(Proj[-1])
pgd> which_end_point(ten_steps)
    ,---- 0.29466 ----- -5
<> -+----- 0.65354 ----- 0
    `---- 0.051806 ---- 5

```

Here, we use the *statistic combinator* `Or` to compute the logical-or of two statistics. (For internal Python reasons, we cannot use Python's `or` keyword in this context.) We see that the sailor has only about a 0.35 chance of reaching an end point in 10 steps, and is about 6 times as likely for that endpoint to be ocean rather than boat if he does.

There is nothing magical here about 10 steps. We can extend this further, for example to sixteen steps, but because we are tracking the sailor's entire path, the number of possibilities – and thus the size of the Kind – gets large quickly:

```

pgd> sixteen_steps = kind(W >> S >> S >> S >> S >> S >> S)
pgd> size(sixteen_steps)
185502
pgd> sixteen_steps ^ Or(Proj[-1] == 5, Proj[-1] == -5))
    ,---- 0.43221 ---- 0
<> -|
    `---- 0.56779 ---- 1

```

This leaves a relatively high chance that the situation will be unresolved. We want to simulate enough steps that to good approximation, we can assume he will reach one endpoint or another.

Fortunately, for many questions, we do not need the sailor's entire path, only selected information about it. We can follow the approach used in Example 4.15, where we transform the output of the conditional FRP by a statistic that keeps the dimension fixed and tracks the information we care about. (We adapt this approach to the next example as well.)

We will keep track of the sailor's current position x and the number of *steps* n he has made. But if the sailor reaches ocean (-5) or boat (5), the value stays as is. We tweak our conditional FRPs to take a tuple $\langle x, n \rangle$ as input; their values will

look like $\langle x, n, x', n' \rangle$ where x' is the sailor's next position and n' is an updated count.

```
pgd> @conditional_frp(target_dim=2, auto_clone=True)
...> def S2(v):
...>     x, n = v
...>     if x == -5 or x == 5:
...>         return frp(constant(x, n))
...>     return frp(move_kind ^ Fork(__ + x, n + 1))
```

We extend D 's value with an initial number of steps (0) as the sailor's initial state. We get the sailor's next state ($\langle x', n' \rangle$) from the FRP by applying a projection to extract the last two components of the value. We package this in a function that has the maximum number of steps as a parameter:

```
pgd> def sailors_walk(up_to_step):
...>     W2 = D ^ Fork(Id, 0)
...>     for _ in range(up_to_step):
...>         W2 = W2 >> S2 ^ Proj[3,4]
...>     return W2

pgd> kind(sailors_walk(16)) ^ Or(Proj[1] == 5, Proj[1] == -5)
,---- 0.43221 ---- 0
<> -|
`---- 0.56779 ---- 1
```

which is just what we got above but about a thousand times faster.

This lets us compute the outcome for longer walks in which the sailor is essentially certain to reach one of the endpoints.

```
pgd> kind(sailors_walk(200)) ^ Or(Proj[1] == 5, Proj[1] == -5)
,---- 9.9261E-7 ---- 0
<> -|
`---- 1.00000 ----- 1
```

So to good approximation, the sailor's will reach an endpoint within the first 200 steps.

```
pgd> many_steps = kind(sailors_walk(200))
```

```

pgd> many_steps ~ IfThenElse(Abs(Proj[1]) < 5, 0, Proj[1])
      ,---- 0.86986 ----- -5
<> -+----- 9.9261E-7 ----- 0
      `---- 0.13014 ----- 5

```

So to good approximation, the sailor is about 7.5 more likely to end up in the ocean than on the boat.

We can predict how many steps it will take the sailor to reach either end:

```

pgd> E(many_steps ~ Proj[2])
18.49313503683753

```

Here, $\text{many\_steps} \sim \text{Proj}[2]$ (or equivalently $\text{Proj}[2](\text{many\_steps})$) is the Kind of the second component of the FRPs value, the number of steps taken. This answer is a bit too high because it includes the very small chance remaining that the situation is not resolved. In the next section, we will see how to impose an observational constraint on the output with the $|$ operator, which is read “given”.

```

pgd> E((many_steps | (Abs(Proj[1]) >= 5)) ~ Proj[2])
18.49295487075902

```

The condition after the given $|$ is the constraint we require. *Given* that the sailor reaches the boat or the ocean, how many steps does it take? The prediction is only very slightly different, as expected. Later, we will also see how to use these Kinds to solve this problem exactly.

GENERAL JOINS IN TERMS OF KERNELS. The join operation on conditional Kinds has a nice expression in terms of trees. We attach a copy of the target Kinds in the conditional Kind to the node in the source Kind tree corresponding to the input value, and propagate the input value through to the leaf nodes.

This operation combined with ALGORITHM CANONICAL from Chapter 3 gives canonical weights that have a nice form. This is easy to see, and often useful, expressed in terms of kernels.²⁷

²⁷See Definition 2.

If k is a Kind and m is a compatible conditional Kind, then $k \triangleright m$ has possible values $u :: v$, where u is a possible value of k and v is a possible value of the target $m(u)$. (If m has type $n \rightarrow p$, then u has dimension n and v has dimension $p - n$.) By the nature of the join operation (attaching target Kind trees at the nodes of the source Kind), the canonical weight on $u :: v$ can be found by multiplying the canonical

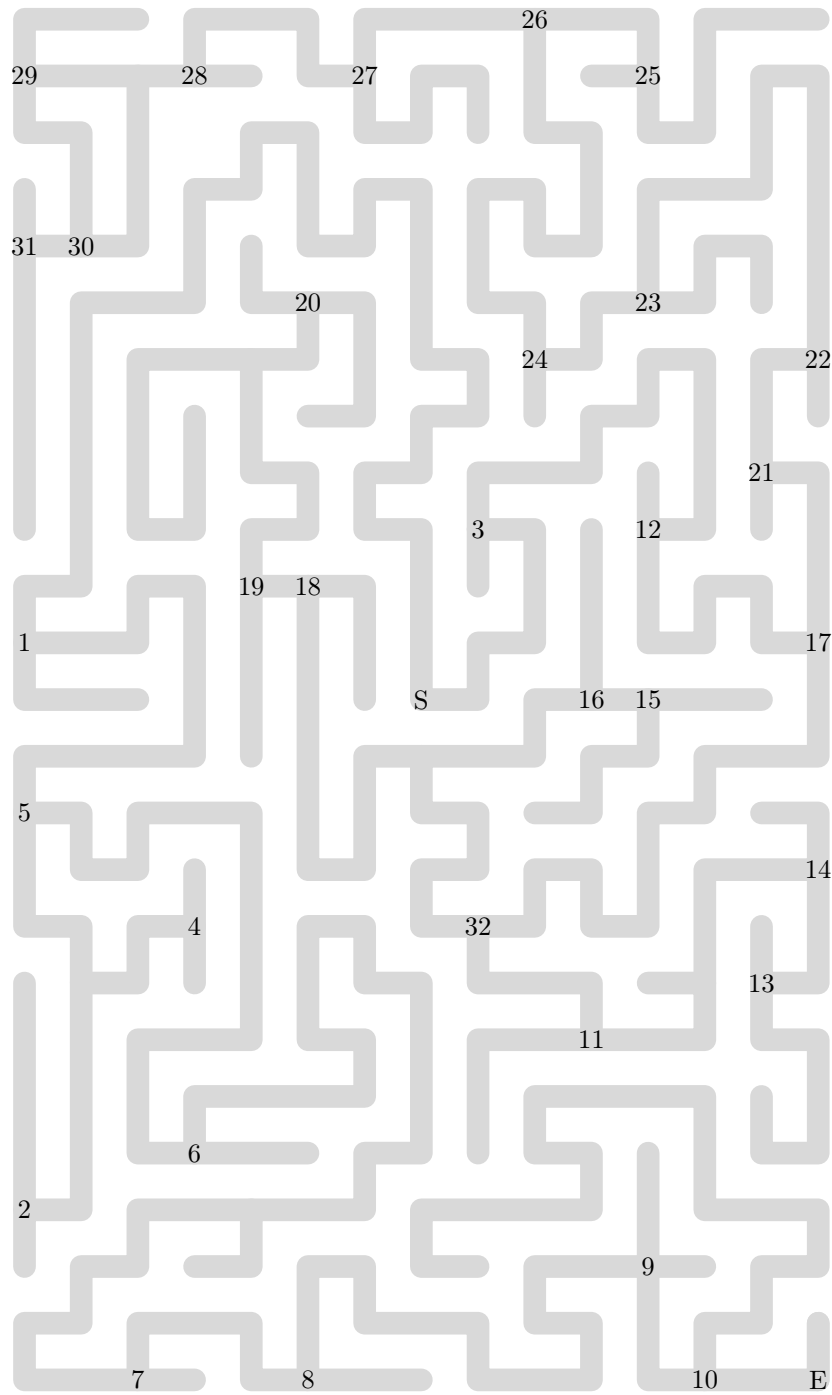


FIGURE 4.18. Another labyrinth that has ensnared poor Theseus. His starting point ($S=0$) and exit (E) are marked, and each juncture is assigned a number. The FRPs will generate a number corresponding to the juncture that Theseus wanders into.

weights of u in k and of v in $\mathbf{m}(u)$. That is, we join the trees and multiply (canonical) weights along the path from root to leaf.

In terms of kernels, this looks like $(k \triangleright \mathbf{m})(u :: v) = k(u) \mathbf{m}(v \mid u)$. Notice that if u is not possible for k or v is not possible for the target $\mathbf{m}(u)$, the Kernel will return 0 as it should.

The Kernel of a Join. If k is a Kind and $\mathbf{m}$ is a compatible conditional Kind, then the joined Kind $k \triangleright \mathbf{m}$ has kernel

$$(k \triangleright \mathbf{m})(u :: v) = k(u) \mathbf{m}(v \mid u). \quad (4.18)$$

We can extend this idea to joins of arbitrarily many stages. Let k and $\mathbf{m}_i$ for $i \in [1..n]$ be compatible. Then any appropriate value can be decomposed into possible values for subsequent targets, and we get the kernel:

$$\begin{aligned} (k \triangleright \mathbf{m}_1 \triangleright \mathbf{m}_2 \triangleright \cdots \triangleright \mathbf{m}_n)(v_0 :: v_1 :: v_2 :: \cdots :: v_n) \\ = k(v_0) \mathbf{m}_1(v_1 \mid v_0) \mathbf{m}_2(v_2 \mid v_0 :: v_1) \cdots \mathbf{m}_n(v_n \mid v_0 :: v_1 :: \cdots :: v_{n-1}). \end{aligned} \quad (4.19)$$

The value $v_0 :: v_1 :: \cdots :: v_n$ identifies a path through the (partially) unfolded tree, and the canonical weight of that value is the *product of the normalized weights on the path from root to leaf*.

Input	Value	Weight
u_1	v_{11}	w_{11}
	v_{12}	w_{12}
	$\vdots$	$\vdots$
u_1	v_{2s}	w_{2s}
u_2	v_{21}	w_{21}
	$\vdots$	$\vdots$
	v_{2s}	w_{2s}
u_2	v_{2s}	w_{2s}
$\vdots$	$\vdots$	$\vdots$
u_m	v_{m1}	w_{m1}
	$\vdots$	$\vdots$
	v_{ms}	w_{ms}

TABLE 4.1. A conditional Kind as a table. Each row gives one combination of input value, target Kind value, and canonical weight.

GENERAL JOINS IN TERMS OF TABLES. We can also describe the join operation on conditional Kinds in terms of tables. We saw earlier (see page 19) that a Kind in canonical form can be expressed as a table with a column for values and a column for weights. This generalizes to conditional Kinds. If r is a conditional Kind with valid input values $u_1, \dots, u_m$, we can express c with the table shown in Table 4.1. Because the inputs are distinct and the target values for each input are distinct, every row of the table is unique. If we restrict our attention to the part of the table with one particular input value and focus on the value and weight columns, we get the table for the corresponding target Kind.

The join operation $r \triangleright s$, creates a new table from the tables for r and s . Each row of $r \triangleright s$ is derived from (a) a row of r , and (b) a row of s for which r 's Value entry equals s 's Input entry, if one exists. The derived row is

Input	Value	Weight
$r.\text{Input}$	$r.\text{Value} :: s.\text{Value}$	$r.\text{Weight} * s.\text{Weight}$

where we write $r.\text{Input}$, $s.\text{Value}$ and so on to denote the corresponding field of the selected rows (a) for r or (b) for s .

In Python, this looks roughly like

```
r_join_s = Table()
for r_row in r:
    for s_row in s:
        if r_row.Value != s_row.Input:
            continue # Skip non matching s rows
        r_join_s.append_row(
            Input=r_row.Input,
            Value=concat(r_row.Value, s_row.Value),
            Weight=r_row.Weight * s_row.Weight
        )
```

This is the *join* operation used in relational databases. The SQL language expresses this operation even more concisely

```
SELECT (r.Input, r.Value :: s.Value, r.Weight * s.Weight)
FROM r JOIN s where r.Value = s.Input;
```

As described above, we select a row from r and a matching row from s and for the new row from fields of the selected rows, when they exist.

Example 4.17 Back to the Labyrinth

Theseus has awoken from a night of revelry to find himself trapped in a labyrinth ... again (Figure 4.18). With both the excess of honey mead and the lack of Ariadne's help, he is not at his best, and he wanders about at random from his starting position, looking for the exit.

Because our FRPs generate lists of numbers, our first step is to assign a number to each relevant outcome. Here, the key information is which junctures in the labyrinth Theseus visits. So we assign a unique number to each juncture, as shown in the Figure.

When Theseus stands at juncture 17, for example, he has three choices (move to junctures 4, 18, 19), and in his stupor he chooses from among them randomly with equal weight (`uniform(4, 18, 19)` in the `frplib`). The same applies at every juncture, beginning with the starting point 0.

Open the playground and follow along. We start by creating the Kind of the FRP for Theseus's starting position. He begins at juncture 0 with certainty, so this is a constant.

```
pgd> start = constant(0)
```

Then, we import some data about the labyrinth so that you do not have to type it all in. The variable `labyrinth` contains a dictionary mapping each juncture to the junctures Theseus can reach from it. Take a look at its value and see how it corresponds to the Figure.

```
pgd> from frplib.examples.labyrinth import *  
pgd> labyrinth
```

Notice that juncture 33 is the exit, and even in his diminished capacity, Theseus will take the exit when he gets there. So, `labyrinth[33] = [33]` to reflect that he exits when he reaches juncture 33.

We want to use `labyrinth` to generate joins, and this is easy to do. We start by creating the Kinds for Theseus's moves at each juncture. For each "item" in the labyrinth – a juncture and its list of neighbors – we associate with that juncture a Kind that gives equal weight to every neighbor (via the `uniform` factory). The call to the `conditional_kind` factory gives `steps` some useful properties and makes it print out nicely.

In Python, this is called a *dictionary comprehension*.


```
pgd> steps = conditional_kind({
...>     juncture: uniform(neighbors)
...>     for juncture, neighbors in labyrinth.items()
...> })
pgd> moves = from_latest(steps)
```

Specifically, `steps` maps each juncture number to the Kind for a move out of that juncture. Look at its value in the playground. The function `from_latest` is defined in the `frplib.examples.labyrinth` module converts the conditional Kind `steps` that takes as input a single juncture to a conditional Kind that accepts as input a path of any length. As the name suggests, the latter uses only the latest juncture in path, so `moves` takes a *tuple* describing Theseus's path so far and uses `steps` to make a move based on the last juncture in the path.

Consider the FRPs describing Theseus's path after one, two, and three moves. Take a moment and think about how to get their Kinds from `start` and `moves`.

Puzzle 64. We could generate a table of FRPs at each juncture like

```
pgd> steps_at = conditional_frp({
...>     juncture: frp(kind)
...>     for juncture, kind in steps.items()
...> })
```

Why isn't this sufficient to simulate Theseus's trip through the maze? Hint: Once you push the button on an FRP, can the value change?

Given a path to any juncture, `moves` returns the Kind for a move *from* that juncture, so that will go on the right-hand side of `>>`. This yields

```
pgd> start                                # starting position
pgd> start >> moves                        # after one move
pgd> start >> moves >> moves              # after two moves
pgd> start >> moves >> moves >> moves     # after three moves
```

The `>>` operator is left-associative, so without parentheses, an expression like the last line groups from the left automatically:

```
((start >> moves) >> moves) >> moves
```

Look at these Kinds. At each leaf, we see – for that particular random outcome – Theseus’s entire path through the maze so far. The results show the Kind trees in canonical form; you can always use `unfold` to see the full tree, e.g., `unfolded(start >> moves)`, though these can get big.

Puzzle 65. How would you find the Kind of the FRP describing Theseus’s path through $n \geq 0$ moves? Write or sketch a function `n_moves` that takes an initial Kind and n and a conditional Kind like `moves` and returns the corresponding Kind when you call `n_moves(start, n, moves)`.

For large numbers of moves, the Kind trees get large because the FRP’s value can reflect many possible paths. Of course, in Theseus’s besotted state, he is not thinking too clearly, and he is making moves that ignore his previous path. So, we will use a trick to make things more manageable: we will look at the Kind of his *most recent move* only. That is, we will create an FRP that answers the question *at what juncture is Theseus after 100 moves?*

The function `after_move_n` has been loaded into your playground to help with this. Let’s simulate Theseus’s 100th move, passing `steps` (not `moves`).

```
pgd> move100_kind = after_move_n(100, start, steps)
pgd> frp(move100_kind).value
pgd> FRP.sample(1000, frp(move100_kind))
```

The first line gives the Kind of an FRP that records just Theseus’s 100th move. The second line gives an FRP with that Kind and pushes the button. The third line generates 1000 FRPs with that Kind and summarizes the result.

Puzzle 66. Use a sample of FRPs to estimate how likely Theseus is to have exited the labyrinth after 10 moves, 50 moves, 100 moves, and 1000 moves.

A key feature of a join is that its value includes the value of both the source and the target. In the previous examples, joins can construct an FRP (and its associated Kind) that represents the random process’s entire evolution up to some point. In Example 4.17, for instance, the function `n_moves` forms the join Kind

```
start >> moves >> moves >> ... >> moves
```

that describes Theseus’s entire path from the start through a specified number of moves. In Example 4.16,

```
D >> S >> clone(S) >> ... >> clone(S)
```

is the join FRP that describes the sailor's history of lurching steps. Joins build parts into a whole, describing the joint evolution of the parts as a random system.

But in many practical cases, we do not need to the entire history in our analysis, and as will be seen in the next section, often we only need to pay attention to the *most recent* state of the process to predict the next state. Indeed, this is what we did in each of the last three examples. The statistic ψ in Example 4.15 updated the track and number of attempts, dropping other information. The Proj [3,4] in `sailors_walk` of Example 4.16 saves only the sailor's current position and number of steps. And in `theseus_latest`, we use a projection to extract Theseus's last position (which is used in `after_move_n`) because Theseus's next position only depends on where he is, not on how he got there.

Suppose we wanted to know the Kind of Theseus's *second* position in the labyrinth without knowing his first position. We could do this in two steps (though one expression in the playground) with

```
pgd> second_pos_kind = (start >> moves)[2]
```

The first step forms the *join* of his first position and the conditional Kind of his second position given his first position. The second step is to transform with a projection that extracts only the second position, dropping the first.

If we wanted to know the Kind of Theseus's *third* position in the labyrinth without knowing his second position, we use the same operation.

```
pgd> third_pos_kind = (second_pos_kind >> moves)[2]
```

Again, we form the join of the second position and the third position *given* the second position and then use a projection to extract the third position.

Joins build combined FRPs/Kinds, and projection statistics extract *marginals*, the FRPs/Kinds of selected components. Thus:

```
pgd> combined = start >> moves >> moves
pgd> Kind.equal( combined[1], start )
True
pgd> Kind.equal( combined[2], second_pos_kind )
True
pgd> Kind.equal( combined[3], third_pos_kind )
True
```

As we have seen, this is a common pattern, a special case of the *data-question* pattern of Figure 2.8: build a combined system as a join of several pieces and then extract marginals that relate to our questions. Call this the *join-marginal* pattern.

The Kinds representing Theseus’s positions²⁸ are instances of the join-marginal pattern with two pieces: his current position and a description of his next position that depends explicitly on his current position. This special case of the join-marginal pattern is common and powerful enough to have its own name: *conditioning pattern*.

To understand the conditioning pattern, we consider two FRPs X and Y along with their Kinds. X has dimension m and represents the early-stage output of a random system. Y has dimension n and represents the later-stage output of the system that is contingent on the value of X . This transition from early to later stage is expressed by a conditional FRP of type $m \rightarrow m + n$ that is compatible with X , call it S for the moment, such that $X \triangleright S = X :: Y$. We can think of this conditional FRP as representing Y when we are *given* X ’s value, so we can denote it with the evocative but unusual name $Y | X$ to emphasize that meaning.²⁹ (We parenthesize $(Y | X)$ in expressions if there is any ambiguity to clarify that it is a single object.) Its conditional Kind is $\text{kind}(Y | X)$, and we write its kernel as $k_{Y|X}(v | u)$ as a synonym for $s(v | u)$.

²⁸The Kind of Theseus is something else entirely, where we replace branches successively with identical looking ones over time and ask if it is still the same Kind.

²⁹As usual, $|$ is read “given.”

Definition 27. The Conditional FRP $Y | X$. Let X be an FRP and S be a compatible conditional FRP with conditional Kind s .

When the joined FRP $X \triangleright S$ has the form $X :: Y$ for some FRP Y , we often write S as $(Y | X)$ to emphasize that it determines the value of the later stage of the system (Y) given the value of the earlier stage (X).

Then, $\text{kind}(Y | X)$ is just the conditional Kind s , and we can write its kernel $s(v | u)$ as $k_{Y|X}(v | u)$.

When working with FRPs X and Y derived from a join where the conditional FRP is implicit, or at least in the background, this notation is often clearer and more convenient.

Define statistics to extract the components from the full system output:

$$\text{early} = \text{proj}_{1..m} \quad (4.20)$$

$$\text{later} = \text{proj}_{(m+1)..(m+n)} \cdot \quad (4.21)$$

The first extracts the output of the early stages of the system’s evolution, and the

second extracts the output of the later stages. So, we have:

$$X = \text{early}(X \triangleright (Y \mid X)) \quad (4.22)$$

$$Y = \text{later}(X \triangleright (Y \mid X)) \quad (4.23)$$

$$\text{kind}(Y) = \text{later}(\text{kind}(X) \triangleright \text{kind}(Y \mid X)). \quad (4.24)$$

This is just the join-marginal pattern. But next, we shift perspectives, in two steps.

First, use the right-hand sides of these equations to define the **conditioning operator** $\parallel$ (written `//` in `frplib`) by

$$(Y \mid X) \parallel X := \text{later}(X \triangleright (Y \mid X)) \quad (4.25)$$

$$\text{kind}(Y \mid X) \parallel \text{kind}(X) := \text{later}(\text{kind}(X) \triangleright \text{kind}(Y \mid X)), \quad (4.26)$$

where $:=$ emphasizes that this is a definition. This conditioning operator is just a join followed by a projection. Equations (4.23) and (4.24) become

$$\begin{aligned} Y &= (Y \mid X) \parallel X \\ \text{kind}(Y) &= \text{kind}(Y \mid X) \parallel \text{kind}(X). \end{aligned}$$

The conditional FRP or Kind goes on the left side of the operator because it is the focus of the operation, and the FRP or Kind of the quantity it depends on goes on the right side. This is the opposite of how we write the join because we are emphasizing the extracted quantity Y . In the Theseus example, `second_pos_kind` and `third_pos_kind` can be written, respectively, as

```
move // start           # same as second_pos_kind
move // second_pos_kind # same as third_pos_kind
```

We specify the update mechanism (`moves`) first and the state to update (e.g., `start` or `second_pos_kind`) second.

Second, if we focus on equation (4.24) above and consider how the join operation works, we can view this combination of join and projection as a *weighted-averaging* operation: $\text{kind}(Y)$ is a weighted average of the Kinds produced by $\text{kind}(Y \mid X)$ using the weights in $\text{kind}(X)$. This is illustrated in Figure 4.19.

In the top panel of the Figure, we *condition on*³⁰ the middle Kind, producing the Kind on the right-hand side. This is just join followed by a projection. Look at these in the playground.

```
pgd> x = choice(0, 1, weight_ratio=1/2)
```

³⁰We are using “condition on” as a verb here to mean applying the conditioning operation.

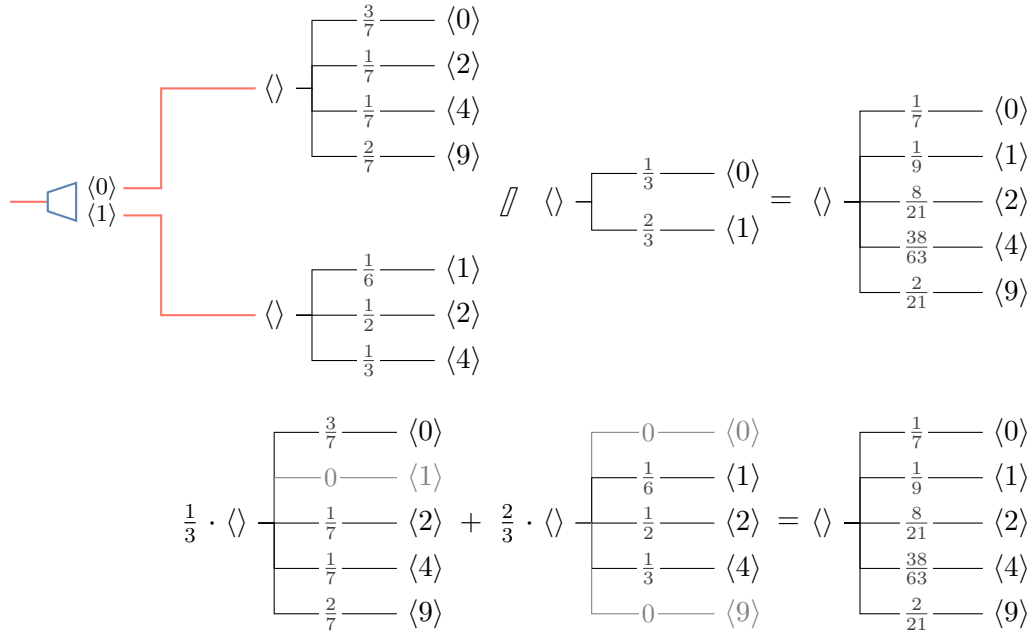


FIGURE 4.19. The conditioning operation as a weighted average over a conditional Kind by a Kind. The top panel combines a conditional Kind and a Kind with the conditioning operator $//$, producing the Kind at the right. The bottom shows how that Kind can be viewed as an average of Kinds.

```
pgd> y_given_x = conditional_kind({
...>    0: weighted_as(0, 2, 4, 9, weights=[3, 1, 1, 2]),
...>    1: weighted_as(1, 2, 4,    weights=[1, 3, 2]),
...> })
```

The conditional Kind on the left of the top panel is `y_given_x`, and the Kind in the middle of the top panel is `x`. We can compute the right-hand side in two equivalent ways:

```
pgd> (x >> y_given_x)[2]
pgd> y_given_x // x
```

The output is omitted here, but you should try it. In particular, compare this Kind `x >> y_given_x` and make sure you see how you get the former from the latter.

The bottom panel of the Figure gives us a different view of what this operation is doing. We first take each of the Kinds produced by `y_given_x` and add some fake, zero-weight branches to each so that both have the same set of values. We then average the weights of these Kinds, branch by branch, using the *corresponding weights*

from $\mathbf{x}$. For instance, $\frac{1}{3} \cdot \frac{3}{7} + \frac{2}{3} \cdot 0 = \frac{1}{7}$ and $\frac{1}{3} \cdot \frac{1}{7} + \frac{2}{3} \cdot \frac{1}{3} = \frac{38}{63}$. This averaging of Kinds gives us the same result.

The operation of conditioning is just an instance of the join-marginal pattern. We join, then project. But *conceptually*, we can view the conditioning operation as a way of finding the Kind of a random quantity by averaging over Kinds for the quantity that are specified conditionally on another random quantity. Each term in the average is weighted according to the Kind of that second quantity.

In Theseus’s case, `moves` specifies the Kind of his *next* position conditionally on his *current* position. If you know the Kind for his current position, you can find the Kind of his next position: `next = moves // current`. The `//` (and `/`) operator knows the dimension of the object on its right side and automatically tailors the projection to the dimension of the object on the left, which makes it convenient in `frplib`. For both Kinds or FRPs, the conditioning pattern looks like

$$y = y\_given\_x // x$$

The term on the left side is a conditional specification of y in terms of the quantity described by x . Later, we will call this the “method of hypotheticals” because it can find the Kind/FRP y without actually observing the other quantity.

Definition 28. Suppose that X and Y are FRPs of dimension m and n , respectively, that are related by a join via

$$Y = \text{proj}_{(m+1)..(m+n)} (X \triangleright (Y | X))$$

for some conditional FRP $Y | X$ of type $m \rightarrow (m + n)$.

Then, equations (4.25) and (4.26) define the **conditioning** operator `//`, called the “conditioning” operator, which is written `//` in `frplib`.

The conditioning operator satisfies

$$Y = (Y | X) // X \tag{4.27}$$

$$\text{kind}(Y) = \text{kind}(Y | X) // \text{kind}(X). \tag{4.28}$$

We say here that we have computed Y (or its Kind) by “conditioning on” X (or its Kind). Equation (4.28) in particular says that we can compute the Kind of Y by combining a conditional Kind for Y ’s given X with the Kind of X .

Example 4.18. We flip a balanced coin. If it comes up tails, you roll a balanced six-sided die and take its value. If it comes up heads, you roll two balanced six-sided dice and take their maximum. If Y is an FRP whose payoff represents the value you take from this system, find $\text{kind}(Y)$ by conditioning.

Let X be the FRP representing the coin flip. We have a description of how Y depends on X and of X , so, we can compute $\text{kind}(Y)$ by conditioning on X

```
pgd> x = choice(0, 1)    # kind(X)
pgd> y_given_x = conditional_kind({
...>    0: uniform(1, 2, ..., 6), 1: Max(uniform(1, 2, ..., 6) ** 2)
...> })
```

A conditional Kind with wiring:

,---- 1/6 ---- 1	,---- 1/36 ----- 1
---- 1/6 ---- 2	---- 3/36 ----- 2
---- 1/6 ---- 3	---- 5/36 ----- 3
<0>: <> -	<1>: <> -
---- 1/6 ---- 4	---- 7/36 ----- 4
---- 1/6 ---- 5	---- 9/36 ----- 5
`---- 1/6 ---- 6	`---- 11/36 ----- 6

```
pgd> y = y_given_x // x    # kind(Y) by conditioning on x
,---- 0.097222 ---- 1
|---- 0.12500 ----- 2
|---- 0.15278 ----- 3  # weight = (1/2)*(1/6) + (1/2)*(5/36)
<> -|
|---- 0.18056 ----- 4  # weight = (1/2)*(1/6) + (1/2)*(7/36)
|---- 0.20833 ----- 5
`---- 0.23611 ----- 6  # weight = (1/2)*(1/6) + (1/2)*(11/36)
```

Solutions to Selected Puzzles.

Puzzle 42. (i) A and B are not independent. If you knew, for instance, that $B == 0$, then it changes your knowledge and predictions about A which must then have a value of the form $\langle 0, j \rangle$ for $j \in [0..5]$. (ii) Looking at the transforms here, we see that V equals $\text{Sum}(A)$ and W equal $\text{Diff}(A)$. Knowing V can tell us something about W . For instance, if we knew that $V == 10$ or $V == 0$, then W must be 0. Because knowledge of V changes our predictions about W , they are not independent. (iii) Similar logic applies here. If we knew that $C == 0$ then

$V \leq 5$ or if we knew that $V == 10$, then $C == 5$. They are not independent. (iv) K is always 2, our predictions of its value are perfect. Knowing the values of the other FRPs does not change that. So K is independent of any other FRP in the list. (v) Z is a fresh FRP with two possible values. None of the other FRPs in the list are related to it in any way nor do their values offer any information about its value. Z is independent of all the other FRPs.

Note that for two FRPs to be independent, **no observation** of one of their values can change our predictions about the other's value. Finding any information that would change our predictions is thus enough to show that they are not independent. That is why it's enough to show that knowing, say, that $V == 10$ changes our predictions about C or W .

It turns out that B and C are independent, for reasons we will see fully in the next Chapter. The short story though is that the observation of B 's value, does not change the Kind of C which is `uniform(0, 1, ..., 5)`, and vice versa.

Puzzle 46. Start with the case $d_1 = 1 = d_2$. Define $\psi(\langle x, y \rangle) = \langle y, x \rangle$. Notice that ψ is its own inverse function, i.e., $\psi = \psi^{-1}$. When $X \star Y$ has value $\langle x, y \rangle$, then so does $\psi(Y \star X)$. The statistic simply swaps the order of components in the independent join, which do not otherwise interact.

More generally, define the statistic ψ with codimension $d_1 + d_2$ so that $\psi(y :: x) = x :: y$ when x and y have respective dimensions d_1 and d_2 . This is invertible and the inverse ψ^{-1} swaps $x :: y$ to $y :: x$. The same logic then holds: $\psi(Y \star X)$ equals $X \star Y$ and $\psi^{-1}(X \star Y)$ equals $Y \star X$ because the values are swapped accordingly and the two components are determined separately.

Puzzle 47. We extract the attribute value from the character, convert it to an ability score, generate a 20-sided die roll, and combine the ability and roll with an independent join, so we can act on them both. The sum of the two must be bigger than the threshold for the saving throw to succeed.

```
@frp_factory
def saving_throw(character: FRP, attribute: Statistic, thresh: int):
    """the outcome of a saving throw for a specific character,
       attribute, and threshold
    """
    roll_d20 = frp(uniform(1, 2, ..., 20))
    ability = character ^ attribute ^ Floor((__ - 10) / 2)
```

```

data = ability * roll_d20

return data ^ (Sum >= thresh)

```

Puzzle 55. Independent joins are a special case: for any independent join, we can build it using the general join operation with proper choice of conditional FRP or Kind. For conditional Kinds s , we get an independent join with a *constant function* $s = \text{const}_k$ that always returns a Kind k . In this case, for a Kind r , $r \triangleright s = r \star k$. For conditional FRPs S , we get an independent join if S always returns an FRP with the same Kind. In particular, if $S = \text{const}_T$ for an FRP T , then for an FRP R , we have $R \triangleright S = R \star T$. So we choose $U = \text{const}_Y$.

The constant function $\text{const}_c(x) = c$ is defined and discussed in Chapter 10 of Interlude F.

Puzzle 56. We make a join from the strength attribute that introduces the exceptional strength, which is 0 except for a fighter with 18 strength.

```

@conditional_frp(domain=irange(3, 18), target_dim=1)
def extraStrength(strength):
    "Conditional kind for a fighter's exceptional strength."
    if strength == 18:
        return frp(uniform(1, 2, ..., 100))
    return frp(constant(0))

def dnd_character(fighter=False):
    "Returns an FRP representing a D&D character's attribute scores."
    # Strength with exceptional strength
    if fighter:
        S = dnd_attribute() >> clone(extraStrength)
    else:
        S = dnd_attribute() * frp(constant(0))
    I = dnd_attribute()    # Intelligence
    W = dnd_attribute()    # Wisdom
    Co = dnd_attribute()   # Constitution
    D = dnd_attribute()    # Dexterity
    Ch = dnd_attribute()   # Charisma

    return S * I * W * Co * D * Ch

```

We use `clone` so we get fresh FRPs each call for the exceptional strength.

Puzzle 60. We follow the template given:

```
@conditional_kind(domain=lambda v: all(x in {0, 1} for x in v))
def roll(flips):
    heads = sum(flips)
    if heads == 0:
        return constant(0) # roll0
    return Max(uniform(1, 2, ..., 6) ** heads) # roll1 or roll2
```

Try entering `roll(0, 0)`, `roll(0, 1)`, and `roll(1, 1)` to check this.

Puzzle 64. As Theseus is wandering around the labyrinth, it is possible – even likely – that he will revisit the same juncture more than once. Each FRP has a single fixed value but Theseus makes a separate decision each time he visits. So more than one FRP per juncture may be needed.

Puzzle 65. We need to keep updating by mixing with `moves` n times, as follows:

```
def n_moves(start, n, moves):
    current = start
    for _ in range(n):
        current = current >> moves
    return current
```

Keep in mind though that the number of paths grows exponentially with n , so the tree gets *very big* rather quickly. We will see a way around that later.

Puzzle 66. For each $n = 10, 50, 100, 1000$, we do something like this

```
exit = 33
iter = 10000
nth = FRP.sample(iter, after_move_n(n, start, steps))
len([juncture for juncture in nth if juncture == exit])/iter
```

This computes the proportion of samples in which Theseus has reached the exit by move n . This works because once he reaches the exit, the FRP will always return that value.

Key Take Aways

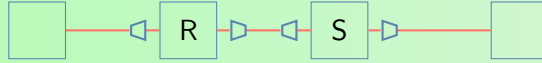
A general **join** connects a source representing the output of all earlier stages of a system's evolution to one or more targets that represent the output of the next stage. With conditional FRPs packaging the targets and generalizing regular FRPs, we can define a general join operation on conditional FRPs.

If R and S are conditional FRPs, they are **compatible** if the joined outputs of R valid inputs to S . When compatible, we can form their join $R \triangleright S$.

If R has type $m \rightarrow p$ and S has type $p \rightarrow n$, then $R \triangleright S$ has type $m \rightarrow n$. For valid input v to R , the output of $R \triangleright S$ is the FRP with value $v :: w :: x$ where

1. w is the value of the activated target $R(v)$, and
2. x is the value of the activated target $S(v :: w)$.

The wiring diagram for a join is



The procedures for the FRP special case and the general case are

FRP $R \triangleright S$:

$w \leftarrow R$

$x \leftarrow S(w)$

return $w :: x$

conditional FRP $(R \triangleright S)(v)$:

$w \leftarrow R(v)$

$x \leftarrow S(v :: w)$

return $v :: w :: x$

We can also define the join of two conditional Kinds, r and s . If r has type $m \rightarrow p$ and s has type $p \rightarrow n$ and they are compatible, then $r \triangleright s$ has type $m \rightarrow n$.

For any valid input v to r and for each value w of $r(v)$, we attach the tree $s(v :: w)$ to the w node of r and propagate $v :: w$ by prepending it to the nodes in the subtree. The procedures for Kinds are analogous:

Kind $r \triangleright s$:

$w \leftarrow r$

$x \leftarrow s(w)$

return $w :: x$

conditional Kind $(r \triangleright s)(v)$:

$w \leftarrow r(v)$

$x \leftarrow s(v :: w)$

return $v :: w :: x$

The **conditioning** operation $C \parallel K$ is a combination of the join $K \triangleright C$ and a projection that extracts the value of the second stage.

Chapter Checkpoints

After reading this Chapter you should be able to:

- Explain the role of the join operation among the Big 3+1.
- Explain how to construct independent join of FRPs and of Kinds and what such joins mean.
- Describe what distinguishes an independent join from a more general join.
- Describe *conditional FRPs* and *conditional Kinds*, how to wire them together to form a join, and how to create them in `frplib`.
- Define the type, codimension, target dimension, and dimension of a (conditional) FRP or Kind.
- Explain why an ordinary FRP or Kind can be consider a special case of a conditional FRP or Kind.
- Explain how to construct a general join between conditional FRPs or between conditional Kinds. In particular, given the tree from a Kind and the trees from the Kinds returned by a conditional Kind, show how to find the join Kind.
- Find the dimension, size, and values of the Kind $k \triangleright m$ from the properties of k and m .
- Recover X from $X \triangleright M$ or k from $k \triangleright m$ by applying an appropriate statistic.
- Define a clean join.
- Calculate the Kind of a join from the join of the Kinds.

Constraining with Observations

5

Chapter

Contents

5.1	Accounting for Partial Information	326
5.2	Events	333
5.3	Observations	337
5.4	Bayes's Rule	361
5.5	Observations and Independence	369
5.6	Observations and Conditional Kinds	376

5.1 Accounting for Partial Information

You run into Alice and Bob at the FRP Marketplace, and Alice is agitated. She had ordered a sizeable batch of b FRPs, $A_{[1]}, A_{[2]}, \dots, A_{[b]}$, all clones with the Kind shown in Figure 5.1. The administrator of the Marketplace tried to rescind the order *after* payment and payoffs were made. Now the lawyers are involved, and things are messy.

Alice's order was specified by a contract that stipulates: (1) for every $i \in [1 \dots b]$, Alice will receive payoff¹ $P_{[i]} = \text{proj}_3(A_{[i]})$ from unit i , and (2) her price per unit is set to $-\$0.825$. This price per unit is the risk-neutral price for the FRPs representing the payoffs, that is $\mathbb{E}(P_{[i]}) = -0.825$. That this price is negative means the Marketplace pays Alice $\$0.825b$ to take ownership of the b FRPs.

¹Positive payoff means that the Marketplace pays Alice; Negative payoff means that Alice pays the Marketplace.

The problem arose through a coincidence and some well-intentioned initiative by the Marketplace staff. Because Alice had been a regular customer, the staff assumed the deal would go through and pressed the buttons on the FRPs $A_{[1]}, \dots, A_{[b]}$, *before* Alice had committed to the deal. By chance, Bob happened to be wandering through the facility at that time got a look at the displays of the FRPs as the buttons were pressed. From Bob's vantage point, he could see only the *second* component of the tuple displayed by each FRP $A_{[i]}$. He quickly tabulated his observations: out of 10,000 FRPs, he saw 4850 0's and 5150 1's. Bob shared this information with Alice,² and Alice proceeded with the order at the original price, keeping to herself that she had extra information.

²Bob has a good recall, or a fast camera!

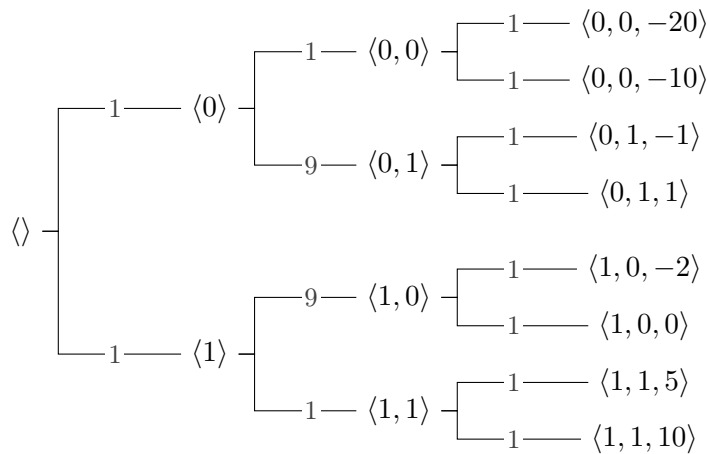


FIGURE 5.1. The Kind for the FRPs in Alice's order, *before* she learned Bob's information.

Alice could have withdrawn from the deal at that point, but she chose to go forward with the original price. Why? And why did the Marketplace administrator

try to rescind the order? The answer to both questions stems from the way Bob's *observations* – partial information about the values of the FRPs in the batch – change our knowledge and predictions about Alice's payoffs.

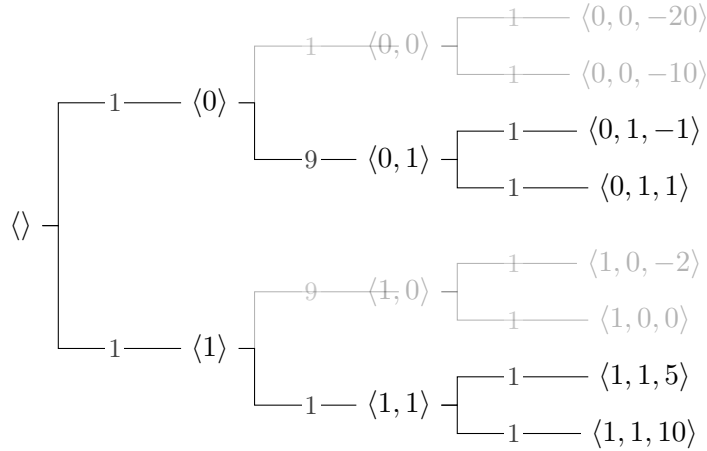


FIGURE 5.2. The Kind from the previous Figure, highlighting the paths in the tree that are consistent (black) and inconsistent (gray) with the observation that the second component of the value equals 1.

To start, we will focus on a single FRP among the batch $A_{[1]}, \dots, A_{[b]}$, call it A , and imagine that Alice obtains partial information from Bob about A 's value. Before Alice learns Bob's observation, her knowledge about A 's value is embodied in the Kind shown in Figure 5.1. It tells us the possible values of A and the relative propensities of these values to occur. *After* Alice learns Bob's observation, her knowledge has changed because the observation precludes some of the FRPs possible values. Suppose, for example, that Bob had observed that A 's value has second component 1. This observation implies that A 's value cannot be any of the original possibilities with a 0 in the second component. The branches of the Kind tree corresponding to the precluded values are *logically inconsistent* with the observation, with what we know to be true. So we eliminate them from consideration, dropping all branches for which the second component of the final tuple is not 1. When we exclude those inconsistent branches, we get the Kind tree in Figure 5.2. This is the Kind updated to reflect *Alice's knowledge of A 's value **given** the observation that A 's second component is 1.*

This process of eliminating inconsistent branches to account for observations is perhaps even clearer if we first reduce the original Kind to canonical form. before eliminating the inconsistent possibilities, as shown in Figure 5.3. Both figures highlight the paths that are consistent with the observation. We simply *drop the branches* for which the second component is not equal to 1. A quick rescaling (useful but not

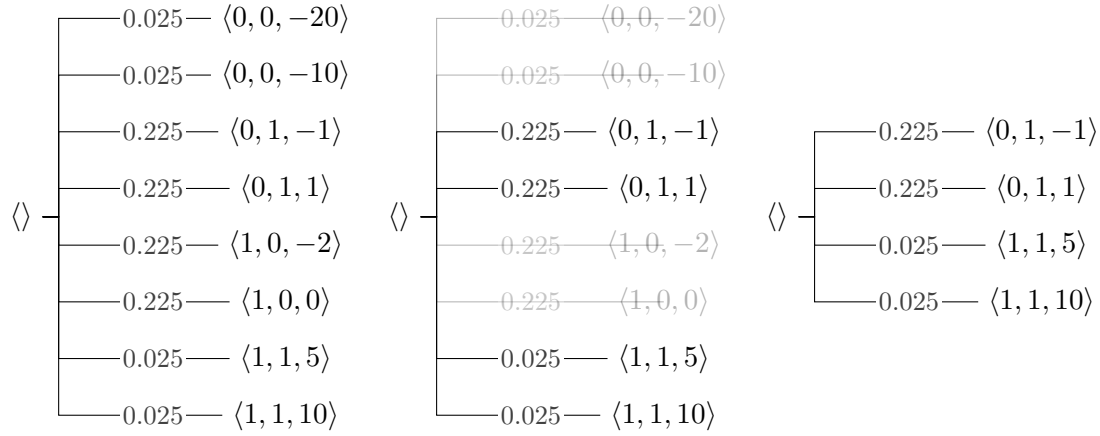


FIGURE 5.3. (Left) The canonical form of the Kind in Figure 5.1. (Center) The Kind highlighting the values that are consistent (black) and inconsistent (gray) with the observation that the second component of the value equals 1. (Right) The updated Kind with inconsistent branches eliminated.

required) returns us to canonical form, giving the Kind in the bottom panel of Figure 5.4. This is the canonical Kind of the FRP that Alice has after the *observation* that the second component is 1.

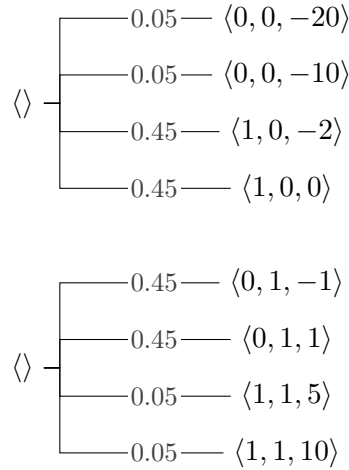


FIGURE 5.4. The *effective* Kind of Alice's FRPs conditional on the information that the second component of the value equals 0 (top) and 1 (bottom).

We can do this same process of identifying and eliminating branches that are inconsistent with the observation that the second component equals 0. Try to write down the highlighted tree analogous to Figure 5.2 for this case. If we reduce to canonical form, we get the Kind in the top panel of Figure 5.4.

Using Observations. To update a Kind with an observation, we eliminate all branches with leaf nodes that are inconsistent with the observation.

One way to express these observations is through a *condition*, a Boolean statistic. For instance, the condition $(\text{proj}_2(\blacksquare) = 1)$ is true when the second component of its argument value is 1 and false otherwise. A 's value has second component 1 if and only if condition $(\text{proj}_2(\blacksquare) = 1)$ is true when given A 's value. This observation can thus be expressed by the condition $(\text{proj}_2(\blacksquare) = 1)$ itself. We write $A \mid (\text{proj}_2(\blacksquare) = 1)$ for the FRP A *constrained by the observation* that its second component equals 1. This $\mid$ bar is read “given” and is an operator. On the left side of the $\mid$ is an FRP or Kind, and on the right side is an expression for an observation. In this case, the observation means that the specified condition is true for the value of the FRP A .

Importantly: A and $A \mid (\text{proj}_2(\blacksquare) = 1)$ are **distinct (but related) FRPs**. They have different Kinds because we have different knowledge about their values. The Kind of the former, $\text{kind}(A)$, is that shown in Figure 5.1 and on the left of Figure 5.3 in canonical form. The Kind of the latter, $\text{kind}(A \mid (\text{proj}_2(\blacksquare) = 1))$ is the Kind of A *updated with the observation* that the condition is true (i.e., that the FRP's value has second component 1). We write this as $\text{kind}(A) \mid (\text{proj}_2(\blacksquare) = 1)$ where the $\mid$ operator (again read as “given”) acts on the $\text{kind}(A)$ to give the Kind in Figure 5.2 and on the right of Figure 5.3 in canonical form. We get $\text{kind}(A) \mid (\text{proj}_2(\blacksquare) = 1)$ from $\text{kind}(A)$ by eliminating branches that are inconsistent with the observation. This is the essence of how we incorporate prior information into our predictions. (While it is often useful to renormalize this to canonical form, it is not necessary nor always better.)

To get an intuitive feel for how Bob's observation changes Alice's expectation of payoff, we will use `frplib` to calculate our predictions under various scenarios. The original Kind in Figure 5.1 is bound to the variable `kindA` in the `frplib` example module `frplib.examples.deal_for_alice`. You can import this with

```
from frplib.examples.deal_for_alice import kindA
```

to use it in the playground or in your code.

In `frplib`, we can write the condition $(\text{proj}_2(\blacksquare) = 1)$ as `(Proj[2] == 1)`. This uses the statistic combinator expressions described in Section 2.1 and produces the condition that checks whether “the second component of the input equals 1.” Thus, the updated Kind for A with this observation is

```
kindA | (Proj[2] == 1)
```

Note that the parentheses are required here because the $\mid$ operator has low precedence in Python. Entering this in the playground will show the Kind in canonical form as

in the bottom panel of Figure 5.4. We can see the unfolded and original forms of this Kind by entering

```
pgd> unfolded(kindA | (Proj[2] == 1))
pgd> raw(kindA | (Proj[2] == 1))
```

We can also create an FRP A and confirm that the updated Kind equals the Kind of $A \mid (\text{proj}_2(\blacksquare) = 1)$:

```
pgd> A = frp(kindA)
pgd> Kind.equal( kind(A | (Proj[2] == 1)), kind(A) | (Proj[2] == 1) )
True
```

It is important that A be fresh for this comparison to make sense.

Try these steps also with the complementary observation that the second component is 0.

Puzzle 67. In `frplib`, find the Kind of A updated for the observation that the second component of the value equals 0. Keeping A fresh, confirm that you get the same Kind for both `kind(A | (??))` and `kind(A) | (??)`, where you should replace `??` here with the relevant condition.

Partial answer: `(0 == [2]Proj) | kindA`.

These two observations are mutually exclusive because the second component of the FRP will be *either* 0 or 1. So you might wonder how we can talk about both observations when only one can be true. The answer is that when we apply observations to update *Kinds*, we can use both real and hypothetical observations. Constraining with hypothetical observations allows us to ask questions like: “how does my knowledge of this FRPs value update if I were to observe this particular outcome?” So we can ask what happens to Alice’s knowledge and predictions about one of these FRPs for each possible observation of the second component. We can compare her payoffs in those two situations and apply that knowledge to understand the outcome of her order.

A simple way to assess the impact of each observation (or none) is to calculate Alice’s best prediction of the payoff in each situation. This prediction is computed via the risk-neutral price, or *expectation*, which, as we will see in Chapter 6, is expressed with the $\mathbb{E}$ operator in mathematics and the `E` operator in `frplib`.

If A is one of the FRPs $A_{[i]}$ for $i \in [1..b]$, then the amount Alice receives as payoff is `proj3(A)`, the third component of A ’s value. Without any special knowledge of A ’s value, Alice’s payoff has Kind

```
kindA ~ Proj[3]
```

With the observation of the second component, her payoff has one of the Kinds

```
(kindA | (Proj[2] == 0)) ~ Proj[3]
(kindA | (Proj[2] == 1)) ~ Proj[3]
```

depending on which second component is observed. The corresponding risk-neutral prices are

```
pgd> E(kindA ~ Proj[3])
-33/40
pgd> E((kindA | (Proj[2] == 0)) ~ Proj[3])
-12/5
pgd> E((kindA | (Proj[2] == 1)) ~ Proj[3])
3/4
```

Without any special knowledge, each FRP in the batch has risk-neutral price $-33/40 = -0.825$. This is the average value of the third component over an arbitrarily large batch of clones, and this is the price per unit that Alice pays in her deal.³ On average, the payoff to Alice will be negative, so the contract mandates that the FRP Marketplace pays her per unit to balance that average loss. The deal is in that sense “fair.”

³Negative prices mean that Alice gets money for each unit.

But with the observation of the second component, her prediction changes. If she observes that the second component is 0, the predicted payoff is even more negative -2.4 , which is worse for Alice. But if she observes that the second component is 1, the predicted payoff is positive, 0.75 .

Bob’s observation tells Alice that the batch of FRPs in her order contains 4850 with Kind shown in the top panel of Figure 5.4 and 5150 FRPs with Kind shown in the bottom panel of Figure 5.4 So, Alice can expect a payoff around

$$4850 \cdot -2.4 + 5150 \cdot 0.75 = 10,000 \cdot -0.77775.$$

The payoff is still negative, but remember that Alice receives $\$0.825$ per unit under the order contract, so her net earnings will be around

$$10,000 \cdot (0.825 - 0.77775) = 472.5$$

dollars, a small but positive gain for Alice. This is the reason that she proceeded with the order given the information she had learned. On the other hand, the Marketplace

felt it was unfair that Alice could choose whether to proceed with the order in a way that was guaranteed to eliminate her possible losses.

Puzzle 68. In `frplib`, build a `Kind kA` that is equal to `kindA`, and use either inspection or `Kind.equal(kA, kindA)` to check that you succeeded.

We can simulate Alice’s payoff from the FRPs she ordered with

```
P = Sum(frp(kindA[3]) ** 10_000)
```

Explain briefly what this means.

In evaluating the Marketplace manager’s motives for trying to cancel the deal, we might wonder: “Is \$472.50 an unusually large payout for the original deal?”

Use `FRP.sample` with `P` and the statistic `(__ + 8250 >= 472.50)` to address the question. (One or two hundred samples should be sufficient to get an idea.)

When Bob told Alice the second component of her FRPs, that information allowed her to update her predictions about those FRPs’ values. It is common in practice to obtain partial information about uncertain quantities as we observe a random system evolve, and it is often useful to determine how our predictions (and actions) would change *if* we had particular information. The purpose of **observations** is to account for partial information about the value of FRPs, whether actual or hypothetical. If we have an FRP and observe something about its value, our predictions may change, and we want to adjust our calculations, decisions, and actions accordingly.

In this Chapter, we formalize what this means, how we represent and denote the constraints from an observation, and how we use them to update our analysis. We will introduce observations in an expression with `|`, the given operator. This `|` bar is read “given.” On the left side of the `|` is the object that we are analyzing, on the right side is a specification of observation.

Key Take Aways

When we make *observations* of a random system while it is running, we obtain partial information about the value of an FRP. This information changes what we know about the FRPs value and thus changes our prediction about that value.

Constraining an FRP with an observation produces a new, related FRP. The

Kind of the constrained FRP *updates* the Kind of the original FRP to account for the information obtained in the observation.

To update a Kind with an observation, we convert it to canonical form then **eliminate all branches whose values are inconsistent with the observation**. We usually convert the constrained Kind into canonical form, but this is optional.

5.2 Events

At its most basic, making an observation of a random system means observing and recording something that happens as the system runs. More specifically, it means that we learn the value of an FRP that represents some feature of the system. In terms of the framework in Figure 2.8, the Data FRP represents all of the relevant quantities we might measure from the random system, and an observation involves learning the value of a Feature FRP derived from that Data FRP.

In practice, we relate each observation to a Feature FRP of a special type, called an *event*.

Definition 29. An **event** is an FRP whose only possible values are 0 and 1.

When an event has the value 1, we say that the event *occurred*, and when it has the value 0, we say that the event did not occur.

An event gives a random Boolean value, an indicator whether some particular outcome has occurred. We generally interpret the values of an event as false (0) and true (1), and we can operate on those values as numbers whenever convenient. Recall that we make no distinction between scalars and 1-tuples.

We have seen many events in earlier examples. In the Monty Hall example of Section 1.4, two relevant events are that (i) you choose the prize door initially, and (ii) you do not choose the prize door initially. These are the FRPs `got_prize_door_initially(Game)` and `didnt_get_prize_door_initially(Game)` that we defined in `frplib`. In Roulette (Example 2.2), for every play, we get the event that that play wins when the wheel is spun. For instance, The FRPs `Red ~ (__ > 0)` and `Str16 ~ (__ > 0)` are the events that, respectively, a play on Red and a straight play on 16 win. In Example 4.11, we generated a random point in the circle, represented by the FRP `XY_m`. We can ask whether that random point falls within a

Event	Logical Operation
$1 - V$	Logical-not: occurs when V does <i>not</i> occur
$V + W = 1$	V and W are complementary events, $W = 1 - V$
VW	Logical-and: occurs when both V and W occur
$V + W - VW$	Logical-or: occurs when at least one of V or W occur
$V \leq W$	Implication: W occurs whenever V does
$\sum_{i=1}^n V_i$	Count of how many events V_i occur

TABLE 5.1. Event Logic. The correspondence between event arithmetic (using inline statistics) and logical operations on the conditions defining the events. Here, V , W , and $V_1, V_2, \dots, V_n$ are events.

distance 3 of the origin. The answer to this question is represented by the FRP $XY\_m \sim (\text{Norm} \leq 3)$, where Norm is the statistic that computes the distance of a value from the origin. In the Alice and Bob example from the previous section, we considered the event that the FRP A has second component 1, expressed as $A \sim (\text{Proj}[2] == 1)$.

These examples reflect a general truth: events arise by asking a binary question about the value of some FRP. Every event V has the form $V = \xi(D)$ for some FRP D and some *condition* ξ . The condition ξ asks a Boolean question about the value of D and V represents the answer to that question. And for any collection of events $V_1, V_2, \dots, V_n$, there is an FRP D and conditions $\xi_1, \xi_2, \dots, \xi_n$ such that $V_i = \xi_i(D)$ for $i \in [1..n]$. D here is often the Data FRP for a random system, but it need not be. Conversely, if X is an arbitrary FRP and ξ an arbitrary, compatible condition, then $V = \xi(X)$ is an event.

The two key points to remember are that (i) **events are FRPs**, and (ii) **events have possible values 0 and 1**, representing an answer to a yes-no question about some FRP. We will see in the next section that an observation of a random system means learning whether some particular event occurred.

THE LOGIC OF EVENTS. In Chapter 12 of Interlude F, we see how particular manipulations of indicator functions correspond to various logical operations. See Table 12.1. Because events are FRPs are transforms of other FRPs with conditions (indicator functions), the same relationships hold. These relationships are described in the following and summarized in Table 5.1.

For instance, if V is an event there is another *complementary event* that has the value false (0) when V has the value true (1) and the value true (1) when V has the

value false (0). We can define this as $\text{not}(V)$ with the statistic $\text{not}(v) = 1 - v$, but we usually write the statistic “inline”⁴ as $1 - V$, or as $!V$ when we want to emphasize its Boolean-ness. The complementary event occurs when V does not and does not occur when V does.

⁴See discussion of inlined statistics on page 131.

Similarly, if V and W are events, then the event VW , whose value is the *product* of the values of the two events, represents their *logical-and*. VW occurs when both V and W occur and does not occur when either V or W do not occur.

This case shows why inline statistics are so convenient because they give the idea without too much formality. We will frequently use inlining in this way in what follows, but it is worth stating the formal definition to clarify what the inlining with multiple FRPs means. The key fact is that if V and W are events, then there is⁵ a 2-dimensional FRP U such that $V = \text{proj}_1(U)$ and $W = \text{proj}_2(U)$. So:

⁵See Aside below for a demonstration of this fact.

- VW is the FRP defined by $\psi(U)$ where $\psi(v, w) = vw$.
- The event $V + W - VW$ is the *logical-or* of the two events, which occurs when at least one of V and W occur. It is defined formally as $\zeta(U)$ where $\zeta(v, w) = v + w - vw$. (The $-VW$ term here ensures that this only has values 0 and 1, even if both V and W occur.)
- V and W are complementary events if $V + W = 1$, meaning that $\varphi(U)$ is the constant FRP with value 1, where $\varphi(v, w) = v + w$.
- We say that $V \leq W$ if the value of V is guaranteed to be $\leq$ the value of W . This holds when $\delta(U)$ is true, where δ is the condition $\delta(v, w) = \{v \leq w\}$. $V \leq W$ means that the event V *implies* the event W . That is: if V occurs, then we know W occurs as well.

Finally, if X is an arbitrary FRP and $V_i = \xi_i(X)$ are events for conditions ξ_i for $i \in [1 \dots n]$, then the FRP

$$V_1 + V_2 + \dots + V_n \tag{5.1}$$

represents the *count of these events that occur*. Again, this inline transform is formally defined as $\bar{\psi}(X)$ where $\bar{\psi}(x) = \sum_{i=1}^n \xi_i(x)$.

Puzzle 69. If V and W are events, what does it mean if $V + W \leq 1$? If $V + W < 1$ strictly, what can you say about the event VW ? What does it mean if $V + W < 1$ strictly?

Aside. Earlier, I made the following claim: if V and W are events, then there is a 2-dimensional FRP U such that $V = \text{proj}_1(U)$ and $W = \text{proj}_2(U)$.

To see that this is true, notice that we can define $U = V \triangleright \text{const}_W$, where const_W is the conditional FRP that returns W for all inputs. We can write this join as $V :: W$.

If V and W are not independent, this is not a *clean join*, so we cannot use equation (4.17) to find $\text{kind}(U)$. To do that, we need more information about the relationship between V and W and construct U with a different join.

SPECIFYING EVENTS. As we have seen, events arise as the transform of an FRP by a condition, a Boolean statistic. By inlining the condition, we often specify events via **Boolean expressions** in terms of one or more FRPs. Recall that a Boolean expression is a mathematical statement in terms of one or more quantities that reduces to either true or false, including multiple statements combined with logical-and ($\wedge$) and logical-or ($\vee$).

A condition is just a special case of an *indicator* that accepts tuples of numbers as input and returns either the value 0 or 1.⁶ We thus often write a condition as a Boolean expression surrounded by **Iverson braces** (see page 614), with the condition returning 1 when the expression is true. For instance, the 1-dimensional condition $\xi(x) = \{3x^2 - 4 > 0\}$ is true (1) its argument x satisfies $x^2 > 4/3$. We can also write this anonymously⁷ as $\{3\blacksquare^2 - 4 > 0\}$, with the hole $\blacksquare$ being filled with the argument to the function. For any compatible FRP U , the transformed FRP $\xi(U)$ is an event, and we can write this in inline form by inlining the definition of the condition. Here, $\xi(U)$ can be written as

$$\{3U^2 - 4 > 0\}.$$

This is a shorthand that gets at the meaning of $\xi(U)$ without the need to define an explicit condition for each such event. If there is more than one FRP $X, Y, Z, \dots$ involved in the Boolean expression, then we are applying the statistic to a combined FRP $X :: Y :: Z :: \dots$ that (uncleanly) joins the constituent FRPs.

For example, in the Alice and Bob example, the observation that the second component of FRP A is 1 tells us that the event $\{\text{proj}_2(A) = 1\}$ occurred. In Roulette, the event that a red play wins is $\{\text{red}(R) > 0\}$, where **red** is the statistic **roulette.red**. In the Monty Hall example, the event that you picked the prize door initially is $\{M = Y\}$ where M and Y are the FRPs for Monty's and your choices.

Keep in mind that **events are FRPs**. They may or may not occur. And we can

⁶See page 94.

⁷See Chapter 11 of Interlude F for more on anonymous functions. This is just a way to define a function without giving a name; we show where to put the argument in the returned expression.

analyze events – e.g., compute their Kinds or expectations – just like we can for any other FRP.

Key Take Aways

An **event** is an FRP whose only values are 0 and 1. We use events to record whether particular outcomes occurred as the random system evolved, with 1 meaning yes and 0 no. An event whose value is 1 is said to have *occurred*.

Events arise by asking a yes-or-no question, so every event V has the form $V = \xi(D)$ for some FRP D and some condition ξ .

Events are random Boolean values, and so common transforms of events express logical operations (e.g., and, or, not, counting), as described in Table 5.1.

We frequently express events using inline statistics (cf. page 131) and Iverson braces (see page 614). An inline Boolean expression in terms of FRPs enclosed in braces describes the event that occurs when the expression is true, e.g., $\{X > 0\}$, $\{Y = 4 \wedge Z < -10\}$.

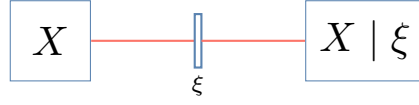
5.3 Observations

An **observation** of a random system occurs when we learn the value of some FRP derived from the system. We use observations to account for new information that we obtain as a random system evolves. The the constraint of an observation changes our knowledge and predictions about the system.

We can always express an observation in terms of *events*: the observation means that we learn whether a particular event occurs. Because each event is the transform of some FRP (e.g., the Data FRP for the system) by a condition, we can describe an observation with either the event whose outcome we learn or with the condition that defines that event.

Constraining with an observation is one of our Big 3+1 operations on both FRPs and Kinds. As in Section 1.5.2, we consider three interpretations of this operation.

Physical. The wiring diagram for constraining an FRP X for the observation that the event $\xi(X)$ occurs (where ξ is a condition) looks like



Here, we connect the All output port for X to the *constraint processor*



and its output to the input port of a new, fresh FRP $X \mid \xi$ that becomes the constrained FRP. The constraint processor can be decorated with a specification of the observation (e.g., a condition as above) to identify the constraint. If X 's value is consistent with the observation, the constraint processor passes its value through unchanged. Otherwise, the constraint processor uses its connection to X to generate a new value that is consistent with the observational constraint.⁸

⁸See “Under the Hood” below.

Algebraic. We use the *given operator* $\mid$, which is read as “given,” to constrain an FRP or Kind with an observation. On the left side of this operator, we put the FRP or a Kind to be constrained. On the right side of the operator, we give any specification of the observation, such as a condition or an event. When we write $X \mid \xi$ for an FRP X and a compatible condition ξ , it means that our observation is that the event $\xi(X)$ occurs, i.e., $\xi(X) = 1$. When we use an event on the right side of the operator, e.g., $X \mid \{X > 0\}$, we can as a convenient shorthand drop the braces around the Boolean expression if we like, e.g., $X \mid X > 0$.

As a general rule, the observation on the right of the given bar should be defined in terms of the FRP (or Kind) on the left. So if X is the FRP on the left, the observation should be determined by some condition ξ compatible with X . But in mathematical work, it is convenient to allow a shorthand that effectively loosens that rule syntactically. If we write something like

$$X \mid \{X = 2 \wedge Y < 4\},$$

where Y is not a transform of X , this is a shorthand for

$$\varphi(D \mid \{\varphi(D) = 2 \wedge \psi(D) < 4\}),$$

where D is an FRP such that $X = \varphi(D)$ and $Y = \psi(D)$. What D , φ , and ψ are in any particular case will typically be clear from context. Often, D will be the Data FRP for the system.

Definition 30. If X is an FRP and ζ is a compatible condition, then $X \mid \zeta$ (read “ X given ζ ”) is the FRP obtained from X by requiring the event $\zeta(X)$ to occur, i.e., that $\zeta(X)$ is true. $X \mid \zeta$ is equivalent to $X \mid \{\zeta(X) = 1\}$.

Think of $X \mid \zeta$ as a copy of X rewired to produce only values consistent with $\zeta(X)$ being true. The two FRPs are related and dependent, but in general they have different Kinds. If x is the value of X and $\zeta(x)$ is true, then $X \mid \zeta$ also has value x . If $\zeta(x)$ is not true, then $X \mid \zeta$ generates a new value consistent with the observation. The only case where this rewiring is not possible is when ζ is false for *every possible* value of X . Then, $X \mid \zeta$ equals the **empty** FRP, representing a logical contradiction.

Under the Hood. Let X be an FRP and ζ be a compatible condition. Assume that ζ is true for some possible value of X . To see the relationship between the values of X and $X \mid \zeta$, we give a procedure that generates a value equivalent to the latter. (This also gives a sense of how the constraint processor works in the wiring diagram.)

Let $X_1, X_2, X_3, \dots$ be clones of X . Assume that X has value x and X_i has value x_i for $i \in [1..)$. Initialize i to 0 and execute the following steps in order until a value is returned:

1. If $\zeta(x)$ is true, return x .
2. Increment i : $i \leftarrow i + 1$.
3. If $\zeta(x_i)$ is true, return x_i .
4. Go to step 2 and continue.

If k is a Kind and ξ is a compatible condition, we write $k \mid \xi$ for the Kind k constrained with the observation ξ . This operation updates the knowledge embodied by the Kind to account for the observation. We again require only that ξ be true for at least one possible value of k , otherwise the constraint makes no sense. The updated Kind is easy to compute: we convert k to canonical form and drop all branches/paths whose leaf nodes are inconsistent with the observation.

Definition 31. Kinds Constrained with an Observation.

If k is a Kind and ζ is a compatible condition that is true for some possible value of k , we form the Kind constrained by an observation $k \mid \zeta$ by

- (i) converting k to canonical form $\tilde{k}$,
- (i) erasing the branches of $\tilde{k}$ whose values v satisfy $\zeta(v) = 0$, and

(i) if desired, renormalizing the weights on the remaining branches to sum to 1. Step (iii), converting the constrained Kind to canonical form, is frequently useful but not required.

For the condition $\top$ that is always true, $k \mid \top = k$. For the condition $\perp$ that is always false, $k \mid \perp = \langle \rangle$, the empty Kind, indicating a logical contradiction.

Constraining FRPs and Kinds with a condition are compatible operations. In particular, if X is an FRP and ζ is a compatible condition, then

$$\text{kind}(X \mid \zeta) = \text{kind}(X) \mid \zeta. \quad (5.2)$$

The Kind of the constrained FRP is the constrained Kind.

In `frplib`, we use the `|` operator as “given,” and we always use a condition (rather than an event) on the right hand side. Because in Python `|` has low operator precedence, any condition on the right side based on an expression should be wrapped in parentheses. So, for instance, we would write `X | (Proj[2] - Proj[2] > 4)` with the expression in parentheses, but it is fine to write `k | And(Proj[1] > 2, Sum == 10)` because the `And(...)` is a single term.

Procedural. FRP and Kind procedures enforce observational constraints with *given* statement in the procedure, which takes a Boolean value.⁹ If the given value is false, the procedure rejects the current case and starts over with another value. If the given value is true, the procedure continues.

⁹In the extended sense: true, false, $\langle 1 \rangle$, or $\langle 0 \rangle$

FRP Y :	Kind ℓ :
$x \leftarrow X$	$x \leftarrow k$
given $\xi(x)$	given $\xi(x)$
return x	return x

This defines $\text{FRP } Y = X \mid \xi$ and $\text{Kind } \ell = k \mid \xi$. The FRP procedure extracts X ’s value and checks the condition. If true, it returns that value. If not, it starts again with a clone of X and continues until it returns a value. A Kind procedure operates in parallel over all possible values of k ; any values for which the condition is false are excluded by the **given** line. The result is automatically normalized using only the remaining canonical weights of k . In `frplib`, observational constraint procedures have the same form, but we use `yield given(b)` instead of `given b`. The analogous procedures in `frplib` are:

```

@frp                                @kind
def Y():                            def ell():
    x = yield X                     x = yield k
    yield given(xi(x))              yield given(xi(x))
    return x                        return x

```

where Y is $X \mid xi$ and ell is $k \mid xi$.

Constraining with observations allows us to use actual or *hypothetical* observations. We can use this operation to investigate how our knowledge and predictions will be updated for various possible events. This is a powerful tool, as we will see in upcoming examples.

Example 5.1. An FRP representing three *independent* flips of a balanced coin, with 0 for tails and 1 for heads, has Kind $k = k_{\text{flip}} \star k_{\text{flip}} \star k_{\text{flip}}$, where k_{flip} is the Kind of a single flip. We will compute the Kind $k \mid \zeta$ for a variety of conditions, in `frplib`.

Puzzle 70. Define conditions that test the following assertions:

1. The first flip is a heads.
2. There is exactly one heads among the three flips.
3. There is at least one tails among the three flips.
4. The first and second flips have the same result.

As an example, the condition that tests if at most one of the first two flips is a heads can be defined as a named condition ψ by $\psi(v) = \{\text{proj}_1(v) + \text{proj}_2(v) \leq 1\}$ or as an anonymous condition by $\{\text{proj}_1(\blacksquare) + \text{proj}_2(\blacksquare) \leq 1\}$.

We start by defining the Kinds

```

pgd> flip = uniform(0, 1)
pgd> three_flips = flip ** 3
pgd> three_flips
,---- 1/8 ---- <0, 0, 0>
|---- 1/8 ---- <0, 0, 1>
|---- 1/8 ---- <0, 1, 0>
|---- 1/8 ---- <0, 1, 1>
<> -|
|---- 1/8 ---- <1, 0, 0>
|---- 1/8 ---- <1, 0, 1>

```

```
|---- 1/8 ---- <1, 1, 0>
`---- 1/8 ---- <1, 1, 1>
```

Next, we define several conditions to study. We show equivalent definitions of these conditions as named functions and as anonymous functions, where $\blacksquare$ is a hole to be filled by the single argument.

When writing a hole by hand, use any consistent mark, like dash (–) or underscore (_).

Named Condition	Anonymous Condition
$\zeta(v) = \{\text{proj}_1(v) = 1\}$	$\{\text{proj}_1(\blacksquare) = 1\}$
$\xi(v) = \{\text{Sum}(v) = 1\}$	$\{\text{Sum}(\blacksquare) = 1\}$
$\varphi(v) = \{\text{Min}(v) = 0\}$	$\{\text{Min}(\blacksquare) = 0\}$
$\gamma(v) = \{\text{proj}_1(v) = \text{proj}_2(v)\}$	$\{\text{proj}_1(\blacksquare) = \text{proj}_2(\blacksquare)\}$

These conditions correspond exactly to those listed in Puzzle 70.

Using an anonymous condition, we can write $k \mid \zeta$ as $k \mid \{\text{proj}_1(\blacksquare) = 1\}$, for instance. Notice the similarity between the anonymous functions and the form of statistics/conditions in `frplib`, e.g., ζ written mathematically as $\{\text{proj}_1(\blacksquare) = 1\}$ is written `Proj[1] == 1` in `frplib`.

```
pgd> three_flips | (Proj[1] == 1)
,---- 1/4 ---- <1, 0, 0>
|---- 1/4 ---- <1, 0, 1>
<> -|
|---- 1/4 ---- <1, 1, 0>
`---- 1/4 ---- <1, 1, 1>
```

In this case, k is in canonical form, so to find $k \mid \zeta$, we eliminate all the branches in k with values v for which $v_1 \neq 1$. This leaves us the bottom subtree with the four branches whose first component is 1, all with weight $1/8$. Renormalizing the weights to sum to 1 gives the Kind shown, which is in canonical form.

This Kind capture our knowledge about the three flips having seen a heads on the first flip. All of its possible values satisfy the condition, and the relative weights of the remaining values are the *same* as the relative weights of these values in the original Kind. We thus see that all four possibilities for the remaining flips are equally weighted.

Notice that `(three_flips | (Proj[1] == 1)) ^ Proj[2,3]` is equal to `flip * flip`. (Try it!) Observing the first component of an independent join does not change our knowledge of the other components!

Similarly, we can see how our knowledge is updated if we know that there is only one heads among the three flips.

```
pgd> three_flips | (Sum == 1)
      ,---- 1/3 ---- <0, 0, 1>
<> -+---- 1/3 ---- <0, 1, 0>
      `---- 1/3 ---- <1, 0, 0>
```

In this case $k \mid \xi$, we again start with the Kind k and eliminate all the branches with other than one heads in three flips. The three remaining branches each have one head, with one branch per flip in which that head occurs. All values have equal weight because the relative weights do not change from the original Kind, where they were equally weighted.

Notice that

```
pgd> Kind.equal(kind(frp(K) | (Sum == 1)), K | (Sum == 1))
True
```

confirming equation (5.2).

We can do the same for the other two conditions. Whether these conditions represent actual or hypothetical observations does not matter. We can use this to investigate how our knowledge and predictions change under various observations.

```
pgd> three_flips | (Min == 0)
      ,---- 1/7 ---- <0, 0, 0>
      |---- 1/7 ---- <0, 0, 1>
      |---- 1/7 ---- <0, 1, 0>
<> -+---- 1/7 ---- <0, 1, 1>
      |---- 1/7 ---- <1, 0, 0>
      |---- 1/7 ---- <1, 0, 1>
      `---- 1/7 ---- <1, 1, 0>
```

Here, the condition only eliminates the branch $\langle 1, 1, 1 \rangle$ with no tails, leaving seven equally weighted branches.

```
pgd> three_flips | (Proj[1] == Proj[2])
```



```

,---- 1/4 ---- <0, 0, 0>
|---- 1/4 ---- <0, 0, 1>
<> -|
|---- 1/4 ---- <1, 1, 0>
`---- 1/4 ---- <1, 1, 1>

```

And here, we eliminate the four branches in K where the first two flips disagree, leaving four equally weighted branches remaining.

Puzzle 71. What do you expect to see when you enter

```
three_flips | (__ == (1, 0, 1))
```

in `frplib`? Explain why this makes sense.

In practice, we will often want to specify more general observations of the form $\text{kind}(X \mid V)$, where V is an event that may not be a direct transformation of X . Because the Kind given the constraint depends jointly on both X and V , to compute $\text{kind}(X \mid V)$, we need to start with an FRP that determines¹⁰ the values of both X and V . Specifically, we require that there is an FRP D such that $X = \psi_1(D)$ and $V = \psi_2(D)$ for some statistics ψ_1, ψ_2 . Then

$$\text{kind}(X \mid V) = \psi_1(\text{kind}(D) \mid \psi_2). \quad (5.3)$$

That is, we constrain $\text{kind}(D)$ with the observation ψ_2 , i.e., that the event $V = \psi_2(D)$ occurs, and then transform that Kind by ψ_1 to extract $X = \psi_1(D)$, yielding the Kind of X given the observation.

Let's see some examples in detail to make this concrete.

Example 5.2 What's in the Box?

We have three boxes, labeled 1, 2, and 3, each of which contains a number of colored balls. Box 1 contains 1 blue ball and 3 red balls; Box 2 contains 2 blue balls and 4 red balls; Box 3 contains 3 blue balls and 5 red balls.

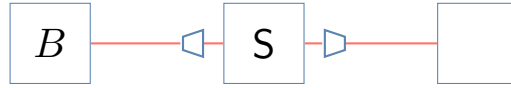
I randomly choose a box and then randomly select a ball from that box, without showing you which box I chose. I then show you the color of the selected ball. Assume that all boxes are equally likely to be chosen and that all balls within the chosen box are equally likely to be selected. What have you learned

¹⁰This is the role of the Data FRP in Figure 2.8, although we may use an FRP derived from that as well.

about the chosen box from the color of the selected ball?

We start by building the system and then will use the form of that system to analyze it.

Let B be the FRP representing the chosen box, and let S be the conditional FRP that represents the selected ball given the chosen box. We will encode B to have values 1, 2, and 3, the box labels. For S , we will arbitrarily assign values 16 for a blue ball and 32 for a red ball. The outcome of the system is represented by the FRP $B \triangleright S$.



This is the Data FRP for this system from which we will extract all features of interest.

Two such features are the color of the selected ball, an FRP $C = \text{proj}_2(B \triangleright S)$ and the selected box $B = \text{proj}_1(B \triangleright S)$. The motivating question in this example is to find how our knowledge of B updated by an observation of C 's value.

Seeing a blue ball is the observation that the event $\{C = 16\}$ has occurred. Seeing a red ball is the observation that the event $\{C = 32\}$ has occurred. (Recall that an event is an FRP with values 0 or 1, so $\{C = 16\}$ occurring means that the FRP $\{C = 16\}$ has the value 1.) Our analysis will consider how each of these observations changes our knowledge of B .

So, we want to find the Kinds of B given both observations of ball color:

$$\begin{aligned} \text{kind}(B \mid \{C = 16\}) &= \text{proj}_1(\text{kind}(B \triangleright S \mid \{\text{proj}_2(x) = 16\})) \\ \text{kind}(B \mid \{C = 32\}) &= \text{proj}_1(\text{kind}(B \triangleright S \mid \{\text{proj}_2(x) = 32\})). \end{aligned}$$

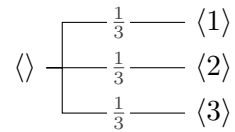
where we put the shorthand on the left and the formal definition on the right.

Because $B \triangleright S$ is a clean join, we can use the identity in equation (4.17) to find

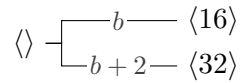
$$\text{kind}(B \triangleright S) = \text{kind}(B) \triangleright \text{kind}(S). \quad (5.4)$$

Here, $s = \text{kind}(S)$ is the conditional Kind of the selected color given the chosen box.

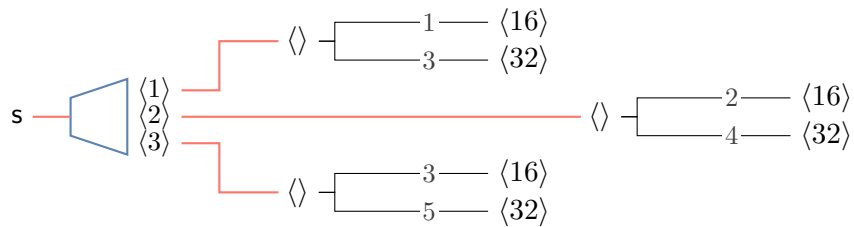
By our assumptions, B has Kind



And within box b there are b blue balls and $b + 2$ red balls, so the Kind of the selected color in box b is



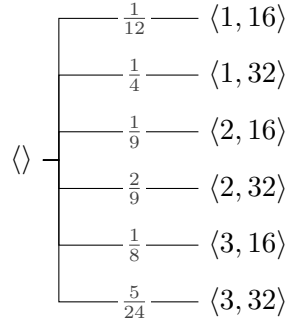
Hence, s is



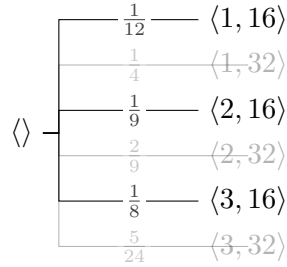
We can find our target Kinds in `frplib`:

```
pgd> b = uniform(1, 2, 3)
pgd> s = conditional_kind({
...>   1: weighted_as(16, 32, weights=[1, 3]),
...>   2: weighted_as(16, 32, weights=[2, 4]),
...>   3: weighted_as(16, 32, weights=[3, 5]),
...> })
pgd> B = frp(b)
pgd> S = conditional_frp(s)
pgd> BC = B >> S
pgd> kind(BC) # equals b >> s
```

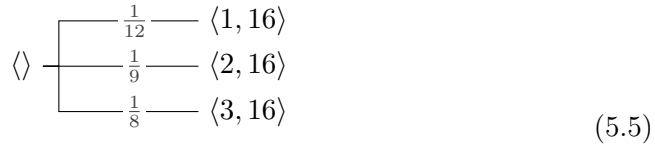
which yields



The FRP BC ($B \triangleright S$) and its Kind describe the values of B and C *jointly*, so we can determine which possibilities are or are not consistent with the observation. We constrain with the observation $\{C = 16\}$ by eliminating the branches whose second component is not 16.



to produce



In `frplib`, we get this by

```
pgd> kind(BC) | (Proj[2] == 16)
```

where the observation is specified by applying the condition on the right of the “given” bar to the values of the Kind or FRP to the left of the bar.

Notice that (i) constraining with an observation has not changed the weights of the remaining branches, and (ii) all the branches have values where $C = 16$ because of the constraint. We are interested in the Kind of B , so we transform the Kind (5.5) by projecting onto the first component

$$\langle \rangle \begin{cases} \frac{1}{12} & \text{---} \langle 1 \rangle \\ \frac{1}{9} & \text{---} \langle 2 \rangle \\ \frac{1}{8} & \text{---} \langle 3 \rangle \end{cases}$$

Also, it is optional but helpful to follow our standard practice and convert this Kind to canonical form, which yields $\text{kind}(B) \mid C = 16$:

$$\langle \rangle \begin{cases} \frac{6}{23} & \text{---} \langle 1 \rangle \\ \frac{8}{23} & \text{---} \langle 2 \rangle \\ \frac{9}{23} & \text{---} \langle 3 \rangle \end{cases}$$

By default, `frplib` will convert to canonical form (see functions `unnormalized` and `raw` for alternatives.) We get this Kind in `frplib` with any of

```
pgd> (kind(BC) | (Proj[2] == 16))[1]
pgd> Proj[1](kind(BC) | (Proj[2] == 16))
pgd> (kind(BC) | (Proj[2] == 16)) ^ Proj[1]
pgd> Proj[1] @ kind(BC) | (Proj[2] == 16)
```

Pay attention to the parentheses here. The form of these and the new `@` operator will be explained below.

In `frplib`, we only use the basic form $K \mid \zeta$ and $X \mid \zeta$ for constraining with an observation. So, if we want to compute a more general form, such as $\text{kind}(B \mid \{C = 16\})$ in the previous example, we need to do the translation of equation (5.3) manually. The first three lines above

```
(kind(BC) | (Proj[2] == 16))[1]
Proj[1](kind(BC) | (Proj[2] == 16))
(kind(BC) | (Proj[2] == 16)) ^ Proj[1]
```

are variations on that identity, where we compute the joint Kind with the observation and then transform it with `proj1` to focus on B. The fourth line provides a shortcut

```
Proj[1] @ kind(BC) | (Proj[2] == 16)
```

using the `@` operator. If `stat` is a statistic and `U` is a Kind, then `stat @ U` is the same as `stat(U)` or `U ^ stat`, with two exceptions: (i) `@` has higher precedence than `^` or `|`, and (ii) the result “remembers” that it came from `U` when it comes to applying the condition after the given bar. So:

```
pgd> kind_B = Proj[1] @ kind(BC)
pgd> kind_B | (Proj[2] == 16)
```

produces $\text{kind}(B) \mid C = 16$, where `kind_B` looks like the $\text{kind}(B)$ but with some extra information behind the scenes. If we try instead to do just $\text{kind}(\text{BC} \sim \text{Proj}[1]) \mid (\text{Proj}[2] == 16)$, we will get an error because $\text{kind}(\text{BC} \sim \text{Proj}[1])$ does not have that context.

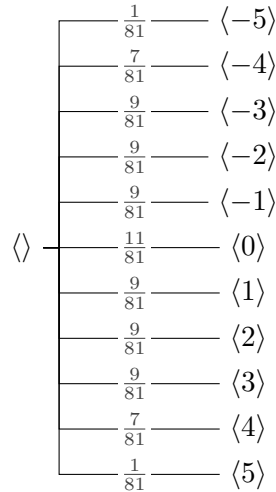
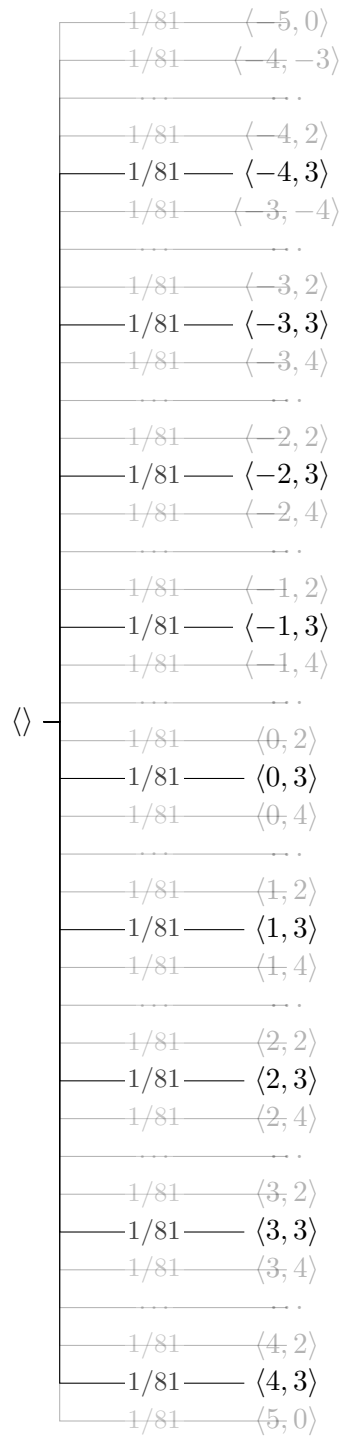


FIGURE 5.5. The Kind of $X = \text{proj}_1(P)$ and of $Y = \text{proj}_2(P)$ in Example 5.3.

Example 5.3 Points in a Circle, With Constraints In Example 4.11, we built a two-dimensional FRP, call it P here, that represents a random point with integer coordinates within a circle of radius 5, where all valid points have equal weight. Express P in terms of its scalar component FRPs, $P = \langle X, Y \rangle$, where X and Y represent the x - and y -coordinates of the random point. P is the Data FRP for this system and X and Y are features of interest.

The Kind of X embodies all our predictions about the x -coordinate alone, and the Kind of Y embodies all our predictions about the y -coordinate alone. Because the points in the circle are symmetric under a 90-degree rotation, both of these have *the same kind* (i.e., $\text{kind}(\text{proj}_1(P)) = \text{kind}(\text{proj}_2(P))$), shown in Figure 5.5. Without any information about the other coordinate, any point in the circle is possible, and the weight on each value of the coordinate is proportional to the number of points with that coordinate (e.g., there are 11 points with x -coordinate 0, where y ranges from -5 to 5).

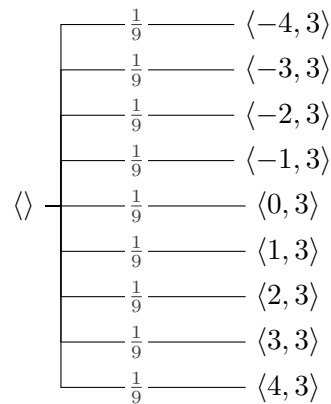
With observed information about one coordinate, however, our predictions

FIGURE 5.6. The Kind of $P \mid Y = 3$ showing the eliminated branches, in Example 5.3.

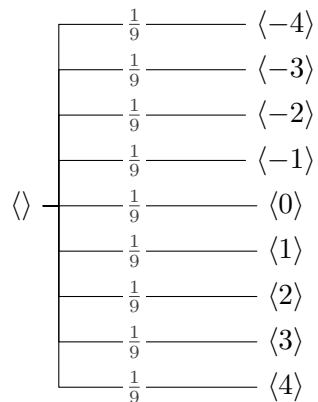
about the other coordinate *change*. Suppose, for example, we observe that $Y = 3$. Remember $\{Y = 3\}$ is an *event* – an FRP that has values 0 and 1. To say that we observe that $Y = 3$ means that we can take it has a fact that the event $\{Y = 3\}$ has occurred. So the event FRP has a value 1 and the FRP Y has value 3.

If we know that $Y = 3$, then P must be a point of the form $\langle x, 3 \rangle$ with x an integer satisfying $x^2 \leq 25 - 9 = 16$, i.e., $x \in [-4..4]$. So, if we eliminate all the branches that are inconsistent with this observation, we get the updated Kind for $P \mid Y = 3$ in Figure 5.6 where all the inconsistent branches have been crossed out. Where before we had equal weight on each of 81 points, we now have 9 remaining branches all with the same weight. An important point to remember: when we eliminmate branches inconsistent with the observation, **the relative sizes of the weights on the remaining branches do not change**.

When we renormalize to canonical form, the Kind of $P \mid Y = 3$ becomes



And we get the Kind of $X \mid Y = 3$ by transforming this with the statistic proj_1 , yielding the Kind



Compare this to the Kind of X without any information about Y , shown above. Once we know the value of $Y = y$, X must be one of the values in the circle along the horizontal line of height y , and all of those values are equally weighted because they were equally weighted originally.

In `frplib`, using `circle_points` from the earlier example, try

```
pgd> P = circle_points()    # The Data FRP
pgd> kind(P)
pgd> kind(P) | (Proj[2] == 3)
pgd> Proj[1] @ kind(P) | (Proj[2] == 3)
```

The output is omitted but should match the displays above. Try it!

Example 5.4 Jockeying for a Win

The *Uncertain Stakes* is one of the most exclusive horse races on the world circuit but little noted by the general public. This year's race has eight elite contenders, numbered as follows

1. *Aldous Aboard*
2. *Eggs Billingsley*
3. *Cinlar's Challenge*
4. *Feller Beast*
5. *Kissing Kolmogorov*
6. *Levy Leaving*
7. *Markov Mania*
8. *Pitman's Pride*

These horses vary strongly in their response to the track conditions. Some like

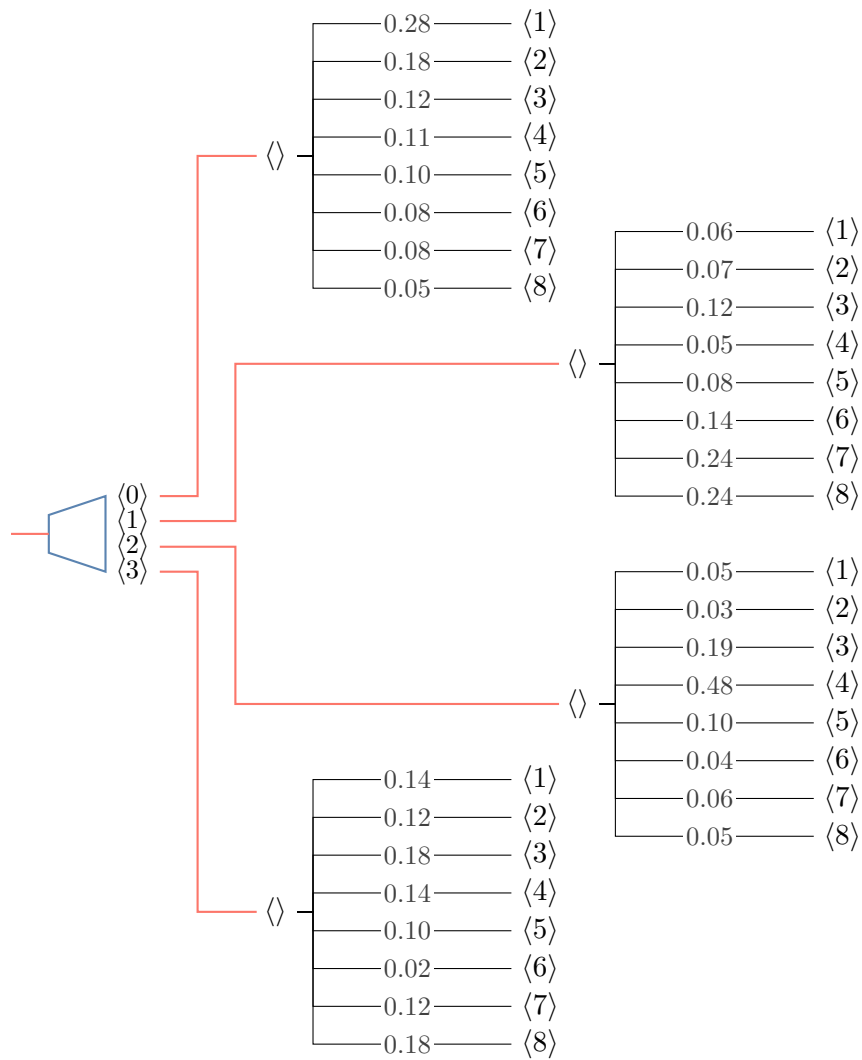


FIGURE 5.7. The conditional Kind of the winning horse given track condition, $\text{kind}(W)$, in Example 5.4. The inputs represent a fast (0), muddy (1), sloppy (2), and slow (3) track. The value of the Kinds is the winning horse's number.

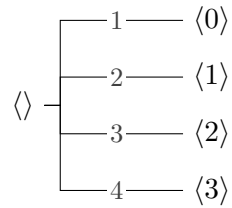
the track muddy; some like it dry; some do better in driving rain. There are many questions we might ask, but consider three. *Who will win? If just before the race, we observe that the track is muddy, who will win? If we observe that Cinlar's Challenge wins (but do not observe the track conditions), what can we say about the track conditions?*

Let T be the FRP representing the track conditions and W be the conditional FRP of the winning horse given the track conditions. T 's values 0, 1, 2, 3 represent the conditions fast (dry, even resilient surface), muddy (wet without standing water), sloppy (saturated with water, with standing water), and slow (wet on both surface and base layers). For each track condition, W has values 1–8 corresponding to the number of the winning horse.

The Data FRP that describes this system is the join $R = T \triangleright W$. This is a two-dimensional FRP that represents both the track conditions and the winning horse jointly. The key feature FRPs of interest represent the winning horse and the track conditions: $H = \text{proj}_2(R)$ and $T = \text{proj}_1(R)$.

We are interested in (i) predicting who will win this year's *Uncertain Stakes*, both with and without information about the track conditions, and (ii) *inferring* the track conditions *given* that *Cinlar's Challenge* is observed to win the race. In the first case, we want to find $\text{kind}(H)$ to make predictions about who will win without information about the conditions and $\text{kind}(H) \mid T = c$ to make predictions about who will win when we know the track has condition c . In the second case, we use the observation $\{H = 3\}$ to find $\text{kind}(T \mid \{H = 3\})$. In both cases, the observations can come from real partial information that we observe or from *hypothetical* partial information that we are interested in considering. In the latter, our questions sound like “What would we predict about which horse wins if we knew the track had condition c ?” or “What would we infer about the track condition if we knew that *Cinlar's Challenge* wins?”

Assume that based on the horses' histories, we can say that $\text{kind}(W)$ is the conditional Kind shown in Figure 5.7. We can see, for instance, that fast conditions favor *Aldous Aboard* and *Eggs Billingsley*, sloppy condition are dominated by *Feller Beast*, and slow conditions disadvantage *Levy Leaving*. Assume also that based on weather forecasts and the previous days' conditions, we take the Kind of T to be



Because R is a clean join, we can use the identity in equation (4.17) to find

$$\text{kind}(R) = \text{kind}(T) \triangleright \text{kind}(W). \quad (5.6)$$

Notice how our *analysis* of the system parallels our *description* of the system.

You can load this example in `frplib` by entering

```
pgd> from frplib.examples.horse_race import T, W
```

Then do

```
pgd> R = T >> W
pgd> H = R[2]      # Shorthand for R ^ Proj[2]
```

Look at the Kinds of these FRPs. Indeed, we can immediately address the first question with

```
pgd> kind(H)
,---- 0.10200 ----- 1
|---- 0.080000 ---- 2
|---- 0.16600 ----- 3
|---- 0.25500 ----- 4
<> -|
|---- 0.096000 ---- 5
|---- 0.058000 ---- 6
|---- 0.11600 ----- 7
`---- 0.12700 ----- 8
```

which gives the chance of winning for each of the horses, absent any other information. Make sure you understand where this comes from. A good place to start would be to look at `unfolded(kind(R))` and apply our procedure for computing a join of Kinds to compute `kind(R)`. Then apply `Proj[2]`.

Our predictions change with additional information, allowing us to answer the second question. For instance, if we observe that the track is muddy, the Kind of the winning horse is instead

```
pgd> Proj[2] @ kind(R) | (Proj[1] == 1)
,---- 0.060000 ---- 1
|---- 0.070000 ---- 2
|---- 0.120000 ----- 3
|---- 0.050000 ---- 4
<> -|
|---- 0.080000 ---- 5
|---- 0.140000 ----- 6
|---- 0.240000 ----- 7
`---- 0.240000 ----- 8
```

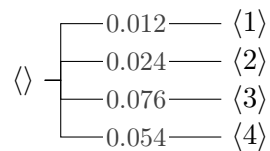
This is $\text{kind}(H) \mid T = 1$ and can also be obtained from $\text{kind}(W)$.

For the third question, we constrain with the observation that *Cinlar's Challenge* wins, meaning that the event $\{H = 3\}$ occurs. We want the Kind of T given that constraint: $\text{kind}(T) \mid H = 3$. We find this in three steps:

1. Start with the (canonical) Kind of the Data FRP R that represents *both* the track condition and the winning horse.
2. Eliminate all the branches in the Kind of R that are inconsistent with the given event, i.e., for which $H \neq 3$. All other branches remain *as is*.
3. Apply a projection statistic to this Kind that extracts the value of T , proj_1 in this case, and optionally, convert the resulting Kind to canonical form if desired.

Let's see these steps in action:

1. We start the Kind of R , which has T and H as components. If we know the value of R we can find the value of T and the value of H and can therefore determine whether $H = 3$.
2. Eliminating branches with $H \neq 3$ from the Kind of R looks like the Kind shown in Figure 5.8, with the eliminated branches grayed out.
3. Applying proj_1 to the Kind in step 2 yields



and following our standard practice, we convert this to canonical form

$$\langle \rangle \begin{cases} \text{---} 0.07229 \text{---} \langle 1 \rangle \\ \text{---} 0.14458 \text{---} \langle 2 \rangle \\ \text{---} 0.45783 \text{---} \langle 3 \rangle \\ \text{---} 0.32530 \text{---} \langle 4 \rangle \end{cases}$$

where the weights are respectively $6/83, 12/83, 38/83, 27/83$.

In `frplib`, the three steps are direct:

```
pgd> kind(R)
pgd> kind(R) | (Proj[2] == 3)
pgd> Proj[1] @ kind(R) | (Proj[2] == 3)
```

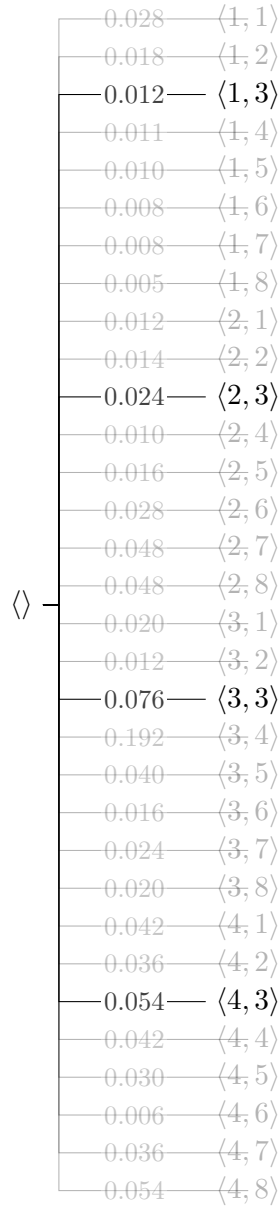
and the output should agree with the above displays.

Key Takeaway about Constraining with Observations.

To account in our analysis for observed information about the value of an FRP – actual or hypothetical – we eliminate from consideration any possible values that are inconsistent with the observation. The relative weights on the remaining branches *do not change*.

Keep in mind that conditions can be combined using logical operators: logical-and (operator $\wedge$), logical-or (operator $\vee$), and occasionally logical “not” (operator $!$). These operators tend to be convenient when writing complicated conditions, but it’s fine to use words (*and*, *or*, *not*) instead in your work. So, $\{\text{proj}_2(A) = 1 \wedge \text{proj}_1(A) = 1\}$ is the event that *both* of A ’s first two components are one, $\{\text{proj}_2(A) = 1 \vee \text{proj}_1(A) = 1\}$ is the event that *at least one* of A ’s first two components are one. Note that for multiple dimensional FRPs like A , we can use subscripts to denote components, so $A_1 = \text{proj}_1(A)$ and $A_2 = \text{proj}_2(A)$. In `frplib`, things are a little more awkward because of restrictions imposed by Python on repurposing the logical operators `and` and `or`. Instead, we use the combinators `And`, `Or`, and `Not` for logical *and*, *or*, and *not* of conditions. The first two of these take any number of conditions and can be nested to produce arbitrary Boolean expressions. For example,

```
A | And(Proj[2] == 1, Proj[1] == 1)
A | Or(Proj[2] == 1, Proj[1] == 1)
A | Not( And(Proj[2] == 1, Proj[1] == 1) )
```

FIGURE 5.8. The Kind of R showing branches eliminated by the observation that $H = 3$.

and so forth. And in `frplib`, we can use bracketed indexing on an FRP or Kind as a short hand for projection, so `A[2]` is the same as `Proj[2](A)` or `A ~ Proj[2]`.

OBSERVATIONS IN TERMS OF KERNELS. The *kernel* of a Kind k is the associated function¹¹ that maps a value v to the (canonical) weight on that value in the Kind tree k , or 0 if that value is not on any branch. See equation (1.1) expresses the kernel for k in terms of indicators; if k has possible values $v_1, v_2, \dots, v_s$ with respective weights $w_1, w_2, \dots, w_s$ its kernel is

¹¹See equation (1.1).

$$k(v) = \sum_{i=1}^s w_i \{v_i = v\}. \quad (5.7)$$

We use the same symbol for the Kind's name and its kernel for convenience.

The operation of constraining with observations excludes all branches from the (canonical) Kind that are inconsistent with the observation. Let ζ be a condition compatible with k . Excluding all the inconsistent branches corresponds to keeping only those values for which ζ is true. This is the function

$$\langle v \rangle \mapsto \sum_{i=1}^s w_i \{v_i = v\} \zeta(v). \quad (5.8)$$

This is like the kernel for $k \mid \zeta$ except that the weights are not normalized to sum to 1, which we require of kernels. But this equation gets at the key feature of the operation – erasing inconsistent branches. To make it a kernel, we just need renormalize:

$$(k \mid \zeta)(v) = \frac{\sum_{i=1}^s w_i \{v_i = v\} \zeta(v)}{\sum_{i=1}^s w_i \zeta(v_i)}. \quad (5.9)$$

The denominator is the sum of all weights of k for values that make the condition true. Notice how this is exactly what we would get if we formed the Kind tree $k \mid \zeta$ and then its kernel. Written in terms of kernels, this becomes equation (5.10) below. We can write this equivalently in terms of FRPs, recalling our notation kind_Z for the kernel of $\text{kind}(Z)$ for any FRP Z . This yields equation (5.11).

The Kernel of a Kind Constrained with an Observation

If k is a Kind and ζ is a compatible condition, the kernel of the constrained

Kind $k \mid \zeta$ is

$$(k \mid \zeta)(v) = \frac{k(v) \zeta(v)}{\sum_x k(x) \zeta(x)}. \quad (5.10)$$

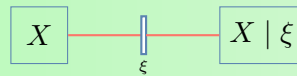
We can express this in terms of FRPs as well. If X is an FRP and ζ is a compatible condition that is true for some possible value of X :

$$k_{X \mid \zeta}(v) = \frac{k_X(v) \zeta(v)}{\sum_x k_X(x) \zeta(x)}. \quad (5.11)$$

Key Take Aways

An **observation** occurs when we measure the value of some FRP as a random system evolves. We express observations in terms of an *event*: we observe whether some event occurs. Thus each observation is associated with a condition that is compatible with the Data FRP for the system.

The operation of constraining an FRP with an observation has a wiring diagram



that connects the FRP through a *constraint processor* tailored to the observation.

The operation is written algebraically using the “given operator” $\mid$ that takes an FRP or Kind on the left and a specification of the observation on the right, either an event or a condition. For instance, if an observation has condition ζ , we write $X \mid \zeta$ or $X \mid \{\zeta(X) = 1\}$ for an FRP and $k \mid \zeta$ for a Kind. This operation can also be expressed in procedures using the **given** operator to enforce the constraint.

Constraining a Kind with an observation *eliminates from consideration any possible values that are inconsistent with the observation*. The **relative weights on the remaining branches do not change**.

We do this in two steps, with a third optional:

1. Convert the Kind to canonical form,
2. Eliminate any branches whose values are inconsistent with the observation, and
3. optionally, convert the resulting Kind to canonical form.

In terms of Kernels, constraining a Kind k with an observation ζ yields a Kind with Kernel

$$(k \mid \zeta)(v) = \frac{k(v) \zeta(v)}{\sum_x k(x) \zeta(x)}. \quad (5.12)$$

In terms of a compatible FRP X , this translates to

$$k_{X \mid \zeta}(v) = \frac{k_X(v) \zeta(v)}{\sum_x k_X(x) \zeta(x)}. \quad (5.13)$$

5.4 Bayes's Rule

The previous examples illustrated a common situation: for a random system that evolves in multiple stages, we observe the output of the later stages and use that observation to infer the output of the earlier stages. In Example 5.2, the early stage is selecting a box and the later stage is choosing a ball from the selected box. We observe the color of the selected ball and learn more about the chosen box it came from. In Example 5.4, the early stage output is the track conditions and the later stage output is the winner of the race. We observe the winning horse and from that learn more about the track conditions that held during the race. Computing the Kind of the early-stage output given the later-stage output is the purpose of **Bayes's Rule**.

The general ingredients for modeling this situation are:

1. A is an FRP of dimension n that represents the output of the early stages of the random system. We also define $a = \text{kind}(A)$.
2. S is a conditional FRP of type $n \rightarrow n + p$ that describes the transition from the early stage to the later stage of the system. Given the output of the early stage, this returns an FRP representing the later-stage output. We also define $s = \text{kind}(S)$ to be its conditional Kind.
3. The full system is described by the join FRP $F = A \triangleright S$, which combines both the early and later stage outputs. Its Kind is $f = a \triangleright s$.¹²
4. The statistics $\text{early} = \text{proj}_{..n}$ and $\text{later} = \text{proj}_{(n+1)..}$ extract the early and later stage outputs from the full system.¹³
5. The FRP $B = \text{later}(A \triangleright S)$ represents the later-stage output of the system. It has dimension p and Kind $b = \text{later}(a \triangleright s)$.

¹²Notice again the parallel between the structure of the system (the FRP) and our analysis of the system (the Kind).

¹³Recall that $\text{proj}_{i..j}$, $\text{proj}_{..j}$, and $\text{proj}_{i..}$ are respectively equivalent to $\text{proj}_{i,i+1,\dots,j}$, $\text{proj}_{1,2,\dots,j}$, and $\text{proj}_{i,i+1,\dots}$.

6. The observation of the later-stage output is specified by either the event $\{B = v\}$ or, equivalently, the condition $\{\text{later}(\blacksquare) = v\}$.

Throughout this discussion, we will consistently use letters u, u', u'' to denote potential early-stage outputs, letters v, v', v'' to denote potential later-stage outputs, and letters x, x' to denote outputs of the full system.

The Kind a represents our initial knowledge about the system's early stage, and the observation of the later-stage output v will constrain and *update* our knowledge to $\text{kind}(A \mid \{B = v\})$. This is what we want to find.

Following equation (5.3), we find this in three steps:

1. Find the Kind for the full system using the join,

$$f = a \triangleright s = \text{kind}(A \triangleright S).$$

2. Constrain this Kind with the observation to get

$$f \mid \{\text{later}(\blacksquare) = v\} = (a \triangleright s) \mid \{\text{later}(\blacksquare) = v\},$$

which is also equal to $\text{kind}(A \triangleright S \mid \{B = b\})$.

3. Transform the constrained Kind to focus on the early stage of the system

$$\text{early}(f \mid \{\text{later}(\blacksquare) = v\}).$$

The latter is exactly $\text{kind}(A \mid \{B = v\})$.

In `frplib`, we can write this simply as

```
pgd> early = Proj[:(n+1)]
pgd> later = Proj[(n+1):]
pgd> early @ kind(A >> S) | (later == v)
```

where the three steps correspond to

1. `kind(A >> S)`, the Kind of both stages jointly
2. `kind(A >> S) | (later == v)`, constraining with the observation, and
3. `early @ kind(A >> S) | (later == v)`, projecting onto the components of interest.

(Recall that slices `i:j` in Python do not include the final index `j`.)

Taken together, these steps form an operation called **Bayes's Rule** in which we **learn more about the early components of a join from observation of later components**. The `frplib` convenience function `bayes(observed_b, a, b_given_a)`

carries out this operation in one step, where `observed_b` is the observed value of the later-stage output, `a` is the Kind of the early stage, and `b_given_a` is the conditional Kind of later stage given the early stage. The result is the Kind¹⁴ of the FRP representing the early stage constrained by the observation of the later stage.

¹⁴`bayes` works with FRPs as well but we almost always use it with Kinds.

To see this in action, we revisit Example 4.5 about disease testing. We know the prevalence of a disease in the population (1/1000), the sensitivity of the test (ability to correctly detect someone with the disease, 950/1000), and the specificity of the test (ability to correctly determine someone does not have the disease, 990/1000). We specify that information in `frplib` as follows.

```
pgd> has_disease = choice(0, 1, 1/999) # No disease has higher weight
pgd> test_by_status = conditional_kind({
...>   0: choice(0, 1, 1/99), # No disease, negative higher weight
...>   1: choice(0, 1, 19)    # Yes disease, positive higher weight
...> })
pgd> dStatus_and_tResult = has_disease >> test_by_status
pgd> dStatus_and_tResult

      ,---- 98901/100000 ---- <0, 0>
      |---- 999/100000 ---- <0, 1>
<> -|
      |---- 1/20000 ---- <1, 0>
      `---- 19/20000 ---- <1, 1>

pgd> Disease_Status = Proj[1] # Naming this statistic
pgd> Test_Result = Proj[2]    # ...and this statistic
```

This produces a Kind with two components that we name to aid understanding. Our question is: if someone tests positive how likely are they to actually have the disease? Think for a moment about how you can do this in `frplib` using the three steps above. Given the value of the `Test_Result` component, what can we infer about the `Disease_Status` component?

Puzzle 72. Try to craft a single expression in `frplib` to answer our main question: if someone tests positive how likely are they to have the disease? (Do not use the `bayes` function for this puzzle.)

Find the explicit correspondence between quantities in this example (e.g.,

`has_disease, test_by_status, Test_Result`) and quantities defined in the ingredient list above (e.g., S , a , B).

We can answer our question with an observation of the test result and a projection onto disease status.

```
pgd> Disease_Status @ dStatus_and_tResult | (Test_Result == 1)
,---- 999/1094 ---- 0    # No disease
<> -|
`---- 95/1094 ---- 1    # Yes disease
```

We restrict attention to values for which the test is positive (our observation) and then marginalize to look only at disease status. That's it. That's Bayes's Rule. We can rewrite this in terms of the `bayes` function with the observed value of the test as the first argument:

```
pgd> bayes(1, has_disease, test_by_status)
,---- 999/1094 ---- 0    # No disease
<> -|
`---- 95/1094 ---- 1    # Yes disease
```

Either way, the result may be surprising: despite a positive test, the probability that the patient has the disease is *small*.

The small weight on having the disease even with a positive test result derives from the low baseline prevalence of the disease in the population, i.e., the small weight on $\langle 1 \rangle$ in the Kind `has_disease`. Work out carefully how this result was derived. The amazing thing is how simple it is; we just exclude branches and renormalize. The statistic `Disease_Status` selects one component, and given that the test result is *known* the other component is not even that interesting. (Take a look at the Kind tree without the marginalizing projection

```
dStatus_and_tResult | (Test_Result == 1)
```

to see this.)

Bayes's Rule has a nice expression in terms of Kind procedures as well. For instance,

```
pgd> @kind
...> def status_when_positive():
...>     dis = yield has_disease
...>     pos = yield test_by_status(dis)
```

```
...>   yield given(pos == 1)
...>   return dis
```

which gives the same Kind as that shown above. For each possible value of the disease status, we look at each possible value of the test result for that status, accounting for the weights of both in the background. We exclude cases with a negative test (via the `given` line) and collect the disease status values. This produces the Kind with the appropriate weights automatically aggregated.

Bayes's Rule gives us the Kind of the FRP representing the early stage updated for the observation of the system's later stage. This combines three of our operations: a join to describe the early and late stage together, a constraint with an observation of the later stage, and a transformation to extract the Kind of the early stage. Using the ingredient list above, Bayes's Rule is expressed in general as follows, algebraically and procedurally.

Bayes's Rule. The Kind of the system's early stage output constrained with the observation that the later-stage output is value v ,

$$a_v = \text{kind}(A \mid \{B = v\}),$$

is computed with the expression

$$a_v = \text{early}(a \triangleright s \mid \{\text{later}(\blacksquare) = v\}) \quad (5.14)$$

or equivalently by the Kind procedure

```
Kind  $a_v$ :
   $u \leftarrow a$ 
   $v' \leftarrow s(u)$ 
  given ( $v' = v$ )
  return  $u$ 
```

In `frplib`, these look almost the same:

```
early_given_later = early( (a >> s) | (later == v) )
```

and

```
@kind
```

```
def early_given_later():
    u = yield a
    v_prime = yield s(u)
    yield given(v_prime == v)
    return u
```

Both the expression and procedure define the same Kind.

What makes Bayes's Rule so useful is that the evolution of the system somewhat obfuscates the relationship between the later-stage value and the early-value. All we are doing is controlling for the later-stage value, eliminating all other possibilities from consideration.

We can similarly infer the later-stage value from the early-stage value

```
later_given_early = later( (a >> s) | (early == u) )
```

and

```
@kind
def later_given_early():
    u_prime = yield a
    v = yield s(u_prime)
    yield given(u_prime == u)
    return v
```

But notice that `later_given_early` is just the Kind `s(v)`. That is exactly what the conditional Kind is doing: producing the Kind of the FRP representing the later-stage output of the system for each value of the early-stage output. So this direction of inference is “more obvious” because we explicitly built the system from `k` and `m`. We do not need a special “rule” for that.

BAYES'S RULE IN TERMS OF KERNELS. Bayes's Rule is nicely described both algebraically and procedurally, but we can also give a mathematical expression in terms of the *kernels* of the relevant Kinds. This arises simply by combining our three operations: a join, an observational constraint, and a transformation. Following the notation given on page 269, we write $s(v \mid u)$ for the kernel of the Kind $s(u)$ evaluated at v .

Let's take the three steps in order.

1. *The Join* ($a \triangleright s$).

$$(a \triangleright s)(u :: v') = a(u) s(v' \mid u).$$

See equation (4.18).

2. *The Constraint with Observation* (given $\{\text{later}(\blacksquare) = v\}$). Let h be the Kind from step 1.

$$(h \mid \{\text{later}(\blacksquare) = v\})(u :: v') = \frac{h(u :: v') \{v' = v\}}{\sum_{u'', v''} h(u'' :: v'') \{v'' = v\}},$$

where the sum in the denominator is over all possible values for the full system's output. See equation (5.10).

3. *The Transformation* (the projection **early**). Let g be the Kind from step 2.

$$(g \hat{\text{early}})(u) = \sum_{v'} g(u :: v')$$

See equations (2.2) and (2.9).

Let $\mathbf{r}$ be the conditional Kind of the early-stage given the observation the later-stage output. Putting the three steps together, we get an expression for its kernel:

$$\begin{aligned} \mathbf{r}(u \mid v) &= \frac{\sum_{v'} a(u) \mathbf{s}(v' \mid u) \{v' = v\}}{\sum_{u', v''} a(u') \mathbf{s}(v'' \mid u') \{v'' = v\}} \\ &= \frac{a(u) \mathbf{s}(v \mid u)}{\sum_{u'} a(u') \mathbf{s}(v \mid u')}, \end{aligned}$$

where the sums in numerator and denominator simplify because the indicator selects for $v' = v$.

Bayes's Rule (Kernel Version)

$$\mathbf{r}(u \mid v) = \frac{a(u) \mathbf{s}(v \mid u)}{\sum_{u'} a(u') \mathbf{s}(v \mid u')}. \quad (5.15)$$

For example, in the Disease Status example above, let d be the Kind of the FRP representing the patient's disease status (**has_disease** above), $\mathbf{t}$ be the conditional Kind of the patient's test given disease status (**test_by_status** above), and $\mathbf{q}$ be the conditional Kind of the FRP representing the patient's disease status given their test

outcome. By Bayes's Rule (equation eq:bayes-rule-kernel), the kernel of $\mathbf{q}$ is

$$\begin{aligned} \mathbf{q}(0 \mid 0) &= \frac{d(0) \mathbf{t}(0 \mid 0)}{d(0) \mathbf{t}(0 \mid 0) + d(1) \mathbf{t}(0 \mid 1)} = \frac{0.999 \cdot 0.99}{0.999 \cdot 0.99 + 0.001 \cdot 0.05} \approx 0.99995 \\ \mathbf{q}(1 \mid 0) &= \frac{d(1) \mathbf{t}(0 \mid 1)}{d(0) \mathbf{t}(0 \mid 0) + d(1) \mathbf{t}(0 \mid 1)} = \frac{0.001 \cdot 0.05}{0.999 \cdot 0.99 + 0.001 \cdot 0.05} \approx 0.00005 \\ \mathbf{q}(0 \mid 1) &= \frac{d(0) \mathbf{t}(1 \mid 0)}{d(0) \mathbf{t}(1 \mid 0) + d(1) \mathbf{t}(1 \mid 1)} = \frac{0.999 \cdot 0.01}{0.999 \cdot 0.01 + 0.001 \cdot 0.95} \approx 0.91316 \\ \mathbf{q}(1 \mid 1) &= \frac{d(1) \mathbf{t}(1 \mid 1)}{d(0) \mathbf{t}(1 \mid 0) + d(1) \mathbf{t}(1 \mid 1)} = \frac{0.001 \cdot 0.95}{0.999 \cdot 0.01 + 0.001 \cdot 0.95} \approx 0.08684, \end{aligned}$$

giving the same results as above.

We can express equation (5.15) in terms of FRPs. If we decompose a join $X \triangleright S = X :: Y$ into two components, then by Definition 27, we can write S as $(Y \mid X)$ and the kernel of its conditional Kind can be written $\mathbf{k}_{Y \mid X}$. From this, we can derive the conditional FRP $(X \mid Y)$, with kernel $\mathbf{k}_{X \mid Y}$ for its conditional Kind. We can rewrite Bayes's Rule (5.15) directly in terms of these kernels and the kernel $\mathbf{k}_X$ for $\text{kind}(X)$.

Bayes's Rule (Kernel Version, expressed with FRPs) Define FRPs X and Y through $X \triangleright S = X :: Y$. Then, Bayes's Rule can be written in terms of kernels as:

$$\mathbf{k}_{X \mid Y}(u \mid v) = \frac{\mathbf{k}_X(u) \mathbf{k}_{Y \mid X}(v \mid u)}{\sum_{u'} \mathbf{k}_X(u') \mathbf{k}_{Y \mid X}(v \mid u')}. \quad (5.16)$$

In the disease-testing example, let FRP D be the patient's disease status and FRP T be the patient's test result. Equation (5.16) gives us

$$\mathbf{k}_{D \mid T}(s \mid t) = \frac{\mathbf{k}_D(s) \mathbf{k}_{T \mid D}(t \mid s)}{\mathbf{k}_D(0) \mathbf{k}_{T \mid D}(t \mid 0) + \mathbf{k}_D(1) \mathbf{k}_{T \mid D}(t \mid 1)},$$

which is the equal to our expressions above.

Key Take Aways

As a random system evolves, we observe the output of a later stage and want to use that observation to learn about output the earlier stage output. *Bayes's Rule* is the method we use do that.

We have an FRP A of dimension n with Kind a representing the early-stage output and a conditional FRP S of type $n \rightarrow n+p$ of the later stage output given the early stage, with conditional Kind s . The later-stage output is $B = \text{later}(A \triangleright S)$ where $\text{early} = \text{proj}_{..n}$ and $\text{later} = \text{proj}_{(n+1)..}$.

Bayes's Rule gives us $a_v = \text{kind}(A \mid \{B = v\})$ via the Big 3+1

$$a_v = \text{early}(a \triangleright s \mid \{\text{later}(\blacksquare) = v\}) \quad (5.17)$$

or equivalently by the Kind procedure

```
Kind  $a_v$ :
   $u \leftarrow a$ 
   $v' \leftarrow s(u)$ 
  given ( $v' = v$ )
  return  $u$ 
```

Bayes's Rule can be expressed in terms of Kernels. Let r be the conditional Kind of the early-stage given the observation the later-stage output. We get

$$r(u \mid v) = \frac{a(u) s(v \mid u)}{\sum_{u'} a(u') s(v \mid u')}. \quad (5.18)$$

Expressed in terms of FRPs X and Y with $X \triangleright S = X :: Y$ for some conditional FRP S , this translates to

$$k_{X|Y}(u \mid v) = \frac{k_X(u) k_{Y|X}(v \mid u)}{\sum_{u'} k_X(u') k_{Y|X}(v \mid u')}. \quad (5.19)$$

5.5 Observations and Independence

In Section 4.2, we introduced the concept of **independence**, with two perspectives. The components that comprise a system are independent if

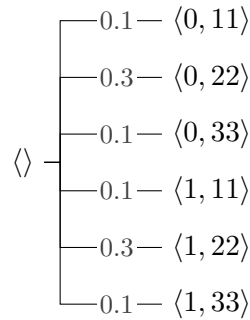
- we can analyze the full system by combining the separate analyses of the components; or
- observing the output of any of the components does not change our predictions about the remaining components.

These perspectives are equivalent, though that will not yet be obvious. We saw how the first perspective is related to the structure of independent joins (e.g., via equation 4.3), and in this Section, we show how the second perspective is expressed in terms of observational constraints.

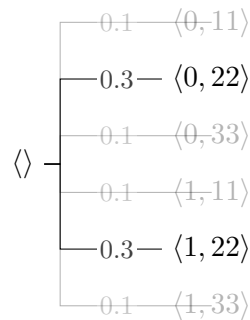
To motivate this perspective, let us start with a situation in which we have two independent components:

```
pgd> Z = frp( choice(0, 1) * weighted_as(11, 22, 33, weights=[2, 6, 2]) )
pgd> X = Z ^ Proj[1]
pgd> Y = Z ^ Proj[2]
```

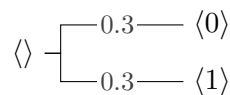
We have constructed Z explicitly so that $Z = X \star Y$, so X and Y are independent components of Z . Z has Kind



What happens if we observe a value of Y , say? Pick any value and try it. Here, I will suppose that we have the observation that $Y = 22$. Constraining with this observation eliminates inconsistent branches,

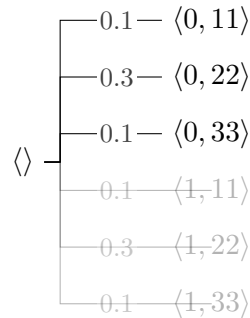


and then we transform with proj_1 statistic to get $\text{kind}(X \mid \{Y = 22\})$

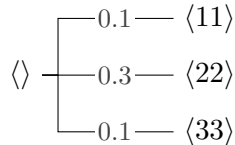


which simply drops the second component. The resulting Kind is equivalent to `choice(0, 1)`, which is `kind(X)`. The Kind `kind(X | {Y = 22})` encapsulates our knowledge of X when we have for a fact observed that Y has value 3. The Kind `kind(X)` encapsulates our knowledge of X without that information. That the two are equivalent means that the information about Y did not change our knowledge or predictions about X .

Try it with an observation of X , say that it is 0. We eliminate the branches that are inconsistent with this observation



and then transform with `proj2` to get `kind(Y | {X = 0})`



which is equivalent to `kind(Y)`. Again, observing X did not change our predictions about Y .

Now to confirm independence, we need to check this for all possible values of X or all possible values of Y . (We do not need to check both for reasons we will see.) Try this.

Puzzle 73. Confirm that X and Y are independent here using observational constraints for either all values of X or all values of Y .

If observing even one value of X or Y gives us information about the other, then X and Y are dependent. Let's see what that would look like.

```
pgd> b_given_a = conditional_kind({
...>   1: choice(-10, 10),
...>   2: weighted_as(-10, 0, 10, weights=[4996, 8, 4996]),
```

```

...> 3: choice(-10, 10),
...> 4: choice(-10, 10)
...> })
pgd> C = frp( uniform(1, 2, 3, 4) >> b_given_a )
pgd> A = C ^ Proj[1]
pgd> B = C ^ Proj[2]

```

Without even doing any additional calculations, we can see here that A and B will *not* be independent. The reason is that the join that creates the second component of C , namely B , quite literally distinguishes among the possible values of A . When A is 1, 2, or 4, B is generated with the Kind `choice(-10, 10)`. But when A is 3, a small weight also goes on 0.

We can confirm this reasoning by directly calculation. Try these yourself to get `kind(B | {A = a})` for each $a \in [1..4]$:

```

pgd> Proj[2] @ kind(C) | (Proj[1] == 1)
pgd> Proj[2] @ kind(C) | (Proj[1] == 2)
pgd> Proj[2] @ kind(C) | (Proj[1] == 3)
pgd> Proj[2] @ kind(C) | (Proj[1] == 4)

```

You should see here exactly the four target Kinds in the conditional Kind `b_given_a`. Though close, none is in fact equivalent to

```

pgd> kind(B)
,---- 0.4999 ---- -10
<> -+---- 0.0002 ---- 0
`---- 0.4999 ---- 10

```

Though it is less apparent from inspection, we can see this dependence in the other direction as well

```

pgd> Proj[1] @ kind(C) | (Proj[2] == -10)
,---- 0.25005 ---- 1
|---- 0.24985 ---- 2
<> -|
|---- 0.25005 ---- 3
`---- 0.25005 ---- 4
pgd> Proj[1] @ kind(C) | (Proj[2] == 0)
<> ----- 1 ---- 2
pgd> Proj[1] @ kind(C) | (Proj[2] == 10)

```

```

,---- 0.25005 ---- 1
|---- 0.24985 ---- 2
<> -|
|---- 0.25005 ---- 3
`---- 0.25005 ---- 4
pgd> kind(A)
,---- 1/4 ---- 1
|---- 1/4 ---- 2
<> -|
|---- 1/4 ---- 3
`---- 1/4 ---- 4

```

So, A and B are not independent.

This generalizes to FRPs of any dimension and to any number of FRPs. To keep things clear, we begin with the case of two FRPs. Recall that if $Z = X :: Y$ with components¹⁵ of dimensions d and $n - d$

¹⁵See Definition 14.

$$\begin{aligned} \text{kind}(X \mid \{Y = y\}) &= \text{kind}\left(\text{proj}_{..d}(Z \mid \{\text{proj}_{(d+1)..}(\blacksquare) = y\})\right) \\ \text{kind}(Y \mid \{X = x\}) &= \text{kind}\left(\text{proj}_{(d+1)..}(Z \mid \{\text{proj}_{..d}(\blacksquare) = x\})\right). \end{aligned}$$

If Z is an FRP of dimension n with $d \in [1..n)$ and

$$\begin{aligned} X &= \text{proj}_{..d}(Z) \\ Y &= \text{proj}_{(d+1)..}(Z), \end{aligned}$$

then X and Y are independent if and only if *either* of the following holds:

1. For all possible values y of Y ,

$$\text{kind}(X \mid \{Y = y\}) \cong \text{kind}(X). \quad (5.20)$$

2. For all possible values x of X ,

$$\text{kind}(Y \mid \{X = x\}) \cong \text{kind}(Y). \quad (5.21)$$

These equations mean that observation of one FRP **does not change our knowledge and predictions** of the other.

In these equations, we wrote $\cong$ for equivalence of Kinds, but in practice we usually elide that distinction, write $=$, and identify equivalent Kinds as equal.

With multiple FRPs, the bookkeeping in the notation becomes more complicated, but the idea is the same. If FRPs $U_1, U_2, \dots, U_m$ are the components of an FRP Z in Definition 14. Then, the $U_1, U_2, \dots, U_m$ are independent if **observing the values of any subset of them does not change our knowledge and predictions about the others**.

To formalize this, let p_i for $i \in [1..m]$ be the projection statistics such that $U_i = p_i(Z)$. Let $\mathcal{I} \subset [1..m]$ denote a *non-empty* set of indices of the U_i 's that we observe, and let v_i for $i \in \mathcal{I}$ be corresponding observed values. Define $\mathcal{J} = [1..m] - \mathcal{I}$ to be the non-empty subset of indices for U_j 's we do not observe,¹⁶ and define $p_{\mathcal{J}}$ to be the projection onto those in $\mathcal{J}$ in increasing order. The condition

¹⁶Note that $\mathcal{I} \cup \mathcal{J} = [1..m]$.

$$\xi(x) = \left\{ \bigwedge_{i \in \mathcal{I}} p_i(x) = v_i \right\},$$

is true precisely when all¹⁷ the observed components equal the observed values. Then, we have the following.

¹⁷ $\bigwedge$ is logical-and here.

The $U_1, U_2, \dots, U_m$ are independent if for all non-empty $\mathcal{I} \subset [1..m]$ and possible values v_i of U_i for $i \in \mathcal{I}$:

$$\text{kind}(p_{\mathcal{J}}(Z) \mid \xi) \cong \text{kind}(p_{\mathcal{J}}(Z)), \quad (5.22)$$

for non-empty set $\mathcal{J} = [1..m] - \mathcal{I}$.

Here, $p_{\mathcal{J}}(Z)$ has as components all the unobserved U_j 's. So equation (5.22) says that our knowledge about the unobserved U_j 's having observed the values of the other U_i 's is the same (up to equivalence) as our knowledge without that information. Again, we write $\cong$ formally for equivalence, but we treat equivalent Kinds as equal in practice. It is fine to write it that way.

The independence of a collection is a stronger assumption than the independence of all pairs in a collection. Each pair of FRPs can be independent without the full collection being independent.

Example 5.5.

Define an FRP F representing two flips of a balanced coin and derive an

FRP P (for parity) that is 1 if the number of heads in those flips is odd, else 0.

```
pgd> F = frp( choice(0, 1) ** 2 )
pgd> FP = F ^ Fork(__, (Proj[1] + Proj[2]) % 2)
pgd> P = FP ^ Proj[3]
```

FP has dimension 3, with third component equal to the sum of the two flips (with heads as 1) modulo 2.

If we look at the Kinds of any pair of these components, we get `choice(0, 1) ** 2`, meaning that the two components are independent.

```
pgd> kind(FP ^ Proj[1, 2])
,---- 1/4 ---- <0, 0>
|---- 1/4 ---- <0, 1>
<> -|
|---- 1/4 ---- <1, 0>
`---- 1/4 ---- <1, 1>
```

```
pgd> kind(FP ^ Proj[1, 3])
,---- 1/4 ---- <0, 0>
|---- 1/4 ---- <0, 1>
<> -|
|---- 1/4 ---- <1, 0>
`---- 1/4 ---- <1, 1>
```

```
pgd> kind(FP ^ Proj[2, 3])
,---- 1/4 ---- <0, 0>
|---- 1/4 ---- <0, 1>
<> -|
|---- 1/4 ---- <1, 0>
`---- 1/4 ---- <1, 1>
```

But if you *observe* that the third component equals 0, you learn something:

```
pgd> kind(Proj[1, 2] @ FP | (Proj[3] == 0))
,---- 1/2 ---- <0, 0>
<> -|
`---- 1/2 ---- <1, 1>
```


Specifically, if you know that there is an even number of heads, you must have 0 or 2. Our knowledge of the two flips has changed based on this observation.

Key Take Aways

In this section, we formalize the perspective on independence where FRPs are independent if observing the values of any of them **does not change our knowledge or predictions of the others**.

For two FRPs, this is described as follows. Equation (5.22) describes the general case, which is notationally more complex but has the same idea.

If $X :: Y$ is an FRP, then X and Y are independent if and only if *either* of the following holds:

1. For all possible values y of Y ,

$$\text{kind}(X \mid \{Y = y\}) \cong \text{kind}(X). \quad (5.23)$$

2. For all possible values x of X ,

$$\text{kind}(Y \mid \{X = x\}) \cong \text{kind}(Y). \quad (5.24)$$

Here $\cong$ is Kind equivalence, but in practice, we usually write this as $=$, identifying equivalent Kinds.

5.6 Observations and Conditional Kinds

Consider again the disease-testing example, where a patient who may or may not have the disease takes a test for the presence of the disease. We did

```
pgd> has_disease = choice(0, 1, 1/999) # No disease has higher weight
pgd> test_by_status = conditional_kind({
...>   0: choice(0, 1, 1/99), # No disease, negative higher weight
...>   1: choice(0, 1, 19)    # Yes disease, positive higher weight
...> })
pgd> dStatus_and_tResult = has_disease >> test_by_status
pgd> Disease_Status = Proj[1]      # Naming this statistic
```

```
pgd> Test_Result = Proj[2]          # ...and this statistic
```

Here, `test_by_status` is a *conditional Kind* because it specifies a Kind that is contingent on some other specified value, namely the patient's disease status. We call it the conditional Kind of the patient's test result given the patient's disease status. That this

For instance, when you evaluate

```
Test_Result @ dStatus_and_tResult | (Disease_Status == 0)
Test_Result @ dStatus_and_tResult | (Disease_Status == 1)
```

you will see that these are just the target Kinds `test_by_status(0)` and `test_by_status(1)`, respectively. *The conditional Kind gives for each input the Kind obtained by constraining with the observation that that input was observed!*

We can likewise create the conditional Kind `status_by_test` of the disease status given the test results using our calculation above:

```
pgd> status_by_test = conditional_kind({
...>    0: Disease_Status @ dStatus_and_tResult | (Test_Result == 0),
...>    1: Disease_Status @ dStatus_and_tResult | (Test_Result == 1)
...> })
```

Puzzle 74. Let `test_result = Test_Result(dStatus_and_tResult)` be the marginal Kind of the patient's test result. Then the Kinds `has_disease >> test_by_status` `test_result >> status_by_test` both describe the patient's disease status and test result jointly. How are they related? Can you transform one into the other?

This observation generalizes. Suppose that a is a Kind of dimension d and s a compatible conditional Kind. From any possible *value* of $a \triangleright s$, we can extract the early-stage value with `Proj[: (d+1)]` and later-stage value with `Proj[(d+1):]`.¹⁸

```
pgd> early = Proj[: (d+1)]
pgd> later = Proj[(d+1):]
```

Then for every possible value u of a :

```
Kind.equal( a, early(a >> s) )
Kind.equal( s(u), later @ (a >> s) | (early == u) )
```

are both `True`. In other words, for the conditional Kind s , $s(u)$ is the Kind of the later value constrained by the observation that the early value equals u . So s just packages all the constrained Kinds given each early value. Mathematically, we can state this precisely in the same way.

¹⁸Recall that `Proj[i: (j+1)]` is a statistic that extracts components $i, i+1, \dots, j$. In math, we write this as $\text{proj}_{i..j}$.

Suppose a is a Kind of dimension d and s is a compatible conditional Kind of type $d \rightarrow n$ for some n , then

$$a = \text{proj}_{..d}(a \triangleright s) \quad (5.25)$$

and for every value u of a ,

$$s(u) = \text{proj}_{(d+1)..}(a \triangleright s \mid \{\text{proj}_{..d}(\blacksquare) = u\}). \quad (5.26)$$

The join $a \triangleright s$ is the Kind of both stages *jointly*, and from that we can define the conditional Kind of either stage given the other. This reveals Bayes's Rule (section 5.4) to be just a version of this mundane operation.

Define $k = a \triangleright s$ to be the Kind of the full system and let r denote the conditional Kind of the early stage given the later stage. Then

$$s(u) = \text{later}(k \mid \{\text{early}(\blacksquare) = u\}) \quad (5.27)$$

$$r(v) = \text{early}(k \mid \{\text{later}(\blacksquare) = v\}). \quad (5.28)$$

We can also define these as conditional Kind procedures:

conditional Kind $s(u)$:

```
 $x \leftarrow k$ 
 $\leftarrow \text{given}(\text{early}(x) = u)$ 
return  $\text{later}(x)$ 
```

conditional Kind $r(v)$:

```
 $x \leftarrow k$ 
 $\leftarrow \text{given}(\text{later}(x) = v)$ 
return  $\text{early}(x)$ 
```

Key Take Aways

Conditional Kinds describe the Kind of an FRP produced contingently on another FRP. We tend to think of these later- and early-stage outputs of a random system, though this is a metaphor. A conditional Kind can describe any components of a system.

We can construct a conditional Kind explicitly by constraining a join FRP that describes the full system output with the observation of either stage.

See equations (5.14), (5.27), and (5.28).

Chapter Checkpoints

After reading this Chapter you should be able to:

- Explain in broad terms how we account for partial information about the value of an FRP, observed or hypothetical.
- Define an event and explain what it means for an event to occur or not occur.
- Show how to combine events with logical-and and logical-or (and logical-not).
- Show how to convert a Boolean expression to an event and how to denote the corresponding FRP.
- Write a Kind or FRP constrained by an observation.
- Explain what an *FRP given a condition* and a *Kind given a condition* mean.
- Describe how to find $k \mid \zeta$ in canonical form when you are handed a Kind k in canonical form and a compatible condition ζ .
- Identify the difference, if any, between $k \mid \top$ and k for a Kind k .
- Explain the purpose of Bayes's Rule, and describe the steps that comprise it.
- Apply Bayes's Rule in `frplib`.

Predicting with Expectations

6

Chapter

Contents

6.1	Fundamental Properties of Risk-Neutral Prices	390
6.2	Computing Expectations	414
6.3	Kinds and Expectations	425
6.4	Probabilities are the Expectations of Events	431

The more we know about a process, the better we are able to predict its outcome. Indeed, we can think of *uncertainty* as quantifying the difficulty of making predictions. Uncertainty reflects limits to the accuracy with which we can predict an outcome. Sometimes these limits arise from our lack of information and sometimes they are intrinsic features of the system we are studying.

At one extreme, an outcome may be certain, and our prediction is perfect. For example, the constant FRP with Kind $\langle \rangle$ ——— $\langle 100 \rangle$ has only possible value (100 in this case), so we can predict with complete certainty that 100 is the value it will display when we push its button. Close to that is what we can call **essential certainty**. It is *possible* that all the air molecules in the room where you are sitting will spontaneously organize themselves in the corner of the room, leaving you in an effective vacuum, but for that to happen would require so many miraculous bounces that there is no reasonable need to factor that possibility into your day.

Toward the other extreme, an outcome may be uncertain with nothing to distinguish the possibilities. For instance, FRPs with Kind

$$\langle \rangle \begin{cases} 1 \text{ — } \langle 0 \rangle \\ 1 \text{ — } \langle 1 \rangle \end{cases}$$

can produce values 0 or 1, and in any sample of such FRPs there is no reason to expect one more than the other. Our uncertainty increases with the number and spread of the possible values. For instance, FRPs with Kinds

$$\begin{array}{cc} \langle \rangle \begin{cases} 1 \text{ — } \langle -3 \rangle \\ 1 \text{ — } \langle -2 \rangle \\ 1 \text{ — } \langle -1 \rangle \\ 1 \text{ — } \langle 0 \rangle \\ 1 \text{ — } \langle 1 \rangle \\ 1 \text{ — } \langle 2 \rangle \\ 1 \text{ — } \langle 3 \rangle \end{cases} & \langle \rangle \begin{cases} 1 \text{ — } \langle -10000 \rangle \\ 1 \text{ — } \langle -100 \rangle \\ 1 \text{ — } \langle -1 \rangle \\ 1 \text{ — } \langle 0 \rangle \\ 1 \text{ — } \langle 1 \rangle \\ 1 \text{ — } \langle 100 \rangle \\ 1 \text{ — } \langle 10000 \rangle \end{cases} \end{array}$$

are both harder to predict than the previous case, and those on the right are harder than those on the left because the distances between values are larger. We can further increase that uncertainty without bound with ever more complicated Kinds.

This raises three questions. First, how should we make predictions in the face of uncertainty? Second, how should we quantify the degree of uncertainty that we face?

And third, how should our decisions be affected by the degree of uncertainty? Here we will focus on the first question, touching only briefly on the other two, but rest assured we will consider all three as we proceed.

The goal of this Chapter is to define a baseline best prediction for the value of an FRP. We express this prediction through the *price* that we would pay to receive the FRP's payoff. Using prices to describe a prediction has a long tradition. The price of a stock, for instance, is (in theory) the market's prediction of the long-run value of each share of the company. When a sports team signs a contract for a player, they are predicting how much revenue (explicitly and implicitly through championships, merchandise, advertising, et cetera) that player will bring to the organization. When an insurance company offers insurance against an event, such as damage to one's home, the price of the insurance premiums reflects the company's prediction about how much they will have to pay out.¹

The last example has a resemblance to what we face with FRPs. An insurance company makes their money *in the aggregate*. An individual homeowner's policy may or may not require a payout, but with good predictions, the company can price the premium to make a profit on a large collection of policies.²

Similarly, with one particular FRP, we can get any of its possible payoffs, with a large enough collection of FRPs of the same Kind, we will see all of its possible payoffs in proportions close to the weights. We can thus control our gains and losses in the aggregate with the choice of price. An important implication is that our prediction about an FRP's value depends only on the *Kind* of the FRP. In effect, we are making predictions about Kinds.³

Our best prediction of an FRP's value will be represented by the **risk-neutral price** for an FRP of that Kind, to be defined below. Here is the setup.

1. We can get from the FRP Marketplace as many FRPs of any Kind.
2. We purchase some number of FRPs of Kind k , paying a price $\$c_k$ per unit, and our total payoff is the sum of the values of all the purchased FRPs.⁴
3. We can *borrow without interest* as much money as we like to purchase FRPs, but we must pay back that loan when their payoffs are revealed.

Our first task is to understand how the choice of price c_k affects our gains or losses in the aggregate. To start, we focus exclusively on *scalar* (1-dimensional) FRPs.

Consider the simplest, non-trivial FRP: *constants*, with Kinds of the form $\langle \rangle \text{ --- } 1 \text{ --- } \langle v \rangle$ for some number v . We know with *certainty* that this will payoff $\$v$. If you pay less than $\$v$, you will make a profit on each FRP you purchase, so you would purchase as many as possible. Of course, the Marketplace knows this too and

¹And the companies have armies of analysts, called actuaries, whose job is to make those predictions based on the available data.

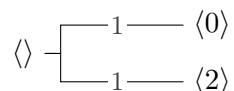
²This assumes that the different policies are close to *independent*; if all of the homes are hit by the same hurricane, the company will lose.

³With an observation c from partial information about an FRP X , we get a new FRP $X \mid c$ and predictions of its value depend on its Kind $\text{kind}(X \mid c)$.

⁴Remember that negative payoffs means that we have to pay out.

thus would not sell such an FRP for less than $\$v$. If you pay more than $\$v$, you will lose money on each FRP you purchase. *You* know this and so would not buy any at such a price. The only reasonable price for these FRPs is $\$v$, which is exactly their payoff. Hence, for all practical purposes, owning an FRP with Kind `constant(v)` is equivalent to having $\$v$.

Consider next simple FRPs with Kind



Use the `frp market` application⁵ to purchase collections of these at different prices. With the `buy` task, you specify how many FRPs you want to buy at each of one or more prices, and the Kind. It shows your net payoff (total and per unit) for the batch purchased at each price. Here are some examples with sample output, where I have omitted some of the Kind displays.

⁵Remember, this is still your free trial, so no money changes hands yet.

```
pgd> Market.buy(1_000_000, [0.5, 0.9, 0.99, 0.999, 0.9999, 1, 1.01], choice(0, 2))
```

Buying 1,000,000 FRPs with kind

```
,---- 1 ----- 0
<> -|
     `---- 1 ----- 2
```

at each price.

+-----+-----+-----+		
Price/Unit (\$)	Net Payoff (\$)	Net Payoff/Unit (\$)
+=====+		
0.50	500,670.00	0.500670
0.90	11,534.00	0.011534
0.99	537.00	0.000537
0.999	-1,990.00	-0.001990
0.9999	753.00	0.000753
1.00	-512.00	-0.000512
1.01	-8,796.00	-0.008796
+-----+-----+-----+		

```
pgd> Market.buy(10_000_000, [0.5, 0.9, 0.99, 0.999, 0.9999, 1, 1.01], choice(0, 2))
```

+-----+-----+-----+		
Price/Unit (\$)	Net Payoff (\$)	Net Payoff/Unit (\$)

```

+=====+=====+=====+
|      0.50   | 5,003,478.00 |      0.500348 |
|      0.90   |  97,792.00  |      0.099779 |
|      0.99   | 102,930.00  |      0.010293 |
|      0.999  |   2,311.00  |      0.000231 |
|      0.9999 |    96.00    |      0.000010 |
|      1.00   |  -2028.00   |     -0.000203 |
|      1.01   | -99,224.00  |     -0.009922 |
+-----+-----+-----+

```

```
pgd> Market.buy(100_000_000, [0.5, 0.9, 0.99, 0.999, 0.9999, 1, 1.01], choice(0, 2))
```

```

+-----+-----+-----+
| Price/Unit ($) | Net Payoff ($) | Net Payoff/Unit ($) |
+=====+=====+=====+
|      0.50   | 49,995,392.00 |      0.499953 |
|      0.90   |  9,976,452.00 |      0.099765 |
|      0.99   | 1,005,452.00  |      0.010055 |
|      0.999  |   103,884.00  |      0.001039 |
|      0.9999 |    24,664.00  |      0.000247 |
|      1.00   |    262.00     |      0.000003 |
|      1.01   | -998,284.00   |     -0.009983 |
+-----+-----+-----+

```

```
pgd> Market.buy(1_000_000_000, [0.5, 0.9, 0.99, 0.999, 0.9999, 1, 1.01], choice(0, 2))
```

```

+-----+-----+-----+
| Price/Unit ($) | Net Payoff ($) | Net Payoff/Unit ($) |
+=====+=====+=====+
|      0.50   | 499,980,344.00 |      0.499803 |
|      0.90   | 99,964,150.00  |      0.099964 |
|      0.99   | 9,939,632.00   |      0.009940 |
|      0.999  | 1,001,286.00   |      0.001001 |
|      0.9999 |    80,598.00   |      0.000081 |
|      1.00   | -38,850.00     |     -0.000039 |
|      1.01   | -10,088,852.00 |     -0.100889 |
+-----+-----+-----+

```

Although it is not perfectly clearcut, there is a pattern here. When the price is low, the net payoff tends to be large and positive. The net payoff shrinks as the price

approaches 1, becoming more and more negative beyond that. The third column makes it easier to see the common pattern because the numbers across runs are all on the same “per unit scale.”

Let’s zoom in near 1 to get a closer look:

```
pgd> Market.buy(1_000_000_000, [0.9999, 0.99999, 1.00, 1.00001, 1.0001], choice(0, 2))
```

(Due to large numbers, the values below may be slightly approximate.)

+-----+-----+-----+			
Price/Unit (\$) Net Payoff (\$) Net Payoff/Unit (\$)			
+=====+			
	0.9999	101,695,118	1.01695e-4
	0.99999	8,953,455	8.95346e-6
	1.00	915,029	9.15030e-7
	1.00001	-10,020,272	-1.00203e-5
	1.0001	-99,822,037	-9.9822e-05
+-----+-----+-----+			

The pattern seems similar, and it appears that 1 is the inflection point. We can guess that \$1 is the right price! And we’ll see below how this might match our intuition that 1 is the midpoint between two evenly weighted values 0 and 2.

If you were offered a price of < 1 for FRPs of the previous two example Kinds, the interest-free loan from the FRP Marketplace would let you make a profit with essential certainty (for a large enough batch). We give such prices a name.

Definition 32. If X is a scalar FRP, an **arbitrage price** for X is a number c such that if you pay $\$c$ per FRP, you can purchase a collection of FRPs of Kind equivalent to $\text{kind}(X)$ and *guarantee* a profit with essential certainty.

If we pay an arbitrage price for any particular number of FRPs, we can still lose money. But if we buy *enough* FRPs of the same Kind, the possibility of losing money becomes like the possibility of your air all gathering in the corner of the room. A profit is essentially guaranteed. This matches the pattern we saw in the above examples: any price less than 1 for FRPs of those Kinds is an arbitrage price. *If* we were offered an arbitrage price to purchase FRPs, we would jump on the deal and purchase as many as possible.

Arbitrage prices have an important property. If c is an arbitrage price for X and $c' < c$, then c' is also an arbitrage price for X . When c is a price that essentially guarantees a profit, then paying a *smaller* price only makes it easier to make a profit,

and this smaller price is then also an arbitrage price. In fact, we can make a stronger statement:

Property A. If c is an arbitrage price for X , then there is some real number $\epsilon > 0$ so that every $c' < c + \epsilon$ is also an arbitrage price for X .

This looks a little more mysterious at first but is based on similar intuition. If we can make an essentially guaranteed profit at some price c , then we can very, very slightly increase the price and still make a profit, just a slightly smaller one. The increase might be tiny indeed – ϵ can be arbitrarily small – but we can always find a higher arbitrage price.

Be careful not to conclude from Property A that we can always find arbitrage prices that are arbitrarily large. Suppose 0.99 is an arbitrage price for a particular FRP. Property A tells us that we can find an arbitrage price that is slightly bigger than 0.99. Suppose then that values < 0.9901 are arbitrage prices. Then Property A tells us that we can find a value slightly bigger than 0.9901 that is also an arbitrage price. Suppose then that values < 0.99010001 are also arbitrage prices. We might continue in this way getting larger arbitrage prices, but the amount of increase at each step can get smaller and smaller. We might *never* reach 1, for instance. In most cases of interest, the set of arbitrage prices is bounded from above, and that is how we define the risk-neutral price.

Definition 33. If X is a scalar FRP, the **risk-neutral price** for X is the smallest value r that is bigger than every arbitrage price for X .

If every finite c is an arbitrage price for X , the risk-neutral price is ∞ . If no finite c is an arbitrage price, the risk-neutral price is $-\infty$.

If we pay the risk-neutral price for the FRPs, then we might make a profit or a loss, no matter how many FRPs we purchase. There are no guarantees. Remember that a positive price means that we pay to get the FRP; a negative price means that we are paid to take it.

The set of arbitrage prices for an FRP contains every number from $-\infty$ up to **but not including** the risk-neutral price r . No reasonable person would offer us an arbitrage price to purchase FRPs because it would (essentially) guarantee them a loss. Nor would you accept an offer to pay *more* than the risk-neutral price, for it would (essentially) guarantee you a loss. But at the risk-neutral price, neither buyer nor seller has the advantage.

The term *risk-neutral* here means that the price accounts only for typical payoff not for the magnitude of the losses that we risk. Consider FRPs with Kinds shown in Figure 6.1: all three have the same risk-neutral price of 10. Most of us facing a choice among these three payoffs would *not* be indifferent among them. The first guarantees a \$10 payoff. The second offers the possibility of a slightly higher payoff (\$11) at the small risk of losing \$1000 – a non-trivial loss. The third offers a bigger payoff with higher risk (losing \$10,000 is nothing to sneeze at). While you would pay \$10 for the first FRP, you would likely pay *less* for the latter two to account for the risk you face. Real betting markets account for this risk and tend to clear at prices lower than the risk neutral price. This risk matters in practice because you have limited funds available.

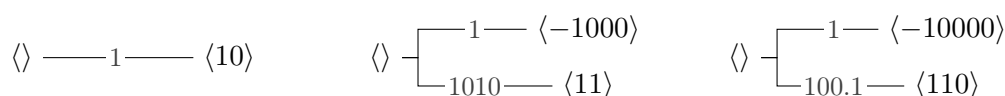
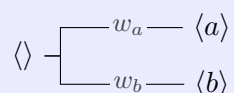


FIGURE 6.1. FRP Kinds with the same risk-neutral price. Are you indifferent to which of these payoffs you get?

But in our setup, at any price below the risk-neutral price, risk is not a consideration because you have *unlimited funds available* and can purchase an arbitrarily large number of FRPs. As such, the risk can be hedged away, and no premium for risk is needed. This pushes the equilibrium to the risk-neutral price.

Puzzle 75. Assuming $a < b$, can the risk-neutral price of the FRP with Kind



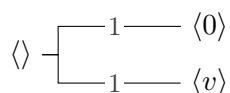
be less than a ? Greater than b ? Why or why not?

Can it be equal to a or b ? (Remember $w_a, w_b > 0$.) Why or why not?

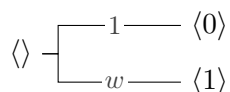
If w_b is very much bigger than w_a , do you expect the risk-neutral price to be closer to a or to b ?

Activity. Empirically evaluate the risk-neutral price of several scalar FRPs, using the `buy` command as above. As a starting point, consider FRPs with a few simple Kinds, like:

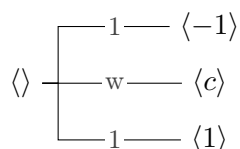
1. For various values of v ,



2. For various values of w ,



3. For various values of $-1 < c < 1$ and of w , starting with $w = 1$,



Try some other examples. Can you guess the relationship between an FRP's Kind and its risk-neutral price? Don't forget the results of our earlier demos. What do you expect to see when you tabulate the values of a large sample of FRPs of the same Kind? What does this mean for the risk-neutral price?

Key Take Aways

How much is an FRP worth?

To begin to answer that question, we consider how well we can predict the value of a scalar FRP. Following a long tradition, we express our prediction through a *price*. A larger predicted payoff corresponds to a higher price and a smaller (even negative) predicted payoff to a lower (even negative) price.

The FRP market lets us purchase *any number* of FRPs of the same Kind at a fixed price per unit. We can borrow and use unlimited funds with no interest but must pay back that loan when our FRPs' values are revealed.

If we can purchase a large number of FRPs of the same Kind at a price $\$c$ that *essentially guarantees* us a profit, we call c an **arbitrage price** for those FRPs. If we have an opportunity to purchase FRPs at an arbitrage price, we would always take it – at scale.

The set of arbitrage prices for an FRP contains every number from $-\infty$ up to **but not including** a value r , which may be a real number or ∞ . This value

r is the **risk-neutral price** for a scalar FRP. It is the smallest value that is bigger than all arbitrage prices for that FRP.

No reasonable person would offer us an arbitrage price to purchase FRPs because it would (essentially) guarantee them a loss. Nor would you accept an offer to pay *more* than the risk-neutral price, for it would (essentially) guarantee you a loss. But at the risk-neutral price, there are no guarantees; you may win or lose, and neither buyer nor seller has the advantage.

The term risk-neutral here means that the price is not sensitive to the risk of loss that you face or the degree of uncertainty in the FRPs value. When you can purchase as many FRPs as you like with interest-free funding, you are not sensitive to risk as we would be in real life. The risk-neutral price reflects a prediction of the value produced by an FRP, it is the best prediction in some sense and can be seen as a “typical value.”

6.1 Fundamental Properties of Risk-Neutral Prices

The risk-neutral price of an FRP X represents a good prediction of X 's value. This statement requires some interpretation. For instance, we saw the FRP with Kind

$$\langle \rangle \begin{cases} 1 & \text{--- } \langle 0 \rangle \\ 1 & \text{--- } \langle 2 \rangle \end{cases}$$

has a risk-neutral price of 1. If this is to be considered a good prediction of the FRP's value, shouldn't it be a concern that *1 is not a possible value of X* ? If you predict 1, you will *always* be wrong. This is true, but a key point is that there are different ways to assess the accuracy of prediction. If we require that we guess the exact value, then it is easy to construct Kinds and “good” predictions that are almost always wrong and almost always far from the true value. For example, with the Kind `uniform(1, 2, ..., 10_000_000_000)`, always guessing 10,000,000,000 does as well as possible in guessing the exact value but is wrong and very far from the true value the vast majority of the time. While it is sometimes sensible to prioritize guessing the exact value, that turns out not to be the most useful criterion in practice.

The market captures a different notion of prediction: prediction accuracy *in the aggregate*. By defining predictions in terms of prices, our predictions are fine-tuned to the structure of the Kind. If you guess below the risk-neutral price, the value will tend to be above your guess (and you can make money almost certainly in the

market). If you guess above the risk-neutral price, the value will tend to be below your guess (and you will lose money in the market if you purchase enough). The risk-neutral price is thus a “typical” value of X ; it gives us a prediction for the value that is *as close possible to the FRP’s value on average*, in a particular sense to be described below.

We can learn quite a lot about X from its risk-neutral price. If X is the *constant* FRP with value v , we know its risk-neutral price is v , for the reasons we saw earlier.⁶

If X and Y are FRPs and we *know* that the value of X will be $\leq$ the value of Y , we write $X \leq Y$. In this case, if c is an arbitrage price for X , it must also be an arbitrage price for Y as our payoff with Y will be at least that with X . Hence, X ’s *risk-neutral price is $\leq Y$ ’s risk-neutral price*. For instance, if all the values of X are between a and b , with $a \leq b$, then X ’s risk-neutral price must be between a and b by applying this fact to X and the constant FRPs at a and b .

For a real-number s , we write sX for the transformed FRP⁷ that *scales the value of X , whatever it is, by s* . Formally, this is $\psi_s(X)$ where ψ_s is the statistic defined by $\psi_s(x) = sx$, which just scales its argument by s . If we know the risk-neutral price for X , can we find the risk-neutral price for sX ? Consider a large batch of FRPs with Kind equivalent to $\text{kind}(X)$: $X_{[1]}, \dots, X_{[m]}$. In the Marketplace, we can choose to purchase the transformed batch $sX_{[1]}, \dots, sX_{[m]}$. If c is any arbitrage price for X , then for large enough m , purchasing the batch $X_{[1]}, \dots, X_{[m]}$ at unit price $\$c$ would essentially guarantee us a profit. If $s > 0$, then we would also get a profit from the same batch with payoffs $sX_{[1]}, \dots, sX_{[m]}$ at unit price $\$sc$ because the payoffs and the profit are all just scaled by s . So, sc is an arbitrage price for sX . (If $s < 0$, we just use the same argument scaling by $-s$, yielding the same result.) Hence, *the risk-neutral price of sX is just s times the risk-neutral price for X* .

We can (and will) go on like this, *deriving properties of the risk-neutral price from the logic of arbitrage prices*. But first it will be nice to have a ... crisper notation.

The $\mathbb{E}$ Operator. If X is a scalar FRP, we will use $\mathbb{E}(X)$ to denote the risk-neutral price of the FRP. (In `frplib`, this is `E(X)`.)

Consider one more property of risk-neutral prices. Let $d = \dim(X)$ and define $Y = \text{proj}_i(X)$ and $Z = \text{proj}_j(X)$ for some $1 \leq i, j \leq d$. If ψ is the statistic of type $d \rightarrow 1$ defined by $\psi(x) = x_i + x_j$, we can write the transform equivalently as the inlined statistic $Y + Z$. This gives us three FRPs derived from X : Y , Z , and $Y + Z$.

We can find $\mathbb{E}(Y + Z)$ from $\mathbb{E}(Y)$ and $\mathbb{E}(Z)$. If $c_1 \leq \mathbb{E}(Y)$ and $c_2 < \mathbb{E}(Z)$ are arbitrage prices for Y and Z , then $c_1 + c_2$ is an arbitrage price for $Y + Z$ because

⁶At any price $c < v$, we make $\$v - c > 0$ for every unit purchased.

⁷This is an inlined statistic. See page 131.

we can make arbitrarily large amounts of money from the Z payoffs even if we lose a little from the Y payoffs. Similarly, if $c_1 < \mathbb{E}(Y)$ and $c_2 \leq \mathbb{E}(Z)$, $c_1 + c_2$ is an arbitrage price for $Y + Z$. But $\mathbb{E}(Y) + \mathbb{E}(Z)$ *cannot* be an arbitrage price for $Y + Z$ because the payoff at this price from any batch of $Y + Z$ clones is equivalent to a payoff from batches of Y 's and Z 's at their risk-neutral price, for which a profit is *not* essentially guaranteed. It follows that $\mathbb{E}(Y) + \mathbb{E}(Z)$ *is* the risk-neutral price for $Y + Z$. As we will see soon, this result generalizes to any dimension.⁸

Together, these arguments give us four key properties of risk-neutral prices.

⁸Bob's equation from Chapter 7, $\text{Sum}(a :: b) = \text{Sum}(\langle \text{Sum}(a), \text{Sum}(b) \rangle)$, does the job.

Box 34. Key Properties of Risk-Neutral Prices. Let X, Y be FRPs.

Constancy. If X is a constant FRP with value v ,

$$\mathbb{E}(X) = v. \quad (6.1)$$

Scaling. For any real number s ,

$$\mathbb{E}(sX) = s \mathbb{E}(X). \quad (6.2)$$

Ordering. If $X \leq Y$,

$$\mathbb{E}(X) \leq \mathbb{E}(Y). \quad (6.3)$$

Additivity. If $X_1, X_2, \dots, X_n$ are FRPs of common dimension,

$$\mathbb{E}(X_1 + X_2 + \dots + X_n) = \mathbb{E}(X_1) + \mathbb{E}(X_2) + \dots + \mathbb{E}(X_n). \quad (6.4)$$

In the condition for Scaling, $sX = \zeta(X)$ for statistic $\zeta(x) = sx$. In the condition for Ordering, $X \leq Y$ means that the value of X is *known as a fact* to be $\leq$ the value of Y . In the condition for Additivity, for *any* FRPs $X_1, \dots, X_n$ of common dimension, we can always build an FRP X and find statistics ψ and φ_i , $i \in [1..n]$ for which $X_1 + \dots + X_n = \psi(X)$ and $X_i = \varphi_i$, for $i \in [1..n]$.

These properties provide critical tools for finding and working with risk-neutral prices. Even with the formula for risk-neutral prices that we derive in the next Section, it will often more direct to solve or simplify a problem with the logic of these properties. In practice, we often use the Constancy, Scaling, and Additivity properties together to establish that

$$\mathbb{E}(\alpha_0 + \alpha_1 X_1 + \alpha_2 X_2 + \dots + \alpha_n X_n) = \alpha_0 + \alpha_1 \mathbb{E}(X_1) + \alpha_2 \mathbb{E}(X_2) + \dots + \alpha_n \mathbb{E}(X_n), \quad (6.5)$$

for constants $\alpha_0, \alpha_1, \dots, \alpha_n$. This derived property that is called **linearity**.

We derived arbitrage prices – and thus the risk-neutral price – for a scalar FRP

by considering an arbitrarily large collection of FRPs with the same Kind. It follows that the risk-neutral price of an FRP is *determined by its **Kind*** not by the FRP's particular value.

The Risk-Neutral Price is a Property of the Kind.

All FRPs with the same Kind have the same risk-neutral price. The risk-neutral price of an FRP is thus **a property of the Kind** and does not depend on any particular FRPs produced value.

Up to now in this section, we have been considering the risk-neutral prices of scalar FRPs. How do we handle FRPs of dimension > 1 ? Because the value of such an FRP is a tuple, there is not a single payoff, so it is unclear how to assign a single risk-neutral price to it. Instead, we extend the risk-neutral price to a more general idea – the **expectation**. If X is a scalar FRP, its expectation $\mathbb{E}(X)$ is its risk-neutral price. If $\dim(X) > 1$, its expectation $\mathbb{E}(X)$ is the tuple whose components are the risk-neutral prices of X 's scalar component FRPs.

Definition 35. Let X be an FRP. We use $\mathbb{E}(X)$ to denote the **expectation of X** .

- If $\dim(X) = 1$, $\mathbb{E}(X)$ is just the risk-neutral price of X .
- If $\dim(X) = n > 1$ and FRPs $\langle X_1, X_2, \dots, X_n \rangle$ are its scalar components, then $\mathbb{E}(X)$ is an n -tuple of numbers defined by

$$\mathbb{E}(X) = \langle \mathbb{E}(X_1), \mathbb{E}(X_2), \dots, \mathbb{E}(X_n) \rangle, \quad (6.6)$$

that is, the tuple containing the risk-neutral prices of the components.

Note that the expectation of an FRP has the *same dimension* as the FRP.

We often write expectations in terms of FRPs, but it is fine to refer to the expectation of a Kind because $\mathbb{E}(X)$ is a property of $\text{kind}(X)$ and does not depend on X 's specific value.

The good news is that because the expectation is derived from risk-neutral prices, **all the properties 6.1, 6.2, 6.3, and 6.4 continue to hold for any FRP**. We can view equations (6.2), (6.3), and (6.4) as telling us that $\mathbb{E}$ maps the operations of scaling, ordering, and adding on FRPs/Kinds to the analogous operations on values. These operations all work for numbers. For numbers x and y , we can scale them sx , order them $x \leq y$, and add them $x + y$. And the operations also work for *tuples of numbers* of a common dimension n ;⁹ we can scale, order, and add them:

⁹See Section 17.3 in Interlude F, which discusses “vector” operations on tuples. This motivates the `VecTuples` used for values in `frplib`.

$$s\langle x_1, x_2, \dots, x_n \rangle = \langle sx_1, sx_2, \dots, sx_n \rangle \quad (\text{Scaling})$$

$$\langle x_1, x_2, \dots, x_n \rangle \leq \langle y_1, y_2, \dots, y_n \rangle \text{ if and only if } x_1 \leq y_1 \wedge \dots \wedge x_n \leq y_n \quad (\text{Ordering})$$

$$\langle x_1, x_2, \dots, x_n \rangle + \langle y_1, y_2, \dots, y_n \rangle = \langle x_1 + y_1, x_2 + y_2, \dots, x_n + y_n \rangle. \quad (\text{Additivity})$$

For instance, Additivity (6.4) and equation (6.6) imply that for *any* FRP X

$$\mathbb{E}(\text{Sum}(X)) = \text{Sum}(\mathbb{E}(X)) \quad (6.7)$$

for the **Sum** statistic that adds up the components of its argument.

A frequently occurring special case of equation (6.6) is when X is an *independent join* of $X_1, X_2, \dots, X_n$ scalar FRPs:

$$\mathbb{E}(X_1 \star X_2 \star \dots \star X_n) = \langle \mathbb{E}(X_1), \mathbb{E}(X_2), \dots, \mathbb{E}(X_n) \rangle. \quad (6.8)$$

Independence tells us that we can price the components separately.

Another important point to keep in mind is that the expectation gives us a prediction based on what we know about an FRP's value, namely its Kind. When we get information about an FRP's value that effectively produces a new (but related) FRP with a new (but related) Kind. Specifically, when we constrain an FRP X with an observation, we get another FRP $X \mid c$ with Kind $\text{kind}(X \mid c) = \text{kind}(X) \mid c$. Its expectation updated for the observation is $\mathbb{E}(X \mid c)$.

Even when we write an expectation without the given bar it is still there implicitly because for any FRP X , $X = X \mid \top$, where $\top$ is the condition that always returns true. (We think of this as representing “common knowledge.”) Hence, $\mathbb{E}(X)$ is exactly the same as $\mathbb{E}(X \mid \top)$.

Puzzle 76. Given that we observe an FRP X to have value 4, what is its expectation (i.e., risk-neutral price)? We would write this with the observation as $\mathbb{E}(X \mid \{X = 4\})$, or $\mathbb{E}(X \mid X = 4)$ without the Iverson braces if we like.

The following examples illustrate how we can use the properties above to find expectations.

Example 6.1. If X has Kind of the form

$$\langle \rangle \begin{cases} a & \langle -1 \rangle \\ b & \langle 1 \rangle \end{cases}$$

what is $\mathbb{E}(X^2)$? (Recall that X^2 is the “inline” notation for the transformed FRP $\psi(X)$ with the simple statistic $\psi(x) = x^2$; see the discussion of inlined statistics on page 131. It is *not* the same as $X \star X = X \star \star 2$.)

When X produces a value v , that value is fed to the input port of X^2 through an adapter that outputs v^2 . Here, v can be either -1 or 1, and in both cases, $v^2 = 1$. This means that 1 is the only possible value of X^2 . It is constant.

Hence, by the Constancy property (6.1), $\mathbb{E}(X^2) = 1$.

Example 6.2 Changing Units

The FRP D represents a measured distance in kilometers, and we’d like to create an FRP X that represents the same distance in *miles*. This transformation is straightforward with the statistic $\psi(d) = cd$ for a constant $c \approx 0.621371$. So, $X = \psi(D)$, though for such a simple transformation we most often *inline* the statistic, writing it as $X = cD$.

The Scaling property (6.2) tells us that $\mathbb{E}(X) = \mathbb{E}(cD) = c\mathbb{E}(D)$. The expectation scales by the same factor.

Example 6.3. Let A be an FRP representing a random angle, measured in degrees. Even without knowing $\text{kind}(A)$, we know that its values must lie in the interval $[0\_360)$. If A' is an FRP derived from A by converting its values to radians, then its values must lie in the interval $[0\_2\pi)$. Let $X = \cos(A')$ and $Y = \sin(A')$ represent the cosine and sine of this angle; their values each lie in the interval $[-1\_1]$.

The Ordering property (6.3) of expectations tells us that

$$\begin{aligned} 0 &\leq \mathbb{E}(A) \leq 360 \\ 0 &\leq \mathbb{E}(A') \leq 2\pi \\ -1 &\leq \mathbb{E}(X) \leq 1 \\ -1 &\leq \mathbb{E}(Y) \leq 1. \end{aligned}$$

Example 6.4 Measuring Uncertainty

The risk-neutral price of a scalar FRP is a prediction of the FRP’s value. The greater the uncertainty in the Kind of that FRP, the harder it is to accurately predict the FRP’s value. However, even when the uncertainty is high, our

First of the “Measuring Uncertainty” example series.

prediction might by chance be close to the actual value, a lucky guess so to speak. So how do we quantify the uncertainty in a Kind of FRPs?

The answer is to construct a transformed FRP with a customized statistic that describes how the value deviates from our prediction. Here, we will introduce three ways to quantify uncertainty that are called the *Mean Absolute Deviation*, the *Variance*, and the *Entropy*. We will focus here only on 1-dimensional FRPs.

Let Y be a scalar FRP with $\mu = \mathbb{E}(Y)$ its risk-neutral price. We denote the **kernel** of $\text{kind}(Y)$ by k_Y . If v is a possible value of Y , $k_Y(v)$ gives the canonical weight on v in $\text{kind}(Y)$ or 0 if it is not a possible value.

First, define a family of statistics φ_c , indexed by real numbers c , where

$$\varphi_c(x) = |x - c|. \quad (6.9)$$

This statistic measures how far its argument value is from the number c . We define the *Mean Absolute Deviation* of Y , denoted $\text{MAD}(Y)$, to be

$$\text{MAD}(Y) = \mathbb{E}(\varphi_\mu(Y)). \quad (6.10)$$

This is *our prediction of how far Y 's value is from our prediction of its value*. That is, by the nature of the statistic φ_μ , the expectation of the FRP $\varphi_\mu(Y)$ measures the deviation between Y 's value and Y 's risk-neutral price.

Because $\varphi_c(x) \geq 0$ for all x , the FRP $\varphi_\mu(Y)$ can never be negative, so by the Ordering property (6.3), $\text{MAD}(Y) \geq 0$. The extreme case is when Y is a constant FRP. In this case, $\mathbb{E}(Y)$ equals the only possible value, so $\text{MAD}(Y) = 0$.

One novel feature of this is that we use a statistic that is customized to match the FRP we are transforming. While we could look at $\mathbb{E}(\varphi_c(Y))$ for any c , we use $c = \mu = \mathbb{E}(Y)$. Because φ_μ has a relatively simple form, we would tend to inline this statistic and write $\text{MAD}(Y) = \mathbb{E}(|Y - \mathbb{E}(Y)|)$.

The Mean Absolute Deviation is intuitive: it predicts how far the actual value of an FRP will be from its predicted value (risk-neutral price). The distance between value and prediction is in the same units as Y , which is nice. Unfortunately, the absolute value $|\cdot|$ in the statistic makes it harder to work with mathematically. For that reason, we introduce the *Variance*.

Define a family of statistics ψ_c , with one statistic for each real number c , where

$$\psi_c(x) = (x - c)^2. \quad (6.11)$$

Like φ_c above, this statistic measures how far its argument value is from the number c , but it measures distance in *squared* units. We define the *Variance* of Y , denoted $\text{Var}(Y)$, to be the expectation of $\psi_\mu(Y)$:

$$\text{Var}(Y) = \mathbb{E}(\psi_\mu(Y)) = \mathbb{E}((Y - \mathbb{E}(Y))^2), \quad (6.12)$$

where the second form is the inlined expression of the statistic. This is *our prediction of how far Y 's value is from our prediction of its value in squared units*. If the variance is high, we predict that Y 's value will typically be far from its expectation; if the variance is small, we predict that Y 's value will typically be close to its expectation.

The quadratic versus the absolute value is the distinction between ψ_c and φ_c and between **Var** and **MAD**. The move to squared units makes the variance's numeric values a bit harder to understand, but the quadratic makes it easier to simplify and manipulate, as we will see.

Again, because $(x-c)^2 \geq 0$ for all x , $\psi_c(Y)$ cannot be negative, so $\text{Var}(Y) \geq 0$ by the Ordering property. The extreme case is when Y is a constant FRP. In this case, $\mathbb{E}(Y)$ equals the only possible value and $\text{Var}(Y) = 0$. A constant FRP embodies no uncertainty.

Finally, define a family of statistics ζ_k , with one statistic for each canonical Kind k , where

$$\zeta_k(x) = -\lg k(x), \quad (6.13)$$

where $\lg$ denotes the logarithm base 2 and $k(x)$ is the **kernel** of the Kind k , giving the canonical weight associated with value x in Kind k . Unlike the previous two statistics that measure distances between values, this statistic measures the weights on each value. The statistic is *larger* for values that have *lower* weight and smaller for values that have higher weight. Remember that a canonical weight is ≤ 1 , so the log of that weight is negative and the negative sign makes the result non-negative. In other words, $\zeta_k(x) \geq 0$ for any value x in a branch of Kind k .

We define the *Entropy* of Y , denoted $\mathbb{H}(Y)$, to be

$$\mathbb{H}(Y) = \mathbb{E}(\zeta_{\text{kind}(Y)}(Y)) = \mathbb{E}(-\lg k_Y(Y)), \quad (6.14)$$

where the second form is the inlined expression of the statistic and k_Y is the **kernel** of $\text{kind}(Y)$. (Note that the subscript in k_Y is just a *label*.)

Because $\zeta_k(Y)$ cannot be negative and can be at most $-\lg p_{\min} = \lg(1/p_{\min})$ where $p_{\min}$ is the smallest weight in $\text{kind}(Y)$, the Ordering property (6.3) implies that

$$0 \leq \mathbb{H}(Y) \leq \lg(1/p_{\min}).$$

If Y is a constant FRP, then $p_{\min} = 1$, which implies that $\mathbb{H}(Y) = 0$. As with the other measures, a constant – with no uncertainty – gives the minimal value 0.

The entropy depends only on the number of values and their weights, not on the values themselves. Its value is measured in *bits*. If Y has $\text{Kind}(\text{uniform}(1, 2, \dots, n))$ for some positive integer n , then all the canonical weights in $\text{kind}(Y)$ equal $1/n$, so $\lg \mathbf{k}_Y(Y)$ is a constant FRP. In this case, $\mathbb{H}(Y) = \lg n$, which is within 1 of the number of bits used to represent the number n in binary (base 2). We will develop the interpretation of $\mathbb{H}(Y)$ later in this example series and even further in Part V.

The previous example introduced several useful quantities that measure the uncertainty in a Kind. Of these, two are particularly important.

If X is a scalar FRP,

- The **variance** of X , denoted by $\text{Var}(X)$, is defined by

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}(X))^2]. \quad (6.15)$$

This is the expectation of the transform $\psi(X)$ where $\psi(x) = (x - \mathbb{E}(X))^2$.

- The **entropy** of X , denoted by $\mathbb{H}(X)$, is defined by

$$\mathbb{H}(X) = \mathbb{E}(-\lg \mathbf{k}_X(X)), \quad (6.16)$$

where $\lg$ is the base-2 logarithm and $\mathbf{k}_X$ is the kernel of $\text{kind}(X)$. This is the expectation of the transform $\varphi(X)$ where $\varphi(x) = -\lg(\mathbf{k}_X(x))$.

Both of these measure the uncertainty – difficulty of prediction – in $\text{kind}(X)$. We will explore properties of both extensively as we proceed.

Example 6.5 Sums

In Chapter 7, Alice and Bob figure out a clever way to compute the Kind for the total of 100 die rolls, `Sum(d6 ** 100)` in `frplib`, where `d6` is the Kind for a roll of a balanced six-sided die. Fortunately, to compute its risk-neutral price, we do not need such clever tricks due to the Additivity property in equation (6.7).

In `frplib`, we can create the Kind `d6` and an FRP `D_6` that represents the roll of a balanced six-sided die.

```
pgd> d6 = uniform(1, 2, ..., 6)
pgd> D_6 = frp(d6)
pgd> D_6
An FRP with value <4>
```

The Kinds for multiple rolls of the dice are computed with an independent join power, but their sizes quickly grow large. For instance, `d6 ** 4`, `d6 ** 8`, `d6 ** 100` have respected sizes 1296, 1679616, and 653318623500070906096690267158057820537143710472954871543071966369497141477376.

We can directly create FRPs, however, to simulate large number of dice rolls without any problem:

```
pgd> Rolls4 = D_6 ** 4
pgd> Rolls8 = D_6 ** 8
pgd> Rolls100 = D_6 ** 100
pgd> Rolls4
An FRP with value <2, 1, 3, 2>
(It may be slow to evaluate its kind.)
pgd> Rolls8
An FRP with value <5, 6, 3, 4, 4, 1, 4, 3>.
(It may be slow to evaluate its kind.)
pgd> Rolls100
An FRP with value <2, 1, 3, 3, 4, 1, 1, 6, 3, 5, 5, 1, 2, 1, 3,
4, 6, 2, 2, 5, 2, 1, 4, 6, 5, 6, 3, 6, 3, 2, 6, 2, 1, 1, 5, 1,
1, 6, 3, 6, 3, 3, 3, 4, 5, 4, 3, 5, 6, 5, 5, 2, 2, 6, 2, 3, 3,
6, 3, 1, 2, 4, 4, 4, 4, 3, 1, 4, 1, 3, 5, 1, 3, 2, 3, 1, 4, 4,
2, 1, 2, 6, 3, 4, 3, 1, 5, 1, 1, 2, 5, 3, 5, 2, 4, 2, 3, 6, 3,
1>. (It may be slow to evaluate its kind.)
```

The value of `Rolls4` shows us the four rolls. This is an independent join, and recall

that it is equivalent to `clone(D_6) * clone(D_6) * clone(D_6) * clone(D_6)`, as in equation (4.9), and similarly for `Rolls8` and `Rolls100`. The Kind of `Rolls4` has not yet been computed, as the message indicates, though we could force it to be by calling `kindn(Rolls4)`. (Try it!) Deferring the computation of the Kind when the size may be large allows us to construct FRPs even if the Kind is slow to compute.

(Note that computing a repeated independent join of an FRP without the `clone` is generally not what we want.

```
pgd> D_6 * D_6 * D_6 * D_6    # not quite right, all values the same
An FRP with value <4, 4, 4, 4>.
```

`D_6` has only one value fixed for all time, so however many terms in the join, they will all have the same value. Recall that independent join *powers* helpfully do the clones for us.)

If we try to compute the risk-neutral price directly with the `E` operator, `frplib` detects that the Kind is hard to compute and approximates the answer instead.

```
pgd> Sum(Rolls100)
An FRP with value <322>. (It may be slow to evaluate its kind.)
pgd> E(Sum(Rolls100))
+- Computing approximation (tolerance 0.01) --+
| 350.0251                                     |
+-----|
```

We could force the Kind to be computed with `E(Sum(Rolls100), force_kind=True)` or change the approximation tolerance with `E(Sum(Rolls100), tolerance=0.001)` if we prefer, though the former calculation would not complete given the size of the Kind.

Additivity gives us a quick and *exact* answer for `E(Sum(Rolls100))`. First, remember that `E(D_6) == E(clone(D_6))` because an FRP's expectation only depends on its Kind not its particular value. And we know that

```
pgd> E(D_6)
7/2
```

Second, Definition 35 tells us that `E(D_6 ** 100)` is equal to

```
<E(clone(D_6)), E(clone(D_6)), ..., E(clone(D_6))>
```

Indeed, we can compute it directly in `frplib`:

```
pgd> E(D_6 ** 100)
<7/2, 7/2, 7/2, ..., 7/2, 7/2>
```

where 95 of the values have been elided for brevity. Finally, the Additivity property (equation 6.7) implies that $E(\text{Sum}(D_6 ** 100)) = \text{Sum}(E(D_6 ** 100))$ and

```
pgd> Sum(E(D_6 ** 100))
350
```

which is the exact answer we want. More generally, the Additivity property tells us that for any $n \in [1...)$, $E(D_6 ** n) = n E(D_6) = \frac{7}{2}n$.

Indeed, combining equations (6.8) and (6.7), we have that for any FRP X

$$E(\text{Sum}(X ** n)) = n E(X). \quad (6.17)$$

Following up on the previous example, consider the statistic `Mean` that computes the arithmetic average of its input's components. For any value v , we have that $\text{Mean}(v) = \text{Sum}(v)/\text{Dim}(v)$. Hence, combining equation (6.17) and the Scaling property (6.2) yields

$$E(\text{Mean}(X ** n)) = E(X). \quad (6.18)$$

The average of an independent join power has the same expectation as each term.

It is fairly common when taking measurements to use averages to compute a more “representative” quantity. Surveyors average multiple measurements to estimate distances. Baseball coaches use batting averages to assess hitters. Teachers take averages of students' exams to assign grades. In light of equation 6.18, we might ask how averaging helps if the expectation – our prediction – does not change by averaging. The next example sheds light on this question.

Example 6.6 Averaging and Uncertainty

Our uncertainty about a system increases with the difficulty of making accurate predictions. Example 6.4 introduced several different ways to *quantify the uncertainty* in a Kind. We consider multiple quantities for this purpose because in some context, one might emphasize subtly different features or have a useful

Part of the “Measuring Uncertainty” series.

practical interpretation or be more convenient to work with.

Here, we consider the *variance* of a Kind and in particular the variance of an independent join power. As defined earlier: if Y is a scalar FRP with risk-neutral price $\mathbb{E}(Y)$, its variance is $\text{Var}(Y) = \mathbb{E}(\psi_{\mathbb{E}(Y)}(Y))$ where $\psi_c(y) = (y - c)^2$ is a family of statistics, one for each constant c . As this statistic is a simple quadratic, we “inline” it as $\text{Var}(Y) = \mathbb{E}[(Y - \mathbb{E}(Y))^2]$. The variance is our best prediction of the squared distance of Y ’s value from its expectation. When the variance is small, Y ’s value tends to be close to its expectation. When it is large, Y ’s value tends to be far from its expectation. We saw in Example 6.4 that $\text{Var}(Y) \geq 0$ is always true.

See page 131 on inlining.

To get a different perspective on the variance, we write the statistic ψ_c by expanding the quadratic: $\psi_c(y) = y^2 - 2cy + c^2$. In this form, we can apply the Constancy, Scaling, Additivity properties to get the **variance shortcut**:

$$\begin{aligned}
 \text{Var}(Y) &= \mathbb{E}(\psi_c(Y)) \\
 &= \mathbb{E}(Y^2 - 2\mathbb{E}(Y)Y + (\mathbb{E}(Y))^2) \\
 &= \mathbb{E}(Y^2) + \mathbb{E}(-2\mathbb{E}(Y)Y) + \mathbb{E}((\mathbb{E}(Y))^2) && \text{(by Additivity)} \\
 &= \mathbb{E}(Y^2) + \mathbb{E}(-2\mathbb{E}(Y)Y) + (\mathbb{E}(Y))^2 && \text{(by Constancy)} \\
 &= \mathbb{E}(Y^2) - 2\mathbb{E}(Y)\mathbb{E}(Y) + (\mathbb{E}(Y))^2 && \text{(by Scaling)} \\
 &= \mathbb{E}(Y^2) - (\mathbb{E}(Y))^2. && (6.19)
 \end{aligned}$$

Remember that the expectation $\mathbb{E}(Y)$ is just a number, so when we apply the Scaling property to $\mathbb{E}(-2\mathbb{E}(Y)Y)$, we pull out the constant $-2\mathbb{E}(Y)$. We called this combined application of Constancy, Scaling, and Additivity *linearity* in equation (6.5). The variance shortcut reveals the variance as the difference between squaring after and squaring before computing the risk-neutral price. There is no difference if Y is a constant FRP, but as the possible values of Y become more widely spread, squaring will spread them further, increasing the expectation of Y^2 relative to the square $(\mathbb{E}(Y))^2$ of the original price.

With these ideas in hand, we will investigate the question of why we average measurements. In this setting, $Y = \text{Mean}(X \star n)$ for a scalar FRP X and a positive integer n . X and its clones represent individual measurements of some quantity (e.g., one surveyor’s distance, one batter’s at bat, one exam), and Y is the average of n such measurements. We saw in (6.18) that $\mathbb{E}(Y) = \mathbb{E}(X)$. The expectation of the FRP representing the average is the same as for the FRP

representing a single measurement. What can we say about the uncertainty in $\text{kind}(Y)$ versus $\text{kind}(X)$?

In this example, we will empirically investigate this question in the playground, using the built-in `Var` operator. We will start with a couple arbitrary Kinds; you can do these steps with other Kinds as well.

There is also a *statistic* `Variance` that is different; it computes the “sample variance” of its input value’s components.

```
pgd> k1 = geometric(1, 2, 3, 4, 5)
pgd> k2 = Mean(k1 ** 2)
pgd> k4 = Mean(k1 ** 4)
pgd> k6 = Mean(k1 ** 6)
pgd> k8 = Mean(k1 ** 8)    # this might take a few moments
pgd> Var(k1)
0.5837669094693028
pgd> Var(k2) / Var(k1)
0.5
pgd> Var(k4) / Var(k1)
0.25
pgd> Var(k6) / Var(k1)
0.166666666666666666666666666665
pgd> Var(k8) / Var(k1)
0.125
```

When we average 2, 4, 6, and 8 independent copies, the variance decreases by factors of $1/2$, $1/4$, $1/6$, and $1/8$. Let’s try it with a different Kind to start

```
pgd> k1 = choice(0, 1)
pgd> k3 = Mean(k1 ** 3)
pgd> k5 = Mean(k1 ** 5)
pgd> k10 = Mean(k1 ** 10)
pgd> k16 = Mean(k1 ** 16)    # this might take a moment
pgd> Var(k1)
1/4
pgd> Var(k3) / Var(k1)
0.333333333333333333333333333333
pgd> Var(k5) / Var(k1)
0.2
pgd> Var(k10) / Var(k1)
```

```
0.1
pgd> Var(k16) / Var(k1)
0.0625
```

This shows the same pattern! So, we hypothesize that

$$\text{Var}(\text{Mean}(X \star n)) = \frac{1}{n} \text{Var}(X). \quad (*)$$

Puzzle 77. The `frplib` function `fast_join_pow` can efficiently compute $\psi(k \star n)$ for a Kind k when ψ is a special type of statistic. (See the first dialogue in Chapter 7.)

`Sum` is a statistic of that special type, and because `Mean` is just a scaled version of `Sum`, we can use `fast_join_pow(Sum, k, n)` to help find $\text{Mean}(k \star n)$ as well.

Use this to check our hypothesis (*) for larger values of n , like 100, 500, or 1000 with at least the Kind `choice(0,1)`.

It turns out that our hypothesis is true. Although our highly suggestive evidence here does not firmly establish this claim, we will see an argument later that proves it. For now, consider how that fact addresses our question – why do we average? When we compute $\mathbb{E}(\text{Mean}(X \star n))$ we want to predict the same value we are measuring in any single copy of X , so it makes sense that $\mathbb{E}(\text{Mean}(X \star n)) = \mathbb{E}(X)$. The average measurement should be measuring the same underlying quantity.

$\text{Var}(\text{Mean}(X \star n)) = \frac{1}{n} \text{Var}(X)$ tells us that *averaging reduces the uncertainty in our prediction* of that quantity by a factor proportional to the number of measurements, when the repeated measurements are all independent. Lower uncertainty means that our predictions tend to get more accurate. This is why we average.

Example 6.7 Aces

Let S be the FRP of dimension 52 (and size 52!) whose values are all 52! permutations of $1, 2, \dots, 52$, where the numbers represent cards in a standard deck in a fixed order by rank and suit. Let $\mathcal{A} = \{1, 14, 27, 40\} \subset [1 \dots 52]$ represent the four aces in the deck.

We can use the FRP S as a model of equally-weighted shuffles of a standard

deck of cards. We will use this model to predict how many cards are between each ace in the deck.

Define a statistic ψ that takes a value of S and returns $\langle a_1, a_2, a_3, a_4, a_5 \rangle$, where a_1 is number of components of S 's value *before* the first ace in the deck (a value in $\mathcal{A}$); a_2 is the number of components strictly between the first two aces in the deck (values in $\mathcal{A}$); a_3 is the number of components strictly between the second and third aces in the deck (values in $\mathcal{A}$); a_4 is the number of components strictly between the third and fourth aces in the deck (values in $\mathcal{A}$); and a_5 is the number of components strictly *after* the final ace in the deck (value in $\mathcal{A}$).

Puzzle 78. Show how to define the statistic ψ in `frplib`. Call it `between_aces`.

We can use the `shuffle` FRP factory to create S :

```
pgd> S = shuffle(irange(52))    # Values are permutations of 1..52
pgd> S
```

```
An FRP with value <15, 48, 16, 41, 20, 22, 29, 4, 17, 34, 39,
14, 26, 8, 49, 1, 18, 2, 31, 32, 52, 37, 36, 42, 19, 3, 35, 28,
50, 25, 38, 33, 45, 24, 44, 46, 11, 43, 7, 23, 9, 13, 12, 40,
30, 47, 10, 21, 27, 6, 51, 5>.
```

We can apply the statistic you defined in the puzzle to look at the gaps between aces in the simulated deck.

```
pgd> A = between_aces(S)
pgd> A
An FRP with value <11, 3, 27, 4, 3>. (It may be slow to evaluate its kind.)
```

The FRP A has dimension 5, and we can write its components as A_1, A_2, A_3, A_4, A_5 whose values have the meaning described earlier. We can look at the components individually, for instance

```
pgd> A[1]
An FRP with value <11>. (It may be slow to evaluate its kind.)
pgd> A[3]
An FRP with value <27>. (It may be slow to evaluate its kind.)
```

and similarly for the other aces.

To start, we would like to compute $\mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5)$. The FRP here is transform of A , which is in turn a transform of S . The result is a transform of S by statistic $\zeta = \text{Sum} \circ \text{between\_aces}$ (“Sum after between_aces”), which can be expressed in multiple equivalent forms in `frplib`:

```
pgd> Sum(A)
pgd> Sum(between_aces(S))
pgd> S ~ Sum(between_aces)
pgd> S ~ between_aces ~ Sum
```

These are all the same FRP. (Make sure you understand why.)

Puzzle 79. In `frplib`, construct the statistic `Sum(between_aces)`. This is the *composition* of the two statistics: to apply it, we first apply `between_aces` to a deck and then apply `Sum` to the value it returns. “Sum after between_aces.”

Evaluate `Sum(between_aces(S))`. If you clone S and do this again, what possible values might you see?

Try:

```
pgd> psi = Sum(between_aces)
pgd> FRP.sample(100, psi(S))
```

What do the results tell you? Do the result change if you demo 1000 samples? Can you explain these tables?

The results of the previous puzzle strongly suggest that the statistic ζ is an *invariant*: it gives the same value for every possible shuffle of the deck. We can confirm that empirical result with logic: for every shuffle s , $\zeta(s)$ counts all the cards in the deck *except the aces*. So $\zeta(s) = 48$, and thus $\zeta(S) = A_1 + A_2 + A_3 + A_4 + A_5 = 48$ is a constant FRP. By the Constancy property (6.1), we therefore have that $\mathbb{E}(\zeta(S)) = 48$.

The Additivity property (6.4) now applies:

$$\begin{aligned} 48 &= \mathbb{E}(\zeta(S)) \\ &= \mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5) \\ &= \mathbb{E}(A_1) + \mathbb{E}(A_2) + \mathbb{E}(A_3) + \mathbb{E}(A_4) + \mathbb{E}(A_5), \end{aligned}$$

which constrains the expectations of the counts for the individual aces. With

a bit more logic, we can go even further. We can show that A_1 , A_2 , A_3 , A_4 , and A_5 have the *same Kind*, and thus the same expectation. By the Additivity property (6.4), it follows that

$$\begin{aligned}\mathbb{E}(A_1 + A_2 + A_3 + A_4 + A_5) &= \mathbb{E}(A_1) + \mathbb{E}(A_2) + \mathbb{E}(A_3) + \mathbb{E}(A_4) + \mathbb{E}(A_5) \\ &= 5\mathbb{E}(A_1)\end{aligned}$$

so,

$$\mathbb{E}(A_1) = \mathbb{E}(A_2) = \mathbb{E}(A_3) = \mathbb{E}(A_4) = \mathbb{E}(A_5) = 9.6$$

And we have found the expectations of the A_i 's.

We will establish that the A_i 's have the same Kind by a direct argument and then illustrate it in `frplib` with a smaller “deck” where we can see the trees.

First, consider A_1 and A_4 , though our argument will apply to any $i \neq j$ with $i, j \in [1..5]$. Let $\mathcal{S}$ be the set of all permutations $\langle 1, 2, \dots, 52 \rangle$, i.e., possible values of the FRP. Define a function `cut14` that maps elements of $\mathcal{S}$ to elements of $\mathcal{S}$ (denoted `cut14:  $\mathcal{S} \rightarrow \mathcal{S}$`) that exchanges the positions of cards strictly before the first ace in s and the cards strictly between the third and fourth aces without changing the order of the cards within each group. Notice that if we apply `cut14` to a shuffle s and then apply `cut14` again, we get the original shuffle back! That is, `cut14  $\circ$  cut14 = id`, the identity function on $\mathcal{S}$. The function `cut14` maps every shuffle in $\mathcal{S}$ to a distinct shuffle, and every shuffle in $\mathcal{S}$ is the return value of `cut14` for some shuffle in $\mathcal{S}$. This is what we call a **bijection**; see Section 14.1 in Interlude F.

See Chapter 13 in Interlude F.

For any possible value m of A_1 , we can in principle list all the shuffles among the $52!$ for which the event $\{A_1 = m\}$ occurs. If we apply `cut14` to these shuffles, we get shuffles for which the event $\{A_4 = m\}$ occurs. In fact, we get all such shuffles because if s is a shuffle for which $\{A_4 = m\}$ occurs then `cut14( $s$ )` is a shuffle for which $\{A_1 = m\}$ occurs. It follows the Kinds of A_1 and A_4 have the same weight on m , and since m was an arbitrary value, they have the same weight on all values, and hence the same Kind.

We can apply exactly the same argument with the function `cutij` for $i, j \in [1..5]$ with $i < j$. Hence, all the A_i 's have the same Kind. Moreover, using `cutij`

as a statistic swaps the values of A_i and A_j :

$$\begin{aligned}\text{proj}_{ij}(\text{between\_aces}(S)) &= \langle A_i, A_j \rangle \\ \text{proj}_{ij}(\text{between\_aces}(\text{cut}_{ij}(S))) &= \langle A_j, A_i \rangle.\end{aligned}$$

To illustrate this more concretely, we will study shuffles of the smaller “deck” with cards $1, 2, \dots, 5$ and a statistic analogous to `between_aces`, which we will call `two_four_gaps`, that uses $\{2, 4\}$ instead of the set $\mathcal{A}$.

```
pgd> T = shuffle(irange(5))
pgd> T
An FRP with value <2, 3, 5, 4, 1>. (It may be slow to evaluate its kind.)
```

With the statistic `two_four_gaps` defined in `frplib.examples.aces`, we have

```
pgd> G = two_four_gaps(T)
An FRP with values <0,2,1>
pgd> kind(G)
,---- 0.10000 ---- <0, 0, 3>
|---- 0.10000 ---- <0, 1, 2>
|---- 0.10000 ---- <0, 2, 1>
|---- 0.10000 ---- <0, 3, 0>
|---- 0.10000 ---- <1, 0, 2>
<> -|
|---- 0.10000 ---- <1, 1, 1>
|---- 0.10000 ---- <1, 2, 0>
|---- 0.10000 ---- <2, 0, 1>
|---- 0.10000 ---- <2, 1, 0>
`---- 0.10000 ---- <3, 0, 0>
pgd> kind(Sum(G))
<> ----- 1 ---- 3
pgd> kind(G[1])
,---- 0.4 ---- 0
|---- 0.3 ---- 1
<> -|
|---- 0.2 ---- 2
`---- 0.1 ---- 3
```

```

pgd> E(G[1])
1
pgd> Kind.equal(kind(G[1]), kind(G[2]))
True
pgd> Kind.equal(kind(G[1]), kind(G[3]))
True

```

So we see that G , the analogue of A , always has sum 3 and that the Kinds of its components G_1, G_2, G_3 are the same.

Let's check our argument with cut_{ij} .

```
pgd> kind(T)
```

I'm not showing the `kind(T)` tree here, but you should look at it. It's still small enough to understand and examine. Compare the Kind tree

```
pgd> kind(T) ~ cut(1,2, deck_size=5, aces={2, 4})
```

with `kind(T)`, where `cut` is a statistic factory defined in `frplib.examples.aces`.

Observe that for each branch of `kind(T)` there is exactly one branch of `kind(T) ~ cut(1,2)` that swaps the first and second segment, and vice versa. If you examine these trees side by side and draw a line from each branch in `kind(T)` to the corresponding branch in `kind(T) ~ cut(1,2, deck_size=5, aces={2, 4})`, every branch in both trees will be accounted for. This is the analogue of the bijection we constructed in our earlier argument.

To try with these in the playground, do

```
pgd> from frplib.examples.aces import *
```

You might also find it interesting to look at the code to see how these statistics are defined.

This example shows us the power of the properties we have discovered about expectations. They let us compute the expectations without explicitly considering every possible value. This also shows that expectations are *often easier to compute than the full Kinds*.

Example 6.8. Let X be an FRP and let ψ, ϕ be two compatible statistics. Consider the conditional Kind that maps a value a to the Kind of $\phi(X) \mid \psi(X) = a$. In other words, we *observe* one transformed value of X and we want to use

that information to *predict* a different transformed value of X . We will see how to compute the expectation of this Kind and in the process will discover a useful general property of expectations in the case where the two statistics are related.

As a concrete example, consider

```
pgd> Z = frp(choice(0,1))
pgd> X = Z >> conditional_kind({
...>           0: uniform(1, 2, 3) * choice(4, 5),
...>           1: choice(7, 9, 7) * choice(4, 5)
...>         })
pgd> X
An FRP of dimension 3 and size 10 with value <0, 3, 5>
pgd> kind(X)
,---- 1/12 ----- <0, 1, 4>
|---- 1/12 ----- <0, 1, 5>
|---- 1/12 ----- <0, 2, 4>
|---- 1/12 ----- <0, 2, 5>
|---- 1/12 ----- <0, 3, 4>
<> -|
|---- 1/12 ----- <0, 3, 5>
|---- 1/32 ----- <1, 7, 4>
|---- 1/32 ----- <1, 7, 5>
|---- 7/32 ----- <1, 9, 4>
`----- 7/32 ----- <1, 9, 5>
pgd> psi = Max
pgd> phi = Max - Min
```

Here, the statistic ψ computes the maximum component value, and the statistic φ computes the range of component values.

Suppose I want the risk-neutral price for $\phi(X)$ having observed the fact that $\psi(X) < 9$. We might want to write this as $E(\phi(X) \mid (\psi < 9))$ in `frplib`, but as mentioned earlier, this does not work because the observational constraint sees $\phi(X)$ but “forgets” X . We can apply ϕ after constraining with the observation using

$$E(\phi(X \mid (\psi < 9)))$$

but the `@` operator gives us an equivalent expression that more closely tracks our

mathematical notation $\mathbb{E}(\phi(X) \mid \psi(X) < 9)$:

```
pgd> E( phi @ X | (psi < 9) )
14/3
```

or if you like,

```
pgd> E( phi@X | (psi < 9))
14/3
```

Think of `phi @ X` (or equivalently `phi@X`), read “phi at X”, as evoking the idea of evaluating a function *at* an argument. The only difference from `phi(X)` is that it passes the value of `X` itself to the condition. This notation works with `Kinds` too, as we will see.

Now, we want to find $\mathbb{E}(\text{phi@X} \mid (\text{psi} == a))$ for each possible value `a` of `psi(X)`. For our concrete example this can be expressed as

```
E( (Max - Min) @ X | (Max == a) ).
```

(The parentheses around `Max == a` are necessary.) This gives our prediction of the range of `X`’s components given an observation of only the maximum component of `X`. Try evaluating these in the playground; you should get expectations 4, 5, 6, and 8 when `a` is 4, 5, 7, and 9, respectively.

Rather than just computing the expectations, it makes sense to consider the associated `Kinds`. In the playground, we can do

```
pgd> range_given_max = conditional_kind({
...>     4: phi @ kind(X) | (Max == 4),
...>     5: phi @ kind(X) | (Max == 5),
...>     7: phi @ kind(X) | (Max == 7),
...>     9: phi @ kind(X) | (Max == 9),
...> })
```

or with an “anonymous” function (denoted by `lambda` in Python):

```
pgd> range_given_max = conditional_kind(
...>     lambda a: phi @ kind(X) | (Max == a)
...> )
```

We'll stick with the first form for now. Print this conditional Kind (in the first form) in the playground to see the results laid out nicely.

The `E` operator in `frplib` can compute all these risk-neutral prices as a package:

```
pgd> f = E(range_given_max)
```

which returns a *function* from `a` to the risk neutral price we want.

```
pgd> [f(a) for a in [4, 5, 7, 9]]
[4, 5, 6, 8]
```

Nice.

The previous example illustrates another useful property of expectations. The statistic φ in the example has a special form: it can be expressed as an operation on the value of ψ : $\varphi(x) = \zeta(x, \psi(x))$ for the function $\zeta(x, y) = y - \text{Min}(x)$. Our predictions about the value $\varphi(X)$ given the knowledge that $\psi(X) = a$ are thus the *same* as our predictions about $a - \text{Min}(X)$ given that same knowledge. We can check this in the playground:

```
pgd> E( (4 - Min)@X | (Max == 4) )
4
pgd> E( (Max - Min)@X | (Max == 4) )
4
```

Given that we know the value of $\psi(X)$, we can *substitute* that value in to the expression for $\phi(X)$ without changing our predictions.

This is one way to state the **substitution property** of expectations. It lets us use given information to *simplify* the expressions for the quantities we want to predict.

Substitution Property. If X is an FRP and ψ, φ are compatible statistics where

$$\varphi(x) = \zeta(x, \psi(x))$$

for some functions ζ , then

$$\mathbb{E}(\zeta(X, \psi(X)) \mid \psi(X) = a) = \mathbb{E}(\zeta(X, a) \mid \psi(X) = a). \quad (6.20)$$

Key Take Aways

For any FRP X , we define the **expectation** of X , denoted $\mathbb{E}(X)$. If X is a scalar (1-dimensional) FRP, its expectation is just its risk-neutral price. If X has dimension $n > 1$ and FRPs $\langle X_1, X_2, \dots, X_n \rangle$ are its scalar components, then $\mathbb{E}(X)$ is an n -tuple of numbers defined by the risk-neutral prices of its components:

$$\mathbb{E}(X) = \langle \mathbb{E}(X_1), \mathbb{E}(X_2), \dots, \mathbb{E}(X_n) \rangle, \quad (6.21)$$

$\mathbb{E}(X)$ depends only on $\text{kind}(X)$ not on X 's produced value.

The logic of risk-neutral prices gives us several key properties of expectations for any FRP X :

- **Constancy.** If X is a constant FRP, with one possible value v , then

$$\mathbb{E}(X) = v. \quad (6.22)$$

- **Scaling.** For any real number s ,

$$\mathbb{E}(sX) = s \mathbb{E}(X). \quad (6.23)$$

- **Ordering.** If all possible values of X are $\geq a$ and $\leq b$, then

$$a \leq \mathbb{E}(X) \leq b \quad (6.24)$$

- **Additivity.** If $X_1, X_2, \dots, X_n$ are FRPs of *common dimension*, then

$$\mathbb{E}(X_1 + X_2 + \dots + X_n) = \mathbb{E}(X_1) + \mathbb{E}(X_2) + \dots + \mathbb{E}(X_n). \quad (6.25)$$

- **Substitution.** If X is an FRP and ψ and $\langle x \rangle \mapsto \zeta(x, \psi(x))$ are compatible statistics

$$\mathbb{E}(\zeta(X, \psi(X)) \mid \psi(X) = a) = \mathbb{E}(\zeta(X, a) \mid \psi(X) = a). \quad (6.26)$$

6.2 Computing Expectations

Let us take stock of what we have so far:

1. A precise definition of risk-neutral prices that captures our “best” prediction of a scalar FRP’s value and depends only on an FRP’s Kind.
2. A simple notation for the risk-neutral price of an FRP.
3. The idea of *expectation* that extends the risk-neutral price to FRPs of any dimension.
4. A set of key properties derived from the logic of the definition that expectations must follow for any FRP.

These are powerful enough already to compute predictions in many cases, but it would be nice if there were a direct way to express the expectation of an FRP/Kind. The good news is that there is and that we can find it from the properties of risk-neutral prices!

First, let’s motivate the argument empirically by looking at large demos of FRPs of a particular Kind. We will compute the average payoff for each demo and compare it to the Kind.

```
pgd> Market.demo(10_000,
...>             weighted_as(-5, 0, 1, 10, weights=[1, 4, 3, 2]))
Activated 10000 FRPs with kind
      ,----- 1 ---- <-5>
      |----- 4 ---- <0>
<> -|
      |----- 3 ---- <1>
      `----- 2 ---- <10>

Summary of output values:
+-----+-----+-----+
| Values | Count | Proportion |
|=====|=====|=====|
|    -5  | 1001  | 10.01%    |
|     0  | 4065  | 40.65%    |
|     1  | 3002  | 30.02%    |
|     10 | 1932  | 19.32%    |
+-----+-----+-----+
```

The average payoff for all these FRPs is

$$\begin{aligned} & \frac{1001 \cdot (-5) + 4065 \cdot 0 + 3002 \cdot 1 + 1932 \cdot 10}{10,000} \\ &= 0.1001 \cdot (-5) + 0.4065 \cdot 0 + 0.3002 \cdot 1 + 0.1932 \cdot 10 \\ &= \mathbf{1.7317}. \end{aligned}$$

Increasing the size of the demo, we have

```
pgd> Market.demo(1_000_000,
...>           weighted_as(-5, 0, 1, 10, weights=[1, 4, 3, 2]))
Activated 1000000 FRPs with kind
      ,----- 1 ---- <-5>
      |----- 4 ---- <0>
<> -|
      |----- 3 ---- <1>
      `----- 2 ---- <10>

Summary of output values:
+-----+-----+-----+
| Values | Count  | Proportion |
|=====|=====|=====|
|    -5  | 100466 | 10.04%     |
|     0  | 400263 | 40.02%     |
|     1  | 299594 | 29.96%     |
|     10 | 199677 | 19.97%     |
+-----+-----+-----+
```

The average payoff for all these FRPs is

$$\begin{aligned} & \frac{100466 \cdot (-5) + 400263 \cdot 0 + 299594 \cdot 1 + 199677 \cdot 10}{1,000,000} \\ &= 0.100466 \cdot (-5) + 0.400263 \cdot 0 + 0.299594 \cdot 1 + 0.199677 \cdot 10 \\ &= \mathbf{1.7943994} \end{aligned}$$

And an even larger demo

```
pgd> Market.demo(100_000_000,
...>           weighted_as(-5, 0, 1, 10, weights=[1, 4, 3, 2]))
Activated 100000000 FRPs with kind
```



```

,----- 1 ---- <-5>
|----- 4 ---- <0>
<> -|
|----- 3 ---- <1>
`----- 2 ---- <10>
Summary of output values:
+-----+-----+-----+
| Values | Count   | Proportion |
|=====|=====|=====|
|    -5  | 9999815 | 9.99%      |
|     0  | 40004024 | 40.00%     |
|     1  | 29997871 | 29.99%     |
|    10  | 19998290 | 19.99%     |
+-----+-----+-----+

```

The average payoff for all these FRPs is

$$\begin{aligned}
 & \frac{9999815 \cdot (-5) + 40004024 \cdot 0 + 29997871 \cdot 1 + 19998290 \cdot 10}{100,000,000} \\
 &= 0.09999815 \cdot (-5) + 0.40004024 \cdot 0 + 0.29997871 \cdot 1 + 0.19998290 \cdot 10 \\
 &= \mathbf{1.79981696}
 \end{aligned}$$

As we've seen earlier, the more FRPs we demo, the closer the relative frequencies in this table will get to the relative weights in the Kind. Now, suppose we purchase these FRPs for a price \$c for a large batch, then our payoff per unit will be approximately

$$-5 \cdot 0.1 + 0 \cdot 0.4 + 1 \cdot 0.3 + 10 \cdot 0.2 = \mathbf{1.8},$$

where the approximation gets better and better as we purchase a larger and larger batch. If we choose a price $c < 1.8$, then for a sufficiently large batch, our payoff per unit will be positive. So any $c < 1.8$ is an arbitrage price. If $c > 1.8$, then for a sufficiently large batch, our payoff per unit will be negative, so no such price is an arbitrage price. And indeed, the risk-neutral price for this Kind is 1.8.

Even simpler, do the same calculation for the Kind of an event:¹⁰

$$\langle \rangle - \begin{cases} 1-p & \langle 0 \rangle \\ p & \langle 1 \rangle \end{cases}$$

In a large demo FRPs with this Kind, a proportion of roughly p of them will payoff

¹⁰Recall that an event is an FRP whose only possible values are 0 and 1.

\$1, with that proportion tending closer to p as the number of FRPs in the demo increases. If we pay more than $\$q$ per such FRP with $q > p$, then for any large enough demo with n FRPs, we are essentially certain to receive $< \$nq$. So q is an arbitrage price, and the risk-neutral price of this Kind is p .

Expectation of an Event. If V is an event, the risk-neutral price $\mathbb{E}(V)$ equals the weight on the $\langle 1 \rangle$ branch of $\text{kind}(V)$.

If X is an FRP of size m with values $v_1, v_2, \dots, v_m$ and respective weights $w_1, w_2, \dots, w_m$, we can define statistics

$$\psi_i(x) = \{x = v_i\}, \text{ for } i \in [1..m],$$

that equals 1 when its input equals v_i and 0 otherwise. Each transformed FRP $\psi_i(X)$ is an event – the event that X 's value equals v_i , and by the argument above

$$\mathbb{E}(\psi_i(X)) = \frac{w_i}{w_1 + w_2 + \dots + w_m}. \quad (6.27)$$

Define the statistic

$$\xi(x) = v_1\psi_1(x) + v_2\psi_2(x) + \dots + v_m\psi_m(x).$$

Notice that if we restrict attention to $x \in \{v_1, \dots, v_m\}$, then $\xi(x) = x$, but for any other value x , $\xi(x) = 0$. Thus,

$$X = \xi(X) = v_1\psi_1(X) + v_2\psi_2(X) + \dots + v_m\psi_m(X).$$

So, by the Scaling and Additivity properties and (6.27)

$$\begin{aligned} \mathbb{E}(X) &= \mathbb{E}(v_1\psi_1(X) + v_2\psi_2(X) + \dots + v_m\psi_m(X)) \\ &= \mathbb{E}(v_1\psi_1(X)) + \mathbb{E}(v_2\psi_2(X)) + \dots + \mathbb{E}(v_m\psi_m(X)) \\ &= v_1 \mathbb{E}(\psi_1(X)) + v_2 \mathbb{E}(\psi_2(X)) + \dots + v_m \mathbb{E}(\psi_m(X)) \\ &= v_1 \frac{w_1}{w_1 + w_2 + \dots + w_m} + \dots + v_m \frac{w_m}{w_1 + w_2 + \dots + w_m} \\ &= \frac{v_1w_1 + v_2w_2 + \dots + v_mw_m}{w_1 + w_2 + \dots + w_m}. \end{aligned} \quad (6.28)$$

Notice also that this argument holds *as is* with X of *any dimension*. The expectation of X is therefore a **weighted average of the FRP's possible values using the weights of each value from its Kind**. If the weights are from the canonical form

of the Kind, then the denominator in the last equality would be 1.

This gives us a formula for the expectation of any FRP. The formula applies for FRPs of *any dimension* because we can scale and add tuples of common dimension.

Definition 36. If X is an FRP of any dimension with size m and values $v_1, \dots, v_m$ and whose Kind has corresponding *canonical* weights $p_1, \dots, p_m$, then the expectation of X can be computed by

$$\mathbb{E}(X) = p_1 v_1 + \dots + p_m v_m. \quad (6.29)$$

If the Kind of X is in compact but not canonical form with corresponding weights $w_1, \dots, w_m$, then equation 6.29 becomes

$$\mathbb{E}(X) = \frac{w_1 v_1 + \dots + w_m v_m}{w_1 + \dots + w_m}. \quad (6.30)$$

Expectations are thus *weighted averages* of the Kind's/FRP's possible values.

We tend to use p 's to indicate canonical weights (that sum to 1) and w 's when this need not be true.

The formula (6.29) immediately generalizes. If X is an FRP and ψ is a compatible statistic, we can transform $\text{kind}(X)$ to obtain $\psi(\text{kind}(X)) = \text{kind}(\psi(X))$. Applying (6.29) to that Kind gives $\mathbb{E}(\psi(X))$. But because we know the Kind transforms, we can find that expectation from $\text{kind}(X)$ and ψ .

Specifically, each leaf node of $\text{kind}(\psi(X))$ is a transform of a value of X by ψ . Suppose $v_1, v_2, \dots, v_m$ are the possible values of X and $p_1, p_2, \dots, p_m$ are the corresponding canonical weights. If values $v_{j_1}, \dots, v_{j_n}$ all map under ψ to the same value u , then the corresponding branch of $\text{kind}(\psi(X))$ has value $u = \psi(v_{j_1}) = \dots = \psi(v_{j_n})$ and weight $p_{j_1} + \dots + p_{j_n}$. The corresponding term in $\mathbb{E}(\psi(X))$ using (6.29) is

$$(p_{j_1} + \dots + p_{j_n})u = p_{j_1} \psi(v_{j_1}) + \dots + p_{j_n} \psi(v_{j_n}).$$

Putting all these terms together, (6.29) becomes

$$\mathbb{E}(\psi(X)) = p_1 \psi(v_1) + p_2 \psi(v_2) + \dots + p_m \psi(v_m).$$

Thus, from $\text{kind}(X)$, we can find $\mathbb{E}(\psi(X))$ for any ψ . If we think of statistics like ψ as embodying questions about X 's value, the, $\text{kind}(X)$ lets us compute the predicted answer to any question. In this sense: **$\text{kind}(X)$ describes our knowledge of X 's**

value.

Definition 37. If X is an FRP of any dimension with size m and values $v_1, \dots, v_m$ and whose Kind has corresponding *canonical* weights $p_1, \dots, p_m$, then for any compatible statistic ψ : expectation of X can be computed by

$$\mathbb{E}(\psi(X)) = p_1\psi(v_1) + \dots + p_m\psi(v_m) = \sum_{i=1}^m p_i\psi(v_i). \quad (6.31)$$

Let $\mathbf{k}_X$ denote the kernel of $\text{kind}(X)$. Then, we can rewrite equation (6.31) in terms of the kernel as

$$\mathbb{E}(\psi(X)) = \sum_v \mathbf{k}_X(v) \psi(v), \quad (6.32)$$

where the sum is over all v . We call this the **mass-balancing equation**.

Equations (6.31) and (6.32) are equivalent because the kernel $\mathbf{k}_X(v)$ is only non-zero for the finite set of values $v_1, v_2, \dots, v_m$ where it returns the respective canonical weights $p_1, p_2, \dots, p_m$. The mass-balancing equation is the fundamental formula we use for direct calculation of expectations from Kinds.

Example 6.9. If X has Kind described by $\langle \rangle \quad 1 \quad \langle -1 \rangle \quad 4 \quad \langle 1 \rangle$,

$$\langle \rangle \begin{cases} \text{---} 1 \text{---} \langle -1 \rangle \\ \text{---} 4 \text{---} \langle 1 \rangle \end{cases}$$

what are $\mathbb{E}(X)$, $\mathbb{E}(X^3)$, and $\mathbb{E}(X \star X)$?

(Recall that X^2 and X^3 are “inlined” expressions for the transformed FRPs by statistics $\psi(x) = x^2$ and $\phi(x) = x^3$, respectively. See page 131.)

Applying equation 6.29 to each of these, we have

$$\begin{aligned} \mathbb{E}(X) &= \frac{1}{5} \cdot (-1) + \frac{4}{5} \cdot 1 = \frac{3}{5} \\ \mathbb{E}(X^3) &= \frac{1}{5} \cdot (-1) + \frac{4}{5} \cdot 1 = \frac{3}{5} \\ \mathbb{E}(X \star X) &= \frac{1}{25} \cdot \langle -1, -1 \rangle + \frac{4}{25} \cdot \langle -1, 1 \rangle \\ &\quad + \frac{4}{25} \cdot \langle 1, -1 \rangle + \frac{16}{25} \cdot \langle 1, 1 \rangle \\ &= \langle \frac{3}{5}, \frac{3}{5} \rangle. \end{aligned}$$

To see where the weights on the last two came from, compute the Kinds in

frplib. The Kind of X is `choice(-1, 1, 4)`.

Example 6.10. I made two claims about the Kinds in Figure 6.1: (i) that the three Kinds have the same risk-neutral price, and (ii) that many people would *not* be indifferent between purchasing FRPs with these Kinds for \$10 because they differ in their *risk* of an adverse payoff. Let's confirm them.

Claim (i). The first Kind is just a constant with value 10, so its expectation is 10. The second Kind has values -1000 and 11 with weights 1 and 1010, so its expectation is

$$\frac{-1000 \cdot 1 + 11 \cdot 1010}{1011} = \frac{10110}{1011} = 10.$$

And the third Kind has values -10000 and 110 with weights 1 and 100.1, with expectation

$$\frac{-10000 \cdot 1 + 110 \cdot 100.1}{101.1} = \frac{1011}{101.1} = 10.$$

Claim (ii). To assess this, we will use the “variance shortcut” in equation (6.19) to compute the variance (a measure of uncertainty from Examples 6.6 and 6.4). If Z is an FRP with one of the three Kinds, then $\text{Var}(Z) = \mathbb{E}(Z^2) - (\mathbb{E}(Z))^2 = \mathbb{E}(Z^2) - 100$. For the first Kind, which is constant, there is no uncertainty, so the variance should be 0. And indeed: $10^2 - 100 = 0$. If Z has the second Kind,

$$\mathbb{E}(Z^2) = \frac{(-1000)^2 \cdot 1 + 11^2 \cdot 1010}{1011} = 1110,$$

so the variance is 1010. If Z has the third Kind,

$$\mathbb{E}(Z^2) = \frac{(-10000)^2 \cdot 1 + 110^2 \cdot 100.1}{101.1} = 1001100,$$

so the variance is 1001000. The uncertainty in the three Kinds increases as the possible values become more spread.

Part of the “Measuring Uncertainty” series.

Example 6.11. In the revisited disease testing example on page 364, we computed the Kind of an FRP, $D \mid \{T = 1\}$, whose outcome indicates whether someone has the disease when it is known that they test positive.

$$\langle \rangle \begin{cases} \text{---} 999/1094 \text{---} \langle 0 \rangle \\ \text{---} 95/1094 \text{---} \langle 1 \rangle \end{cases}$$

where the events D and T represents the disease status and test result, respectively. The FRP $D \mid \{T = 1\}$ is also an event, so we know how to compute its expectation. Applying the mass-balancing equation, we have

$$\mathbb{E}(D \mid T = 1) = \frac{999}{1094} \cdot 0 + \frac{95}{1094} \cdot 1 = \frac{95}{1094} \approx 0.0868.$$

This tells us that after observing a positive test result, our prediction is that the patient has only about an 8.6% chance of having the disease.

This illustrates another common pattern. An event¹¹ acts as an indicator of whether something happens. Equation (6.29) shows that for any event I , $\mathbb{E}(I)$ is just the canonical weight on its 1 branch.¹² We interpret this quantity as a measure of how likely that event is to occur. In that sense, the disease-testing example is suprising in that the disease remains unlikely after a positive test.

¹¹Recall: FRP whose only values are 0 and 1.

¹²The discussion just after Puzzle 82 also made good use of this fact.

Example 6.12. Try these calculations; look at the weighted averages that you get; and compare them to the expectations that we compute. I've omitted the results here.

```
pgd> coin = choice(0, 1) # Model: 1 for heads, 0 for tails; equally weighted
pgd> flips10 = coin ** 10
pgd> num_heads_in_10 = Sum(flips10)
pgd> num_heads_in_10
pgd> E(num_heads_in_10)
pgd> E(num_heads_in_10 - 5) # We know this from Additivity and Constancy. Why?
```

We can see that the Kind's canonical form shows us the weighs and values and can confirm that the resulting weighted average is just the risk-neutral price. Notice how the Properties we discovered earlier help us compute the last value without actually hitting enter.

For a value like $E(\text{num\_heads\_in\_10})$, there are two ways to think about the computation. We can look at the Kind `flips10`, takes its expectation, and then sum its components. Or, we can generate the *new* Kind `num_heads_in_10 = Sum(flips10)` and take its expectation. *Both ways give the same answer.*

Let's consider that in a smaller case. Look at both Kinds `coin ** 4` and `Sum(coin ** 4)` and do the calculation from each tree:

```
pgd> coin ** 4
pgd> Sum(E(coin ** 4))
```

```
pgd> E(Sum(coin ** 4))
```

The reason these give the same answer is that we defined the transformed Kind by passing the values through the statistic and combining equal weights.

Example 6.13. Continuing the previous example, let us ask at which of 16 flips of our coin does the first heads occur.

```
pgd> flips16 = coin ** 16
```

We define a statistic that answers our question

```
pgd> @scalar_statistic(description='Index of first heads, or 1000')
...> def first_heads(x):
...>     return 1 + index_of(1, x, not_found=999)
```

This has type $n \rightarrow 1$ for every $n \in [1..1000]$, and returns 1000 (arbitrarily) if a heads did not occur.

```
pgd> when_heads = first_heads(flips16)
```

Look at the following Kinds:

```
pgd> when_heads
pgd> when_heads ~ IfThenElse(__ > 10, 1000, __)
pgd> (when_heads | (__ > 4)) ~ IfThenElse(__ == 1000, __, __ - 5)
```

What do each of these Kinds mean? How do they compare? Can you explain?

The first is the Kind of the number of flips to the first heads in 16 flips. The second is a transform of the Kind `when_heads`, where we map every value bigger than 10 to the arbitrary value 1000. We do this to ease comparison with the third Kind. The third is the Kind of an FRP that gives the number of 1's (heads) *after* having observed that there are no 1's (heads) in the first four flips. The statistic at the end shifts the values back to the scale of the first two Kinds.

Example 6.14 Entropy

Example 6.4 introduced the *entropy* of an FRP/Kind as a measure of uncertainty. Unlike variance or mean absolute deviation, the entropy does not predict the distance between value and risk-neutral price but rather predicts the size of the

Part of the “Measuring Uncertainty” series.

weight on the FRP's value. Here, we will use equation (6.29) to get an expression for the entropy and develop a first interpretation of its meaning.

Let Z be an FRP with Kind K in canonical form. If z is a possible value of Z , we write $K(z)$ for the weight associated with branch z , treating K as a function. Then, applying (6.29), we have

$$\begin{aligned}\mathbb{H}(Z) &= \mathbb{E}(-\lg K(Z)) \\ &= \sum_{z \in \text{values}(Z)} K(z)(-\lg K(z)) \\ &= - \sum_{z \in \text{values}(Z)} K(z) \lg K(z).\end{aligned}\tag{6.33}$$

The terms in the sum are well defined for any value of the weights: the function $\langle p \rangle \mapsto -p \lg p$ goes to 0 as p goes to 0 because $-\lg p$ grows much more slowly than p shrinks. We can thus write $\mathbb{H}(Z) = -\sum_z K(z) \lg K(z)$, taking $K(z) = 0$ for any z that is not a possible value of Z .

To start, assume that we have a large supply of balanced, independent coin flips, i.e., events $C_{[1]}, C_{[2]}, C_{[3]}, \dots$ with Kind F $\langle \rangle \begin{array}{l} \xrightarrow{\frac{1}{2}} \langle 0 \rangle \\ \xrightarrow{\frac{1}{2}} \langle 1 \rangle \end{array}$, where 0 represents tails and 1 represents heads. Think of a coin flip as a random bit, so a single flip represents 1 bit of randomness.

Suppose we have another FRP X with Kind F . This has entropy

$$\mathbb{H}(X) = -\frac{1}{2} \lg \frac{1}{2} - \frac{1}{2} \lg \frac{1}{2} = \frac{1}{2} + \frac{1}{2} = 1.$$

Entropy is measured in *bits*, and this result tells us that X has 1 bit of “uncertainty.” One way to interpret this is that it takes 1 coin flip to simulate a clone of X . For instance, we can take the value of $C_{[1]}$ as the value of our clone. This has the same Kind as X .

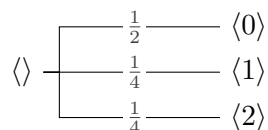
Let Y be an FRP with Kind

$$\langle \rangle \begin{array}{l} \xrightarrow{\frac{1}{4}} \langle 0 \rangle \\ \xrightarrow{\frac{1}{4}} \langle 1 \rangle \\ \xrightarrow{\frac{1}{4}} \langle 2 \rangle \\ \xrightarrow{\frac{1}{4}} \langle 3 \rangle \end{array}$$

Then, $\mathbb{H}(Y) = -\frac{1}{4} \cdot (-2) - \frac{1}{4} \cdot (-2) - \frac{1}{4} \cdot (-2) - \frac{1}{4} \cdot (-2) = 2$. We can simulate a

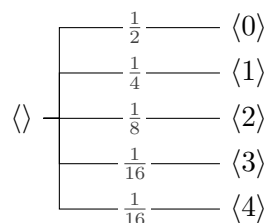
value of `clone(Y)` with two coin flips, say $C_{[2]}$ and $C_{[3]}$, with the former's value determining the ones-bit of the simulated value and the latter's value determining the twos-bit of the simulated value: $2C_{[3]} + C_{[2]}$. Two coin flips, two bits of uncertainty.

The FRP Z with Kind



has entropy $\mathbb{H}(Z) = -\frac{1}{2} \cdot (-1) - \frac{1}{4} \cdot (-2) - \frac{1}{4} \cdot (-2) = 1.5$ bits. So this Kind embodies 1.5 bits of uncertainty.

But what does it mean to require 1.5 coin flips? Well, to simulate a value of `clone(Z)`, we can start with a coin flip $C_{[4]}$. *If* its value is 0 (tails), we use 0 as the simulated value and we are done, which requires 1 flip. If $C_{[4]}$'s value is 1, however, we pull out $C_{[5]}$ and give a simulated value of 1 or 2 as this flip has value 0 (tails) or 1 (heads), which requires 2 flips. So half the time we need only 1 flip and half the time 2 flips – thus 1.5 flips.



The entropy of an FRP with the Kind above is 2.375. Explain how this value is derived. How would you use coin flips $C_{[6]}, C_{[7]}, C_{[8]}, C_{[9]}$ to simulate the value of an FRP with this Kind? How might you interpret the value of the entropy in light of this?

You need not use all four coin flips for every value.

The `entropy` function in `frplib` computes the entropy of a Kind or FRP. For instance, try

```
pgd> entropy(uniform(1, 2, ..., 16))
```

One interpretation of entropy is that it measures how many bits of randomness are used on average in generating the value of an FRP. We will see other, related interpretations later.

Key Take Aways

For any Kind k in canonical form with values $v_1, \dots, v_m$ and corresponding weights $p_1, \dots, p_m$, then for any FRP X with Kind k and any compatible statistic ψ :

$$\mathbb{E}(\psi(X)) = p_1\psi(v_1) + \dots + p_m\psi(v_m) = \sum_v k(v) \psi(v). \quad (6.34)$$

We can express this equivalently in terms of the kernel of $\text{kind}(X)$, also denoted by k_X , giving

$$\mathbb{E}(\psi(X)) = \sum_v k_X(v) \psi(v). \quad (6.35)$$

This is the **mass-balancing equation**.

6.3 Kinds and Expectations

Because the FRP's expectation depends only on the FRP's Kind, we can therefore talk about the expectation of an FRP or the risk-neutral price of a Kind, as convenient. The expectation represents a *prediction* of an FRPs value, a notion of “typical value” that is *in some sense* as close as possible to whatever value is produced by the FRP. In some sense?? What does that mean?

Let X be a scalar FRP and consider a family of statistics $\psi_c(x) = (x - c)^2$ for every real number c . Define a function

$$L(c) = \mathbb{E}(\psi_c(X)) = \mathbb{E}((X - c)^2),$$

where the second form is the inline expression of the statistic.¹³ This function gives the risk-neutral price of an FRP that represents *the squared distance between the value of X and the number c* . Thus, $L(c)$ measures for each c how “close” c is on average to the value of X . The value that *minimizes* $L(c)$ is $\mathbb{E}(X)$. To see this apply the properties of expectations:

$$\begin{aligned} L(c) &= \mathbb{E}((X - c)^2) \\ &= \mathbb{E}((X - \mathbb{E}(X) + \mathbb{E}(X) - c)^2) && \textcircled{1} \\ &= \mathbb{E}((X - \mathbb{E}(X))^2 + 2(\mathbb{E}(X) - c)(X - \mathbb{E}(X)) + (\mathbb{E}(X) - c)^2) && \textcircled{2} \end{aligned}$$

¹³See page 131.

$$= \mathbb{E}((X - \mathbb{E}(X))^2) + \mathbb{E}(2(\mathbb{E}(X) - c)(X - \mathbb{E}(X))) + \mathbb{E}((\mathbb{E}(X) - c)^2) \quad \textcircled{3}$$

$$= \mathbb{E}((X - \mathbb{E}(X))^2) + 2(\mathbb{E}(X) - c)\mathbb{E}(X - \mathbb{E}(X)) + (\mathbb{E}(X) - c)^2 \quad \textcircled{4}$$

$$= \mathbb{E}((X - \mathbb{E}(X))^2) + 2(\mathbb{E}(X) - c)0 + (\mathbb{E}(X) - c)^2 \quad \textcircled{5}$$

$$= \mathbb{E}((X - \mathbb{E}(X))^2) + (\mathbb{E}(X) - c)^2.$$

In $\textcircled{1}$, we add $0 = \mathbb{E}(X) - \mathbb{E}(X)$ inside quadratic. In $\textcircled{2}$, expand the quadratic in terms of $X - \mathbb{E}(X)$ and $\mathbb{E}(X) - c$. $\textcircled{3}$ applies Additivity. $\textcircled{4}$ applies Scaling on the middle term with constant $2(\mathbb{E}(X) - c)$ and Constancy on the last term. In $\textcircled{5}$, the middle term cancels by Constancy, Scaling, and Additivity because

$$\mathbb{E}(X - \mathbb{E}(X)) = \mathbb{E}(X) - \mathbb{E}(\mathbb{E}(X)) = \mathbb{E}(X) - \mathbb{E}(X) = 0.$$

Finally, we are left with two terms: the first *does not depend on c* ; you might recognize it as $\text{Var}(X)$. The second term is a simple quadratic in c , which is minimized at $c = \mathbb{E}(X)$. So, $c = \mathbb{E}(X)$ minimizes $L(c)$, and the minimum value is $\text{Var}(X)$, the variance of the FRP.

If we define $|x| = \sqrt{x_1^2 + x_2^2 + \cdots + x_d^2}$ for tuples x of dimension d . This argument carries over, almost directly, to FRPs of any dimension. Here, we use a family of statistics $(|\blacksquare - c|^2)$, where c is a tuple of dimension d and $x - c$ is the component-wise difference. This reduces to ψ_c in the scalar case.

The Expectation Minimizes the Predicted Squared Prediction Error.

If X is an FRP of dimension d , its expectation $\mathbb{E}(X)$ is a tuple of dimension d that minimizes the *predicted squared prediction error*

$$\langle c \rangle \mapsto \mathbb{E}(|X - c|^2). \quad (6.36)$$

over all d -tuples c .

When X is a scalar FRP, the minimum value $\mathbb{E}(|X - \mathbb{E}(X)|^2)$ is called the **variance** of X , denoted $\text{Var}(X)$, as discussed in Examples 6.4 and 6.6. (The idea of variance also extends to the multi-dimensional case but with some nuance, so we will discuss it later.)

Thus, the sense in which the expectation is an optimal prediction of an FRP's value is it that the expectation minimizes our “mean squared” prediction error. The expectation is a “typical value” that is on average as close as possible to the value produced by the FRP. As we proceed, we will see several different interpretations of

expectations, and we will emphasize the connection to risk-neutral prices.

Puzzle 80. Consider a Kind like `uniform(1, 2, ..., 999999)`. For each $j \in [1..999999]$, what is $\mathbb{E}(|X - j|^2)$, where FRP X has this Kind? (You can do this by reasoning or by computation.) Which value gives the smallest predicted prediction error?

We have seen that the expectation of an FRP is determined by the FRP's Kind. It turns out the Kind of an FRP is determined by expectations of *all transforms* of the FRP by compatible statistics.

Property 38. Two FRPs X and Y have the same Kind if and only if they have the same set of possible values and

$$\mathbb{E}(\psi(X)) = \mathbb{E}(\psi(Y)) \quad (6.37)$$

for *every compatible statistic* ψ .

This tells us that, *in aggregate*, the expectations of transformed FRPs/Kinds contain all the information needed to determine the Kind of the original FRP. The word “determine” here does not necessarily mean that we can compute the Kind directly from this information (though we often can) but rather that any computation, analysis, or prediction that depends only on the Kind will be the same for any FRP with that Kind. This needs a little unpacking.

In one direction, Property 38 says that two FRPs with the same Kind yield equal expectations when transformed by the same statistic. The other direction *seems* less useful because it requires that the expectations of transformed FRPs be the same for all compatible statistics. (It's hard to compute the expectation in practice for *all* compatible statistics.) Fortunately, this direction also holds with smaller collections of statistics as long as the collection is sufficiently rich. In other words, we can “determine” the Kind of an FRP with the expectations of a well-chosen collection of statistics.

Definition 39. A collection of statistics $\mathcal{S}$ is said to **determine** the Kind of compatible FRPs if for any FRPs X, Y with the same possible values that are

compatible with all the statistics in $\mathcal{S}$

$$\mathbb{E}(\psi(X)) = \mathbb{E}(\psi(Y)) \text{ for all } \psi \in \mathcal{S} \text{ implies } \text{kind}(Y) = \text{kind}(X). \quad (6.38)$$

That is, if X and Y have the same expectation when transformed by all statistics in $\mathcal{S}$, they have the same risk-neutral price when transformed by *all* compatible statistics.

Chapter 2 emphasized that statistics often represent questions whose answers we would like to predict. It is useful then to view a collection of statistics as a set of questions we might want to answer. These determine the Kind of an FRP if the predicted answer to these questions uniquely specify the predicted answers to all questions we might ask.

This requirement is not trivial. For $\mathcal{S} = \{\text{id}\}$, the singleton collection containing only the identity function, this would require that $\mathbb{E}(X) = \mathbb{E}(Y)$ implied that X and Y had the same Kind, but that is not true, as we saw for instance in Figure 6.1.

Puzzle 81. For $\mathcal{S} = \{\text{id}, (\blacksquare^2)\}$, find two distinct Kinds that give the same expectation for all statistics in $\mathcal{S}$.

As a positive example, let $\mathcal{E}$ be the collection of statistics of type $1 \rightarrow 1$ that indicate a specific value:

$$\mathcal{E} = \{ \{ \blacksquare = c \} \mid c \text{ is a real number} \}.$$

(Recall that the indicator $\{ \blacksquare = c \}$ is the function that returns 1 when its argument equals c and 0 otherwise. See Chapters 12 and 11.) For a scalar FRP X , the transformed FRPs by the statistics in $\mathcal{E}$ are all the events $\{X = c\}$ for real number c . If c is not a possible value of X , then $\mathbb{E}(\{X = c\}) = 0$. If c is a possible value of X , then $\mathbb{E}(\{X = c\})$ is the canonical weight on value $\langle c \rangle$ in $\text{kind}(X)$. This collection of expectations thus gives us the kernel of $\text{kind}(X)$

$$k_X(c) = \mathbb{E}(\{X = c\}) \quad (6.39)$$

and thus the Kind itself. Another scalar FRP Y with $\mathbb{E}(\{Y = c\}) = \mathbb{E}(\{X = c\})$ for all c must have the same possible values and the same weights – and so the same Kind. Thus, $\mathcal{E}$ determines the Kind of any scalar FRP.

Another example is the collection of indicator statistics $\mathcal{F}$ of type $1 \rightarrow 1$

$$\mathcal{F} = \{ \{ \blacksquare \leq c \} \mid c \text{ is a real number} \}.$$

For a scalar FRP X , $\mathbb{E}(\{X \leq c\})$ sums the weights for all values of X that are at most c . By varying c , we can find X 's possible values (where the expectation jumps) and their weights (the size of the jump). So if $\mathbb{E}(\{X \leq c\}) = \mathbb{E}(\{Y \leq c\})$ for all c , X and Y must have the same Kind. $\mathcal{F}$ determines the Kind of scalar FRPs.

Let $\mathcal{T}$ be the set of three statistics $(\frac{1}{2}(\blacksquare - 1)(\blacksquare - 2))$, $(-\frac{1}{2}\blacksquare(\blacksquare - 2))$, and $(\frac{1}{2}\blacksquare(\blacksquare - 1))$ restricted to the domain $\{0, 1, 2\}$. For any FRP X with possible values $\{0, 1, 2\}$, the expectations $\mathbb{E}(\psi(X))$ for $\psi \in \mathcal{T}$ give the canonical weights on 0, 1, and 2. The statistics in $\mathcal{T}$ thus determine the Kind of any FRPs with only those three possible values.

Puzzle 82. If Z is an FRP of dimension 2, the collection $\text{proj}_1, \text{proj}_2$ does *not* determine the Kind of Z . Find an example to demonstrate this.

Specifically, you need another FRP U with the same possible values as Z where $\mathbb{E}(\text{proj}_1(Z)) = \mathbb{E}(\text{proj}_1(U))$ and $\mathbb{E}(\text{proj}_2(Z)) = \mathbb{E}(\text{proj}_2(U))$ but Z and U have *different Kinds*.

Looking back to Figure 2.8, a useful and powerful perspective on how we use probability theory in practice is that we measure some data from whatever system we are studying and try to predict the answers to a range of questions about those data. For each question, we want a predicted answer.

If X is an FRP, like the Data FRP in the Figure, we can define an operator D_X that does exactly that: maps questions to predicted answers.

$$D_X(\psi) = \mathbb{E}(\psi(X)). \quad (6.40)$$

This takes as input a statistic that is compatible with X – that is, a question we can ask about our data – and returns as output a predicted answer to that question.

Property 38 tells us that the expectations $\mathbb{E}(\psi(X))$, varying over compatible statistics ψ , determine the Kind of X . So D_X embodies the same information as $\text{kind}(X)$ in a different form. We can say that our *knowledge* about some random quantity is reflected by how well we can make predictions about the quantity. Both $\text{kind}(X)$ and D_X thus fully describe what we know about X 's value. As the system evolves, we may learn more information about X 's value (expressed through observational constraints), and our knowledge changes (to $\text{kind}(X \mid C)$ and $D_{X \mid C}$ for

conditional constraint C).

The D operator is available in `frplib` as D_- . It takes an FRP or Kind as input and returns a *function* mapping every (compatible) statistic to its corresponding prediction. If X is an FRP, $D_-(X)$ is a function that acts on statistics ψ to give

$$D_-(X)(\psi) = E(\psi(X)).$$

For example,

```
pgd> X = frp(uniform(1, 2, 3) * uniform(1, 2, 3))
pgd> f = D_-(X)
pgd> f(Sum)      # == E(Sum(X))
4
pgd> f(Max)      # == E(Max(X))
22/9
pgd> f(Min)      # == E(Min(X))
14/9
```

Definition 40. The Distribution Operator.

For any FRPX, its **distribution operator** D_x is a function that accepts any statistic ψ that is compatible with X and is defined by

$$D_x(\psi) = \mathbb{E}(\psi(X)). \quad (6.41)$$

In `frplib`, the distribution operator is D_- .

By the mass-balancing equation (6.32), we can write

$$D_x(\psi) = \sum_v k_x(v) \psi(v), \quad (6.42)$$

where k_x is the kernel of $\text{kind}(X)$. By choosing statistics $\psi_v(x) = \{x = v\}$, we get $D_x(\psi_v) = k_x(v)$. So:

- From the distribution operator, we can find the kernel and the Kind.
- From the kernel, we can find the Kind and the distribution operator.
- From the Kind, we can find the kernel and the distribution operator.

All three objects package the *same information* in different forms.

Key Take Aways

The expectation of a scalar FRP Y is the number that minimizes the *predicted squared prediction error*

$$\langle c \rangle \mapsto \mathbb{E}(|Y - c|^2).$$

The minimum value at $c = \mathbb{E}(Y)$ is called the **variance** of Y , denoted $\text{Var}(Y)$.

Two FRPs X and Y have the same Kind if and only if they have the same set of possible values and

$$\mathbb{E}(\psi(X)) = \mathbb{E}(\psi(Y)) \quad (6.43)$$

for *every compatible statistic* ψ . We can often find smaller collections of statistics that *determine* the Kind of FRPs.

The **distribution operator** of a Kind/FRP is defined by

$$D_X(\psi) = \mathbb{E}(\psi(X)), \quad (6.44)$$

a function that accepts as input *statistics* that are compatible with X (and its Kind).

The Kind, the kernel, and the distribution operator are each derivable from the others and provide the same information in different forms.

6.4 Probabilities are the Expectations of Events

In practice, we are often interested in whether particular events occur. Because an event is an FRP with possible values 0 and 1, the Ordering property (6.3) tells us that the expectation of an event is a *number between 0 and 1*. This is a prediction of how likely the event is to occur, increasing from an essential certainty that it will not occur at 0 to an essential certainty that it will occur at 1. with value 0 meaning that it will.

Thus for any event V , the expectation $\mathbb{E}(V)$ satisfies $0 \leq \mathbb{E}(V) \leq 1$ and measures in some sense our confidence that the event V will occur. Because we use such expectations so often, we give them a distinctive name.

Definition 41. A **probability** is the expectation of an event. It is a number in $[0, 1]$ that measures our prediction of whether the event will occur.

If V is an event, $\mathbb{E}(V)$ is called the *probability of V* . Events with higher probability are said to be more *likely* than events with lower probability.

Probabilities are nothing new. They are just expectations – risk-neutral prices – for FRPs that can have particular values. They inherit all the properties of general expectations and are computed according to the same rules.

If X is an FRP and if v is a possible value of X , then $\{X = v\}$ is the event that occurs when X produces value v and does not occur if it produces another value. The Kind of $\{X = v\}$ looks like

$$\langle \rangle \begin{cases} \text{---} 1-p \text{---} \langle 0 \rangle \\ \text{---} p \text{---} \langle 1 \rangle \end{cases}$$

where p is the canonical weight on v in $\text{kind}(X)$. Applying equation (6.29), we see that $\mathbb{E}(\{X = v\}) = p$. **The canonical weight on value v in the Kind $\text{kind}(X)$ is the probability that X has value v .**

Two events V and W are *complements* if $V + W = 1$. This means that exactly one of the two events must occur. If V occurs, then we know W does not, and vice versa. Applying Additivity ($\mathbb{E}(V + W) = \mathbb{E}(V) + \mathbb{E}(W)$) and Constancy ($\mathbb{E}(1) = 1$), we can take expectations on both sides of this equation to find

$$\mathbb{E}(V) + \mathbb{E}(W) = 1.$$

Complementary events have probabilities that sum to 1.

As in Table 5.1, for two events V and W , the event VW represents their logical-and. If V and W are *independent*, then $\text{kind}(VW) = \text{kind}(V \star W) \wedge (\blacksquare_1 \blacksquare_2)$. Equation 4.6 tells us the kernel of $\text{kind}(V \star W)$ and combined with the mass-balancing equation (6.32), we can express $\mathbb{E}(VW)$ in terms of the *kernels* k_V and k_W of $\text{kind}(V)$ and $\text{kind}(W)$:

$$\mathbb{E}(VW) = \sum_x k_V(x_1) k_W(x_2) (x_1 x_2) = \sum_{x_1} k_V(x_1) x_1 \sum_{x_2} k_W(x_2) x_2 = \mathbb{E}(V) \mathbb{E}(W).$$

The probability of independent events *all* occurring is the product of the events' probabilities.

As described in section 5.2, we frequently specify events with Boolean expressions in *Iverson braces*,¹⁴ like $\{X = 4\}$, $\{Y > 10\}$, and $\{U = 4 \wedge 0 \leq V \leq 10\}$. When

¹⁴On Iverson braces: see page 614.

taking the expectations, we typically put the FRP in parentheses after the $\mathbb{E}$, like $\mathbb{E}(\{X = 4\})$, $\mathbb{E}(\{Y > 10\})$, and $\mathbb{E}(\{U = 4 \wedge 0 \leq V \leq 10\})$. However, in this specific case where the FRP is a single event in Iverson braces, the extra parentheses may seem superfluous, and we treat them as optional. So $\mathbb{E}\{X = 4\}$ and $\mathbb{E}(\{X = 4\})$ mean the same thing.

Expectation of an Event (Optional). When taking the expectation of a *single event in Iverson braces*, the parentheses around the argument to $\mathbb{E}$ are *optional*.

Note that this convention does not apply to FRPs that are expressed as inlined statistics with multiple terms, such as $X \cdot \{X > 2\}$ or $\{Y = 2\} + \{Z < 4\}$, we keep the parentheses.

Key Take Aways

A **probability** is the expectation of an event.

Because an event can have only values 0 or 1, the Ordering Property of expectations tells us that a probability is a number in $[0\_1]$. If V is an event, $\mathbb{E}(V)$ is called the *probability of V* .

As an expectation, a probability is a prediction of whether the event will occur. that measures our prediction of whether the event will occur. The closer the probability is to 1, the more *likely* the event is said to be; the closer the probability is to 0, the less *likely* the event is said to be.

Keep in mind that **probabilities are just expectations** and inherit all their properties from that.

Chapter Checkpoints

After reading this Chapter you should be able to:

- Define an arbitrage price and the risk-neutral price for a scalar FRP.
- Use the market to estimate risk-neutral prices for simple FRPs.
- Use the definitions to guess at some basic properties of risk-neutral prices.
- Define the expectation of an n -dimensional FRP / Kind for $n \geq 1$.
- Explain the Constancy, Scaling, Ordering, and Additivity properties of

expectations.

- Describe various measures of uncertainty.
- Explain the formula for expectations from a Kind.
- Describe briefly why the expectation depends only on the Kind and not on the FRPs value.
- Describe how the expectations of transformed FRPs determine a Kind.
- Define probabilities in terms of expectations.

Three Dialogues: Computation, Systems, Simulation

7

Chapter

Contents

7.1	A Dialogue on Computation	436
7.2	A Dialogue on Systems and State	445
7.3	A Dialogue on Solutions and Simulation	460

Summary

In friendly interactions with clients and employees of the FRP Marketplace, you explore practical aspects of building probabilistic systems to solve problems.

A Dialogue on Computation highlights several computational techniques that are frequently useful, including “monoidal” parallelism, symmetry, and sampling.

A Dialogue on Systems and State examines how to build random systems that evolve over time or space and describes the idea of a system’s *state*.

A Dialogue on Solutions and Simulation develops tools for computing answers to questions about the long-term evolution of random systems as solutions to “one-step” equations or as simulations of the system’s dynamics.

7.1 A Dialogue on Computation

Alice and Bob are clients of the FRP Marketplace whom you met during the orientation for new users. This conversation took place during an orientation workshop.

BOB: I'm getting frustrated, Alice. This calculation is hanging.

ALICE: The dice example again? What's the problem?

BOB: I want to compute the Kind for an FRP that models the sum of 100 rolls of a six-sided dice. So I define the Kind for one roll, `d6 = uniform(1, 2, 3, 4, 5, 6)`, compute the Kind for 100 rolls – `d6 ** 100` – and then transform ...

ALICE: Well that's your problem right there. What is the size of `d6 ** 100`?

BOB: There are 6 possibilities for each of 100 rolls, so 6^{100} possible values. Ah...that's a big number. No wonder it's taking so long.

But the sum of the rolls doesn't care about the distinction between most of those possibilities, so it seems it should be possible to do this calculation efficiently.

ALICE: It actually is. I've been considering that problem for another project. What does the playground display when you print the statistic `Sum`?

BOB: Let's see. It says

A Monoidal Statistic 'sum' that returns the sum of all the components of the given value. It expects a tuple and returns a scalar.

What the heck is a "Monoidal Statistic?"

ALICE: That threw me too, but after I dug into it, I realized it was a simple idea.

The `Sum` statistic takes in a value that is a list of numbers and adds up all the components to give a number, so it takes in a list of numbers and returns a number. What happens to the sum if you add some elements to the list, as we do when we take joins?

BOB: The `Sum` just adds up the new elements and adds that sum to the total.

ALICE: Exactly! Let's write `::` for the operation of joining two lists, so `<10, 20, 30> :: <40, 50> = <10, 20, 30, 40, 50>` and `<10, 20> :: <> = <10, 20>` and so on. What you said is

$$\begin{aligned} \text{Sum}(\langle 10, 20, 30 \rangle :: \langle 40, 50 \rangle) &= \text{Sum}(\langle 10, 20, 30 \rangle) + \text{Sum}(\langle 40, 50 \rangle) \\ &= \text{Sum}(\langle \text{Sum}(\langle 10, 20, 30 \rangle), \text{Sum}(\langle 40, 50 \rangle) \rangle), \\ \text{Sum}(\langle 10, 20 \rangle :: \langle \rangle) &= \text{Sum}(\langle 10, 20 \rangle) + \text{Sum}(\langle \rangle) \\ &= \text{Sum}(\langle \text{Sum}(\langle 10, 20 \rangle), \text{Sum}(\langle \rangle) \rangle), \end{aligned}$$

because the sum of an empty list is 0. Make sense?

BOB: That's a mouthful, but yes, I see. $\text{Sum}(a :: b) = \text{Sum}(\text{Sum}(a) :: \text{Sum}(b))$. So I can apply `Sum` as I go along and get the same answer. That means that the Kind `Sum(d6 ** 100)` is equal what I get by doing

```
pgd> sum_of_4_rolls = Sum(d6 ** 4)
pgd> sum_of_100_rolls = sum_of_4_rolls      # initialize
pgd> for _ in range(24):                    # loop to successively update
...>     sum_of_100_rolls = Sum(sum_of_100_rolls * sum_of_4_rolls)
```

ALICE: I think so, but let's do a simpler example to make sure we understand it correctly. Suppose we are just summing 12 rolls. The values of `d6 ** 4` are lists with four numbers in $[1..6]$ like $\langle 1, 4, 3, 5 \rangle$, $\langle 3, 6, 5, 6 \rangle$, and $\langle 6, 2, 1, 1 \rangle$. Transforming by `Sum` adds these up giving values for `Sum(d6 ** 4)` like $\langle 13 \rangle$, $\langle 20 \rangle$, and $\langle 10 \rangle$ respectively. Your `sum_of_4_rolls` looks like

```
,---- 1/1296   ---- 4
|---- 4/1296   ---- 5
|---- 10/1296  ---- 6
|---- 20/1296  ---- 7
|---- 35/1296  ---- 8
|---- 56/1296  ---- 9
|---- 80/1296  ---- 10
|---- 104/1296 ---- 11
|---- 125/1296 ---- 12
|---- 140/1296 ---- 13
<> -+---- 146/1296 ---- 14
|---- 140/1296 ---- 15
|---- 125/1296 ---- 16
|---- 104/1296 ---- 17
|---- 80/1296   ---- 18
|---- 56/1296   ---- 19
|---- 35/1296   ---- 20
|---- 20/1296   ---- 21
|---- 10/1296   ---- 22
|---- 4/1296    ---- 23
`---- 1/1296    ---- 24
```

If we mix it with itself, `sum_of_4_rolls * sum_of_4_rolls` corresponds to rolling 4 dice once and then rolling 4 dice again and recording a pair of sums, with values like $\langle 13, 8 \rangle$ and so on. Then, `Sum(sum_of_4_rolls * sum_of_4_rolls)` adds those values up, giving us the sum of eight dice. And doing this yet again gives us

```
Sum(sum_of_4_rolls * sum_of_4_rolls * sum_of_4_rolls)
```

which is like rolling 4 dice three times, getting the sums for each set of 4, and then adding up those subtotals to get the total sum. This is the Kind of the sum of 12 rolls as we wanted and is the same as:

```
Sum( Sum(d6 ** 4) * Sum(d6 ** 4) * Sum(d6 ** 4) )
```

BOB: Excellent. So “monoidal statistics” like `Sum` are those that let you do this decomposition and compute the statistic in parallel. They could have called them “parallel statistics,” eh?

Looking at the predefined statistics in `frplib`, I see that `Min`, `Max`, and `Count` also have this property. I suppose that makes sense; after all,

$$\begin{aligned} \min(\langle 10, 20, 30 \rangle :: \langle 40, 50 \rangle) &= \text{Min}(\langle \text{Min}(\langle 10, 20, 30 \rangle), \text{Min}(\langle 40, 50 \rangle) \rangle) \\ \min(\langle 10, 20 \rangle :: \langle \rangle) &= \text{Min}(\langle \text{Min}(\langle 10, 20 \rangle), \text{Min}(\langle \rangle) \rangle), \end{aligned}$$

which looks just like the formula for `Sum` above. (We take the minimum of an empty list of numbers to be ∞ by convention.)

ALICE: Right, so we have basically the same formula $\text{Min}(a :: b) = \text{Min}(\text{Min}(a) :: \text{Min}(b))$.

BOB: So, we can get the Kind for the minimum of 12 rolls by

```
Min( Min(d6 ** 4) * Min(d6 ** 4) * Min(d6 ** 4) )
```

like before. That’s great, but what if I want to do something more complicated, like the mean of the rolls or the range (difference between max and min). Those statistics don’t have this property.

ALICE: True, but we can get them both from statistics that do. For instance, if you can find the Kind of the sum, you can transform that to get the mean with `sum_of_100_rolls ^ (__ / 100)`.

But let’s solve the problem more generally. Have you seen the `Fork` combinator in `frplib`?

BOB: Yes, it combines a bunch of statistics with common dimension into a big tuple containing all of their results. For example, $\text{Fork}(s_1, s_2)(x) = s_1(x) :: s_2(x)$ and $\text{Fork}(s_1, s_2, s_3)(x) = s_1(x) :: s_2(x) :: s_3(x)$.

ALICE: And notice that if the statistics you give to `Fork` are “monoidal statistics”, so is the statistic that it returns.

BOB: Because we can just apply our formula above to each component.

ALICE: Yes. So if you want to compute the range (max - min), apply our formula above with the statistic `min_max = Fork(Min, Max)` and then take the difference at the end. That is,

```
min_max(min_max(d6 ** 4) * min_max(d6 ** 4) * min_max(d6 ** 4)) ^ Diff
```

BOB: Beautiful! Complicated but beautiful.

ALICE: Yes, it’s a lot. The good news is that `frplib` can automate this in many cases, but that’s a story for another day.

BOB: My problem is solved, thanks.

ALICE: Well, I have a related problem. You know how much I enjoy playing poker.

BOB: You’re a shark!

ALICE: Well, I thought I might parlay that interest into a way to beat the Marketplace. I’m trying to create FRPs to model shuffling a deck of cards, by drawing one card at a time.

BOB: I see. The next card depends on which cards you’ve seen already. Sounds like a join.

ALICE: Exactly, but I found it a bit tricky to define. Can I show you? Fair warning, there’s some Python here.

BOB: I’m not really fluent in Python, but I’m guessing I can follow along.

ALICE: Absolutely you can, it should be clear, though I’ll explain any Python oddities. Let’s start with a standard deck of 52 cards. We’ll arbitrarily assign the cards numbers 1 through 52; we can be more specific later if needed. At the first stage, I need an FRP that selects each card with equal weight; that’s just

```
pgd> card1 = uniform(1, 2, ..., 52)
```

For the next card, I need a conditional Kind that picks uniformly among *all but the first card chosen*. I’ll use the `frplib` function `irange` that gives an *inclusive* range

of integers from its first to second arguments, with an option to exclude values in a set. This looks like

```
pgd> card2 = conditional_kind({
...>     (first_card,): uniform( irange(1,52, exclude={first_card}) )
...>     for first_card in irange(1,52)
...> })
```

For each card, an integer from 1..52, this uses the `uniform` factory to make an equally weighted kind on all the other cards.

BOB: And your code is building a mapping of key-value pairs for each value of `first_card`, where `(first_card,)` is the key and the Kind excluding `first_card` is the value.

ALICE: Right. A conditional Kind maps the values of one Kind to other Kinds of equal dimension. Now, I could continue like this all the way to `card52`, but I think I need to be more systematic. Here's what I tried, a function that returns a conditional Kind for a particular card draw:

```
def card(n):
    "Returns the conditional kind for the nth card drawn."
    if n == 1:
        return uniform(1, 2, ..., 52)                                # (1)

    def draw_kind(previous_cards):                                   # (2)
        next_cards = list( irange(1, 52, exclude=set(previous_cards)) ) # (3)
        return uniform(next_cards)                                   # (4)

    return conditional_kind(draw_kind)                               # (5)
```

In (1), if this is the first card, we need to start things off with no previous cards, so we return a conditional Kind of type $0 \rightarrow 1$, which is just an ordinary Kind. In (2), we receive the list of $n - 1$ and compute a Kind for the next card, so this is a conditional Kind of type $n - 1 \rightarrow n$ that we define as a Python function. In (3), we create the list of valid next cards, which just excludes all the previous cards. In (4), we use the `uniform` factory to produce a Kind with equal weight on all of these cards. And finally, in (5) we convert the function `draw_kind` to a conditional Kind object, using `conditional_kind`.

BOB: OK, there's some hairy stuff there, but I'm generally following. How do you use this?

ALICE: Well `card(1)` is the Kind after one drawn card, and in succession

```
pgd> card(1) >> card(2)
pgd> card(1) >> card(2) >> card(3)
pgd> card(1) >> card(2) >> card(3) >> card(4)
```

give the Kind of the shuffle after 2, 3, and 4 cards are picked, respectively. We could write a loop to do it for all cards

```
pgd> shuffle = card(1)
pgd> for n in irange(2, 52):
...>     shuffle = shuffle >> card(n)
```

Of course, that's impossibly slow because there are $52!$ different shuffles in the tree, another big number.

BOB: So, you're looking for a trick like what worked for my problem.

ALICE: A trick would be nice, but I want to *understand* my options for tackling this.

BOB: I have two thoughts. First, what questions are you trying to answer? In my case, it mattered that I was interested in the Sum, for example, which made it possible to reduce the complexity. If you want to predict your hand, say, then you don't need to draw all 52 cards.

Second, are you sure you need an exact answer?

ALICE: It's true that if I'm looking at what happens in my hand it's easier. Like if I've drawn five specific cards, and I want to know the chance of getting a fifth card, it's easier, but I still may have to deal with 20, 25, 30 cards.

BOB: What happens if you permute the labels? Does it matter if you observe cards 1, 5, 9, 13, and 17 (in a four player game with five cards each) versus cards 1, 2, 3, 4, and 5?

ALICE: Interesting. I think that's an important observation, and I want to come back to that. If I had cards 1-5 in my hand and were drawing the sixth, I could predict it like this, for example. Draw six cards and compute the Kind of the sixth card given the observation of the specific five cards in my hand.

```
pgd> my_hand = card(1) >> card(2) >> card(3) >> card(4) >> card(5)
pgd> next_card = (my_hand >> card(6) | (Proj[1:5] == (16,17,18,19,4)))[6]
```

```
pgd> next_card ~ Or(__ == 20, __ == 15)
```

BOB: Cool, you're assuming you got a particular four cards in a row and want to see whether you get a straight. You could do this for any cards in your hand or write a function that checks various combinations.

ALICE: Yes, that's useful. I do want to come back to the other idea you had. You were suggesting that I demo FRPs instead of computing the Kinds?

BOB: Right. The Kinds let you compute everything exactly, but if you know what question you want to answer, you can tailor an FRP to that and demo it.

Now that I think of it, that's not a bad approach to my problem earlier.

```
pgd> d6_frp = frp(d6)
pgd> FRP.sample( 10_000, Sum(d6_frp ** 100) )
```

It's not as fast or exact as what we came up with earlier, but pretty good.

ALICE: The key is that `frplib` does not have to compute the Kind of an FRP like `Sum(d6_frp ** 100)` until you ask for it. It just hooks output ports to input ports and pushes the button. I could do something similar:

```
def draw(n):
    "Returns a conditional FRP for the nth card drawn."
    return conditional_frp(card(n))
```

The `conditional_frp` turns every Kind in a conditional Kind into a *new* FRP of that Kind, which is what I need.

BOB: Then just do your loop with

```
pgd> deck = draw(1)
pgd> for n in irange(2, 52):
...>     deck = deck >> draw(n)
pgd> deck
```

to get the value of a random deck, or do `FRP.sample` to demo a bunch of them. It won't be fast or exact, but it will give you useful information.

ALICE: That's quite good; I can use that. But I've also been thinking about your comment on permutations.

The `Permute` statistic factory in `frplib` produces statistics that just rearrange the order of the list. For example, `Permute(3,2)` swaps the second and third components in a value list.

```
pgd> psi = Permute(3,2)
pgd> psi( (10,20,30) ) = (10, 30, 20)
```

Applying a permutation to the labels for our deck is just a relabeling of the cards, but we don't really care which number is assigned to which card.

Suppose I start with some Kind k on $n-1$ cards and compute the resulting Kind for n cards: `k >> cards(n)`. If do any permutation of the labels after this, it is equivalent to doing the *same* permutation on the values of k . That is, for any permutation "...", these two Kinds are equal

```
k >> cards(n) ^ Permute(...)      (k ^ Permute(...)) >> cards(n)
```

BOB: That's not obvious to me, but I'm trying it out in the playground and it does seem to work.

ALICE: Think of it this way. If I put new labels on *all* the cards after I've drawn $n-1$, then since all n^{th} cards have equal weight, it's the same as if we draw the n^{th} card before doing the relabeling.

BOB: Hmm. I think I've got it. And I see where you are going. Since the Kind of the shuffled deck is

```
card(1) >> card(2) >> card(3) >> ... >> card(51) >> card(52)
```

then if we apply a permutation at the end, we can just move it up through the `>>`'s.

```
card(1) >> card(2) >> card(3) >> ... >> card(51) >> card(52) ^ Permute(...)
card(1) >> card(2) >> card(3) >> ... >> (card(51) ^ Permute(...)) >> card(52)
...
card(1) >> card(2) >> (card(3) ^ Permute(...)) >> ... >> card(51) >> card(52)
card(1) >> (card(2) ^ Permute(...)) >> card(3) >> ... >> card(51) >> card(52)
(card(1) ^ Permute(...)) >> card(2) >> card(3) >> ... >> card(51) >> card(52)
```

All these are the same Kind!

ALICE: And here's the punchline. We know that `cards(1)` is just `uniform(1,2,...,52)`, so `cards(1)` does not change when you relabel the cards.

```
Kind.equal( cards(1) ^ Permute(...), cards(1) )
```

BOB: So, permutating the deck doesn't change the Kind, or your analysis!

So, if you want to consider only your hand and a few cards to draw from, you can use `cards(1) >> ... >> cards(8)`, which is more manageable.

In fact, if the cards in your hand are `c_1`, `c_2`, `c_3`, `c_4`, and `c_5`, you can just do

```
pgd> my_hand = (c_1, c_2, c_3, c_4, c_5)
pgd> constant(my_hand) >> cards(6) >> cards(7) >> cards(8)
```

ALICE: Nice. You used the fact that `my_hand` equals the Kind `also_my_hand` where

```
pgd> first_five = cards(1) >> cards(2) >> cards(3) >> cards(4) >> cards(5)
pgd> also_my_hand = first_five | (__ == (c_1, c_2, c_3, c_4, c_5))
```

That makes it easier to answer many interesting questions. Good team work!

BOB: Don't risk too much money at the table...

ALICE: Restraint is my middle name.

BOB: (*Rolls eyes affectionately*)

Puzzle 83. Suppose you are interested in when a specific pattern of die rolls – 4, 6, 2 – occurred during successive rolls at any point during 100 rolls of a six-sided die. Using the same `d6` that Bob did in this section, compute the Kind of the *event* that the pattern occurs, i.e., an FRP that outputs 1 if the pattern occurs and 0 otherwise.

For the next puzzles, we refer to the following example.

Example 7.1. A language is a set of strings made up of symbols from a fixed alphabet. Consider the language consisting of one or more `a`'s with a zero or one commas between each sequence of `a`'s. Strings "`a,aaa,a,aaa,a`" and "`a`" and "`aaaaaa,a,a,a`" belong to this language, but strings "`a,a`" and "`,a,`" and "`,,`" do not. We will describe this language by a graph whose nodes represent "states" and whose edges represent "transitions." The graph is shown in Figure 7.1.

We start in the blue S node – the "Start" state. We will process a string of `a`'s and `,`'s one character at a time, moving from state (node) to state (node).

Suppose we are in a particular state (node) at a given time. If the next unseen character in the string is an `a`, we follow the edge labeled `a` out of our node. If the next unseen character in the string is an `,`, we follow the edge labeled `,` out of our node. This determines the next state

After seeing "`a,aa`", for instance, we would be in state A; after "`a,a,`" in

If you are familiar with *regular expressions*, this language is described by `a+(,a+)*`.

state C; and after “a,” in state F. The green state A is “Accept” – *ending* there means that the input string belongs to the language, but ending in any other state means that it does not. The red state F is “Failure” – *reaching* that state automatically means the input does *not* belong to the language.

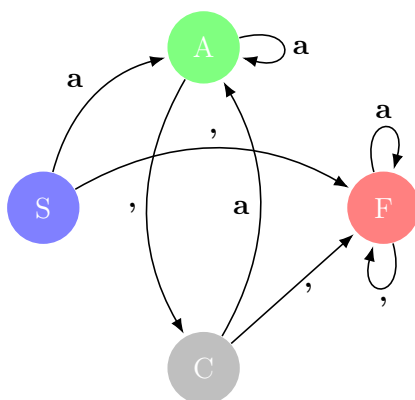


FIGURE 7.1. The language described in Example 7.1.

Puzzle 84. Referring to the situation in Example 7.1, assign the number 0 to the character `a` and the number 1 to the character `,` and the number 2 to an “end of string” marker, which can be repeated.

Define `char = uniform(0,1,2)`. What does `char ** 80` represent?

Write (in code or pseudo-code) a function that takes a value of `char ** 80` and returns the corresponding string.

Puzzle 85. We are interested in whether the string produced by `char ** 80` in the previous puzzle belongs to the language described by Example 7.1.

Assign the number 1 to the case where the string is accepted and 0 to the case where the string is not accepted. Compute the corresponding Kind.

You will want to construct an initial Kind and a conditional Kind for each move. Like Alice and Bob, you only need some information not the whole path.

7.2 A Dialogue on Systems and State

Carlos and Danielle work in the research division of the FRP Marketplace. This conversation took place when you met them during a tour.

CARLOS: The FRP Marketplace has a contract with the city of Uncertain, Texas to run all the local traffic lights, and our team is charged with devising new ways to

build dynamic, random systems like that.

DANIELLE: And even more complicated as well.

YOU: What do traffic lights have to do with FRPs?

DANIELLE: At each tick of the clock, you can think of a traffic light as a conditional FRP. It gets input that represents the current traffic at the intersection and returns an FRP that represents how the light's state will change at the next tick of the clock.

YOU: Change *state*? Like from liquid to gas?

CARLOS: Well, not exactly, but it's a similar idea. The **state** of a system is information that fully – and if possible, concisely – describes the internal configuration of a system at any given moment.

If you know a system's state, you can describe what you will observe from the system and how the state can change.

DANIELLE: It might help to start with a simple traffic light to make this clearer.

YOU: The traffic light moves from green () to yellow () to red () so the state of the system is the current color of the light?

DANIELLE: It can be in simple cases. However, to allow the light to behave the way we might want, it is useful to keep track of some extra information.

YOU: Ah, I see. We probably want the yellow light to be on for a shorter time than the green or red, for instance. We would need to know not only what color the light is but also how long it has been that color.

CARLOS: Exactly! We could, for instance, define the state of the traffic light to have the form $\langle c, n \rangle$ where c is the current color and n is the number of ticks of the clock for which the light has had color c . As examples, possible states are $\langle \text{green}, 0 \rangle$, $\langle \text{yellow}, 10 \rangle$, and $\langle \text{red}, 3 \rangle$.

DANIELLE: There's usually *not* just one right way to define a system's state. We *choose* how to define the state based on what we want the system to do and what information we need to update the state.

YOU: Update the state? You mean when the clock ticks and the state is $\langle \text{green}, 7 \rangle$, the state might change to $\langle \text{green}, 8 \rangle$ or to $\langle \text{yellow}, 0 \rangle$, depending on how long we want each color to show for.

CARLOS: Yes. We can specify a rule $\langle n_g, n_y, n_r \rangle$ so the light stays  for n_g ticks of the clock, then  for n_y ticks, and then  for n_r ticks.

DANIELLE: We would write the rule as a *function*:

$$\langle c, n \rangle \mapsto \langle c, n + 1 \rangle \{n < n_c\} + \langle \text{next}(c), 0 \rangle \{n = n_c\}$$

where $\text{next}(\text{red}) = \text{green}$, $\text{next}(\text{green}) = \text{yellow}$, and $\text{next}(\text{yellow}) = \text{red}$.

YOU: You are using indicator functions¹ here?

¹See Chapter 12 in Interlude F.

CARLOS: Yes, we use the indicators like $\{n < n_c\}$ to select among cases. As a named Python function, this would be

```
def update_state(state):
    color, ticks = state
    if ticks < rule[color]:
        return (color, ticks + 1)
    return (next(color), 0)
```

YOU: OK, I'm still getting used to indicators, but that makes sense. I do wonder though why you need FRPs for any of this.

DANIELLE: That's a good question. One answer is that other rules are possible, and you do not always *want* a color to have the same fixed duration. You might want to randomize the light a bit so people do not learn to jump the light. You might also want to incorporate data into a guess for whether the light should stay at its current color.

CARLOS: Another answer is that this is a very simple system, and we want to build a technology for describing this *and* other more complex systems.

Let's take the next step that Danielle is suggesting and randomize the update rule we just discussed to see how this would work. You will replace the update function with a conditional FRP that takes as input the value of an FRP representing the current state.

YOU: That's clearer, thanks. Let me give it a try in the playground. (*thinks*)

DANIELLE: Excellent idea. I've loaded some infrastructure to make this easier from `frplib.examples.traffic_light`. For instance, `TrafficLight.GREEN` stands for the  color, and `next(TrafficLight.GREEN)` gives you . These colors are encoded as integers 0, 1, and 2 for , , and .

YOU: Thanks. OK, here's code I might enter into the playground:

```
change_on = {
```



```

TrafficLight.GREEN: 1/30,
TrafficLight.YELLOW: 1/5,
TrafficLight.RED: 1/30,
}

@conditional_frp(codim=2, target_dim=2, auto_clone=True)
def tick_light(state):
    color, ticks = state
    change_probability = change_on[color]
    next_kind = weighted_as(
        (color, ticks + 1),
        (next(color), 0),
        weights=[1 - change_probability, change_probability]
    )
    return frp(next_kind)

start_any_color = uniform(change_on.keys()) ^ Fork(Id, 0)

```

The conditional FRP `tick_light` decides randomly at each tick whether to change the color. The dictionary `change_on` associates with each color the weight on changing color. The Kind `start_any_color` is one possible Kind for the initial state.

CARLOS: Great! Now, how would you use this to “run” the traffic light if I gave you an FRP S that represents the state of the traffic light at a particular moment.

YOU: I would compute a join $S \gg \text{tick\_light}$, but this would give me a 4-dimensional FRP that includes the state from S and the updated state. So I would apply `Proj[3,4]` to drop the old state.

Actually, now that I think of it, that’s what the conditioning operator $//$ is for. So I would define

```

def n_ticks(n, S):
    assert n >= 0, "Number of ticks must be non-negative."

    State = S
    for _ in range(n):
        State = tick_light // State # use clone(tick_light) without auto_clone

    return State

```

where I just successively update the state with a fresh clone of `tick_light`. (We could also use the built-in function `evolve` instead of the loop.) Then I could do, say

```
pgd> n_ticks(10, frp(start_any_color))
An FRP with value <2, 5>
pgd> ten_ticks = _      # _ is the last value
```

and I could pass this FRP to `n_ticks` to continue.

```
pgd> n_ticks(30, ten_ticks)
An FRP with value <0, 4>
```

DANIELLE: That's great. Having the values of the FRPs is nice, but to make predictions, we would like to be able to find the Kinds of these FRPs. Does that work?

YOU: Let's see. To understand this, it will help me to start from a known state where we are just starting a  cycle.

```
pgd> start_green = constant(TrafficLight.GREEN, 0)
pgd> StartGreen = frp(start_green)
pgd> kind( n_ticks(0, StartGreen) )
<> ----- 1 ---- <0, 0>
pgd> kind( n_ticks(1, StartGreen) )
,---- 29/30 ---- <0, 1>
<> -|
    `---- 1/30 ----- <1, 0>
pgd> kind( n_ticks(2, StartGreen) )
,---- 0.93444 ----- <0, 2>
|---- 0.032222 ----- <1, 0>
<> -|
    |---- 0.026667 ----- <1, 1>
    `---- 0.0066667 ---- <2, 0>
pgd> kind( n_ticks(3, StartGreen) )
,---- 0.00022222 ---- <0, 0>
|---- 0.90330 ----- <0, 3>
|---- 0.031148 ----- <1, 0>
<> -+---- 0.025778 ----- <1, 1>
    |---- 0.021333 ----- <1, 2>
```

```

|---- 0.011778 ----- <2, 0>
`---- 0.0064444 ----- <2, 1>
pgd> kind( n_ticks(100, StartGreen) )
...output omitted

```

As far as I can see, this is right. Each time, there is only a small probability of the light changing, and these accumulate so we get the chance of a yellow, then a red, then back to green. And on the second tick, we can see that the chance of a red is $1/30 \cdot 1/5$ with two changes in a row, as it should be. That last one has size 298, but glancing at the values, they make sense.

CARLOS: If you pay attention only to the colors, what do you see.

YOU: I can transform those Kinds with the projection statistic `Proj[1]` to get the Kind of the color.

```

pgd> kind( n_ticks(10, StartGreen) ^ Proj[1] )
,---- 0.72831 ---- 0
<> -+---- 0.12186 ---- 1
`---- 0.14983 ---- 2
pgd> kind( n_ticks(30, StartGreen) ^ Proj[1] )
,---- 0.51840 ----- 0
<> -+---- 0.091777 ---- 1
`---- 0.38983 ----- 2
pgd> kind( n_ticks(50, StartGreen) ^ Proj[1] )
,---- 0.47336 ----- 0
<> -+---- 0.080086 ---- 1
`---- 0.44655 ----- 2
pgd> kind( n_ticks(100, StartGreen) ^ Proj[1] )
,---- 0.46177 ----- 0
<> -+---- 0.076985 ---- 1
`---- 0.46124 ----- 2
pgd> kind( n_ticks(500, StartGreen) ^ Proj[1] )
,---- 6/13 ---- 0
<> -+---- 1/13 ---- 1
`---- 6/13 ---- 2

```

Interesting. We start at the beginning of a  cycle, so after a few ticks, we are much more likely to still be  because the probability of changing is so low. But as the number of ticks increases our predictions change, as though the system were

“forgetting” that started out as  . I can see with a little fiddling that above, say, 300 ticks the weights do not change to numerical precision.

If we wait long enough, we will predict  and  each 6/13 of the time and  1/13. We would expect  to be less likely as the system is more likely to switch out of yellow at any tick.

DANIELLE: So, you can make both short-term and long term predictions. If you were observing the traffic light from the street, you would see the color but not the full state. How would you compute your predictions in that case?

YOU: An interesting question. But before I answer that, something’s on my mind.

I’ve been computing the Kinds of FRPs here, and it works. But I realize I could also compute the Kinds and then generate FRPs from them. I assume both ways would give the same result.

DANIELLE: They would. How would you compute the Kinds?

YOU: Well, instead of a conditional FRP, I would use a conditional Kind, and in fact, I already did that. Let me refactor my code a bit:

```
@conditional_kind
def tick_light_kind(state):
    color, ticks = state
    change_probability = change_on[color]

    return weighted_as(
        (color, ticks + 1),
        (next(color), 0),
        weights=[1 - change_probability, change_probability]
    )

@conditional_frp
def tick_light(state):
    return frp(tick_light_kind(state))

def n_ticks_kind(n, initial_state):
    assert n >= 0, "Number of ticks must be non-negative."

    state = initial_state
    for _ in range(n):
```

```

    state = tick_light_kind // state

    return state

def n_ticks(n, InitialState):
    return frp(n_ticks_kind(n, kind(InitialState)))

```

As before, the conditional `Kind` either adds to `ticks` or takes the next `color` with corresponding probabilities.

It's very similar to the FRP version, but it makes it easier to work with the `Kinds` to make predictions. And now `n_ticks` can take a `Kind` or an FRP in the second argument.

CARLOS: If `tick_light_kind` were hard to compute, then your original approach to `n_ticks` would be more efficient. But here, it's good.

YOU: OK, Danielle, back to your question. If I observed only the color of the traffic light, I would need to apply a constraint with an observation to make my predictions.

DANIELLE: Right! Try it.

YOU: Using the more recent code, if I observed , I'd find the `Kind` of the state after, say, 30 ticks to be:

```

pgd> n_ticks_kind(30, start_any_color) | (Proj[1] == TrafficLight.GREEN)
,---- 0.047580 ---- <0, 0>
|---- 0.045730 ---- <0, 1>
|---- 0.043751 ---- <0, 2>
|---- 0.041604 ---- <0, 3>
|---- 0.039242 ---- <0, 4>
<> -+---- 0.036610 ---- <0, 5>
|---- 0.033640 ---- <0, 6>
|---- 0.030255 ---- <0, 7>
|---- 0.026360 ---- <0, 8>
|---- 0.021841 ---- <0, 9>
`---- 0.63339  ----- <0, 10>

```

CARLOS: Nice! Here's a challenge. Your conditional FRP makes the same decision no matter how long the light has been at its current color. Can you modify your work so that it uses `ticks` in a reasonable way?

YOU: This only requires changing `tick_light_kind`.

```

def stay_factor(ticks):
    return numeric_exp(-ticks / 2)  # must be >= 0

@conditional_kind
def tick_light_kind(state):  # only change is this factor //
    color, ticks = state      #                               vv
    change_probability = 1 - (1 - change_on[color]) * stay_factor(ticks)

    return weighted_as(
        (color, ticks + 1),
        (next(color), 0),
        weights=[1 - change_probability, change_probability]
    )

```

Now the change probability starts off as before when `ticks` is 0 but gets closer to 1 as `ticks` grows from. We can set the function `stay_factor` to any non-negative function.

CARLOS: That works, very good. It can be a bit hard to understand what the choice of `stay_factor` means for how long the light will stay one color. An alternative is to choose a Kind for how long the light will stay as the current color when the color changes and to change the state to hold that information.

YOU: So the state would become $\langle c, n, \ell \rangle$ where c and n are like before and ℓ is the number of ticks remaining until the color changes.

I would need a Kind for each color representing the time that the light stays that color. I'll make that a dictionary `duration` that maps colors to Kinds.

That would give something like this:

```

@conditional_kind
def tick_light_kind(state):
    color, ticks, remaining = state

    if remaining > 0:
        return constant(color, ticks + 1, remaining - 1)

    return duration[color] ^ Fork(next(color), 0, Id)

```

The `Fork` makes a Kind with the next color and 0 in the first two components and

the duration in the third. I would then have to modify the initial state `Kind` in the same way, but everything else should work as is.

CARLOS: Excellent. You can play with these ideas more with code in the `frplib.examples.traffic_light` module.

YOU: Thanks. That gives me a better idea of how these systems work. Danielle had mentioned, though, that you could make the lights respond to current conditions. How would that work?

DANIELLE: Systems have a *state* that describes their internal configuration, which is what you've been exploring here. In practice, we often also want them to respond to external *inputs* to produce *outputs* we can use.

YOU: What distinguishes inputs and outputs from state?

DANIELLE: The inputs can influence the state, and the outputs can be derived from the state. But in general, we think of input and output as separate parts of the system. Let me give an example.

Our traffic light is at an intersection in which cars are arriving, stopping, and passing through. Define two-dimensional FRPs

- $T^{(n)}$, representing the number of cars waiting in the direction of the light (component $T_1^{(n)}$) and in the other direction (component $T_2^{(n)}$), after n ticks of the clock.
- $M^{(n)}$, representing the number of cars who pass through the light in the direction that is   . (For simplicity, assume cautious drivers who stop on   .)

The $T^{(n)}$'s comprise a system, which we view as inputs to the light, and $M^{(n)}$'s comprise a system, which we view as outputs,

YOU: If the $T^{(n)}$'s are inputs, does the state just include the number of cars waiting in each direction.

CARLOS: It could, but to sharpen the ideas, assume that there are sensors on the traffic light that estimates the number of waiting cars but max out at ten cars.

Now our state looks like $\langle c, t, w_1, w_2 \rangle$ where w_1 and w_2 are the values recorded from the sensors in $[0 \dots 10]$.

DANIELLE: Let $L^{(n)}$ be the conditional FRP representing the traffic light after n ticks, which takes the number of waiting cars in each direction as input. Then we have a 6-dimensional FRP $S^{(n)} = T^{(n)} \triangleright L^{(n)}$.

The update of $T^{(n)}$ to $T^{(n+1)}$ depends on three things:

1. how many cars are waiting,

2. whether the green light stays on during tick $n + 1$, and
3. how many cars arrive during tick $n + 1$.

At the same time, we can design the traffic light to adjust its timing based on the estimated number of waiting cars that its sensors read. If w_1 is much bigger than w_2 , then the  will tend to remain on longer, and shorter in the reverse case.

YOU: I get it. The T and L systems are coupled; they depend on each other. So we can write a rule that transitions $S^{(n)}$ to $S^{(n+1)}$ along the lines you just described.

CARLOS: Right! And similarly, the output process $M^{(n+1)} = \psi(S^{(n)})$ for some statistic. (Note the $n + 1$ on the left and the n on the right.) The number of cars passing through depends on how many cars are waiting and which light, if any, is .

YOU: By taking appropriate joins and transforming with well-chosen statistics, we build the entire system. We just need to describe the Kinds of the traffic-light transitions, the number of cars that arrive in each direction, and the number of cars that get through during a given tick. And that breaks the problem into three separate things that are easier to understand and specify. Very nice!

DANIELLE: Exactly. There are some details, but you could code it up with what you already know. And `frplib` has some other tools to help, which I'm sure you'll encounter soon.

YOU: Fantastic. It occurs to me that we are making an assumption about these systems: that to find the next state we only need to know the current state but not the earlier history.

CARLOS: You're absolutely right about that. It is a huge assumption!

DANIELLE: It's possible to make systems whose evolution depends on their earlier history – even their whole history. Imagine if the next state is determined by a conditional Kind that uses not only the current state but the state before that and the state before that. Such systems can be useful, but the farther back in that history we have to go, the harder it is to do the analysis and computations.

CARLOS: For many practical problems, we can get what we need by making the assumption you mentioned.

DANIELLE: That assumption is called the **Markov property**. A system has the Markov property if its future evolution depends only on the current state but not on how it got there.

CARLOS: More formally, the Markov property means that if you know the current state, then your predictions about the system's future do not change if you also learn

its full history.

We say that *given* the current state, the future and past are independent.

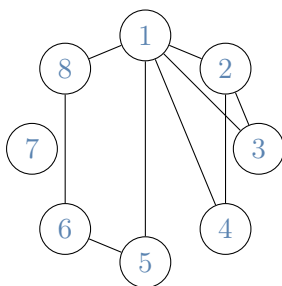
YOU: What’s an example, besides the traffic light, of a system with the Markov property?

DANIELLE: Are you familiar with graphs? The graphs with nodes and edges, I mean.

YOU: A little bit, yes.

CARLOS: A graph is a mathematical structure that describes pairwise relationships among several entities.² A graph has a set of nodes representing the entities and a set of edges representing the relationships between pairs of entities.

Here’s an example with eight nodes and various edges.



We call this graph *simple*, because there is at most one edge connecting any two nodes); *undirected* because the edges do not have a preferred direction; and *without loops* because there are no edges connecting a node to itself.

DANIELLE: Let’s build a system that describes a “random walk” on this graph. Our state will be the number of the current node, $\langle n \rangle$ for $n \in [1..8]$. If we are currently at node n , the Kind of the next state will be either `constant(n)` if node n has no edges connected to it or `uniform(neighbors(n))` where `neighbors(n)` lists all other nodes connected to n . We call each such update a **transition** or step, even if the state itself does not change.

How would you build a system like that?

YOU: I see that it’s not that different from what did with the traffic light. I’ll write everything in terms of the Kinds, and we can apply `frp` where needed to generate the FRPs.

The conditional Kind given that we are at a particular node n returns a Kind with equal weight on every node connected to the input node n by an edge.

For the random walk, we successively update the Kind of the current state (starting from the initial state) with the conditional Kind.

²The Random Graphs example in Chapter 2 shows various examples. See Interlude F Examples 10.18 and 10.19 for an overview and Interlude G for a detailed discussion.

```

@conditional_kind
def next_node(node):
    adjacent = neighbors(node)
    if len(adjacent) == 0:
        return constant(node)
    return uniform(adjacent)

def random_walk(start, n_steps=1):
    assert n_steps >= 0, "Number of steps must be >= 0."

    current = start
    for _ in range(n_steps):
        current = next_node // current

    return current

```

CARLOS: Yes, the similarity with the traffic light is not a coincidence. Your line `current = next_node // current` is just an expression of the Markov property. The next node only depends on the current state, not the history of how you got there.

Notice also that your `random_walk` function works equally well when `start` is a Kind or an FRP.

YOU: I get that this lets us describe the system, but what can we do with this.

CARLOS: We can answer questions about the system like: How likely is the system to move from one particular state to another? How long before the system visits a specific state? How many times is the system in a particular state during the first n steps? And many more.

YOU: I understand how to answer a question like “Will the system be in node 8 after 100 steps if it starts at node 1?” Just run `random_walk(constant(1), 100)` and look at the weight on branch $\langle 8 \rangle$.

But I do not see how to answer those other questions.

DANIELLE: That question opens some very interesting doors. You should go talk to Erin and Fuyuan over in Building Y. They specialize in those techniques, and I think you’ll enjoy a chat.

But in the meantime, let me illustrate a simple approach here. Consider two questions:

1. If we let the system run for a long time, does it settle down into an equilibrium that is independent of where it started, and if so, what state will it be in?
2. If the system starts at node 1, how long (if ever) until it reaches node 8?

For the first question, you will recall that the traffic light system you built did exactly that. After more than about 200 steps, the chance of being  or  or  did not change noticeably. Using your `random_walk` function, we can do:

```
pgd> random_walk(constant(1), 2500)
,---- 5/18 ---- 1
|---- 3/18 ---- 2
|---- 2/18 ---- 3
<> -+---- 2/18 ---- 4
|---- 2/18 ---- 5
|---- 2/18 ---- 6
`---- 2/18 ---- 8

pgd> random_walk(constant(3), 2500) # same for any node except 7
,---- 5/18 ---- 1
|---- 3/18 ---- 2
|---- 2/18 ---- 3
<> -+---- 2/18 ---- 4
|---- 2/18 ---- 5
|---- 2/18 ---- 6
`---- 2/18 ---- 8

pgd> random_walk(constant(7), 2500)
<> ----- 1 ---- 7
```

We can see that node 1 is a juncture between two parts of the graph, so we'd expect the system to spend more time there. And that's what we see. For any starting node except 7, the long-term behavior of the system is the same. Indeed, the numerator in those weights is just the number of neighbors of each node. For node 7, there's nowhere to go, so we fully understand the behavior if the system starts there.

For question 2, Erin and Fuyuan will show you a beautiful answer, and you can see from the answer to question 1 that we *will* eventually visit node 8 unless we start in node 7.

Still, if we modify the state of our system, we can get an output that answers our question. Our new state will have the form $\langle n, v, t \rangle$ where n is the current node as before, v is 1 if we have ever *visited* node 8 or 0 otherwise, and t is the number of

steps (the “time”) until we first visit node 8.

We can modify your code as follows:

```
@conditional_kind
def next_node(state):
    "Version of next_node that tracks whether we visited node 8."
    node, visit, time = state
    adjacent = neighbors(node)
    if len(adjacent) == 0:
        return constant(state)

    node_kind = uniform(adjacent)
    if visit == 1:
        return node_kind ^ Fork(Id, visit, time)
    if node == 8:
        return node_kind ^ Fork(Id, 1, time + 1)
    return uniform(adjacent) ^ Fork(Id, 0, time + 1)
```

All we do is keep track of whether we’ve visited 8 and either increment or freeze the time accordingly. Now, we can look at the Kind of the time component, using -1 to indicate that we have not visited node 8.

```
pgd> random_walk(constant(1, 0, 0), 360) ^ IfThenElse(Proj[2] == 1, Proj[3], -1)
... output of size 359 omitted
pgd> E(random_walk(constant(1, 0, 0), 360) ^ IfThenElse(Proj[2] == 1, Proj[3], -1))
23/2
```

The 360 here is arbitrary, long enough to be “long term” but short enough to be computed quickly. We want this as large as possible, infinite really, but in practice it does not need to be very large. Above 360, for instance, there is at most negligible change. Because we know in this case that the system will eventually visit node 8, the weight on $\langle -1 \rangle$ tells us how good our approximation is. Here, it’s about 4×10^{-15} , which is plenty close to zero. (Other techniques do not require even this accommodation.)

Applying the expectation operator E , we get our best prediction of how long until we visit node 8: 11.5 steps on average.³

YOU: That is very cool. So by augmenting the state, you can summarize many different aspects of the history and make predictions about them.

³Chapter 6 will discuss interpretation of this number in detail.

CARLOS: Right, we have a lot of flexibility to define the state in ways that let us answer interesting questions. And this graph example has broad application to lots of real problems.

YOU: I appreciate all your time here. I've got a lot to think about. I'll certainly wander over to Building U after the tour.

DANIELLE: You won't regret it. It is a bit hard to find, so wear comfortable shoes.

7.3 A Dialogue on Solutions and Simulation

After a long walk, you arrive in the basement of Building U. Walking through dank hallways with dripping pipes and flickering fluorescent lights, you discover a shiny, modern lab. Inside, you see Marketplace researchers Erin and Fuyuan, who look up at you quizzically.

ERIN: Do you have the pizza?

YOU: Pizza? No, I'm hoping to talk with you if you have a little time.

FUYUAN: That's another pizza delivery failure. We should put up signs.

YOU: Signs would help.

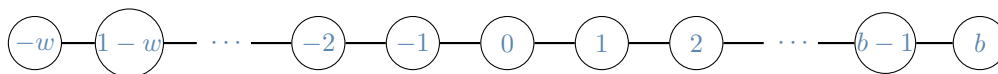
ERIN: Why are you here?

YOU: Carlos and Danielle recommended that I talk to you about techniques for answering questions about random systems.

FUYUAN: They showed you the random walk on graphs, didn't they?

YOU: Yes.

ERIN: *(moving toward a large display)* That's a good start. Think about this graph for some positive integers w and b :



FUYUAN: Imagine that a gambler comes to a casino with $\$w$ and places a series of identical, independent $\$1$ bets until either losing all her money or increasing her wealth by $\$b$. The state of the system – the node in the graph – is the *change* in the gambler's wealth. How likely is the gambler to be ruined – lose all her money – in this game?

YOU: If we had $w = b$ and if she had the same chance of winning and losing each bet, then she would be equally likely to be ruined and achieve her goal by the

symmetry between winning and losing. As b grows (shrinks) relative to w , her chance of achieving her goal should get smaller (larger), I'd guess.

ERIN: Good. We'll do that case first, but we want to handle cases where the bet's outcomes are not evenly weighted, like real casino games for instance.

FUYUAN: We can consider three broad approaches to this problem: (i) solve for an exact solution, (ii) solve for an approximate solution and make the approximation error small if possible, and (iii) simulate the system to estimate the solution. Let's start with (ii) and (iii) to set things up.

YOU: From my discussion with Danielle and Carlos, I think I know how to do that.

```
pgd> def gamble_with(wealth, goal):
...>     assert wealth >= 0 and goal >= 0
...>     bet = either(-1, 1)
...>
...>     @conditional_kind(codim=1, target_dim=1)
...>     def wealth_change(state):
...>         if state == -wealth or state == goal:
...>             return constant(state)
...>         return bet ^ (__ + state)
...>
...>     return wealth_change

pgd> def gamblers_walk(next_state, start=constant(0), n_steps=1):
...>     current = start
...>     for _ in range(n_steps):
...>         current = next_state // current
...>     return current
```

Here, I wrapped the conditional Kind in a factory function so that we could set the gambler's initial wealth and goal. If the system gets to state $-w$ or b , it stays there; otherwise, it goes up or down according to the output of the bet. The `gamblers_walk` function is like what I did earlier with the traffic lights, except I take the conditional Kind as an argument and by default start at state 0.

Then, I think I can do both (ii) and (iii):

```
pgd> K = gamblers_walk(gamble_with(10, 15), n_steps=3000)
pgd> C = frp(K) ^ Cases({-10: -1, 15: 1}, 0)
```

```

pgd> FRP.sample(10_000, C)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| -1     | 6043  | 60.43%    |
| 1      | 3957  | 39.57%    |
+-----+-----+-----+
pgd> K ^ Cases({-10: -1, 15: 1}, 0)
      ,---- 3/5 ---- -1
<> -+---- 0/5 ---- 0
      `---- 2/5 ---- 1

```

The `Cases` built-in statistic converts the original values to -1 for ruin, 1 for success, and 0 for unresolved. And we can see that after 3000 steps, there is a negligible chance that the Gambler has not achieved her goal or ruin.

ERIN: Both strategies (ii) and (iii) work well in this case as you say, and we can even guess at the *exact* solution here. When $w = b$, we get 1/2 each for goal and ruin; when $w = 10$ and $b = 15$, we get 2/5 and 3/5. Can you guess?

YOU: I think I have it, but let me try a few experiments

```

pgd> def ruin_or_goal(w, b, steps=3000):
...>     s = Cases({-w: -1, b: 1}, 0)
...>     return s(gamblers_walk(gamble_with(w, b), n_steps=steps))
pgd> ruin_or_goal(5, 15)
      ,---- 3/4 ---- -1
<> -+---- 0/4 ---- 0
      `---- 1/4 ---- 1
pgd> ruin_or_goal(5, 20)
      ,---- 4/5 ---- -1
<> -+---- 0/4 ---- 0
      `---- 1/5 ---- 1
pgd> ruin_or_goal(19, 1)
      ,---- 0.050000 ----- -1
<> -+---- 1.4481E-17 ---- 0
      `---- 0.95000 ----- 1

```

My guess is that the probabilities of ruin and goal are $\frac{b}{w+b}$ and $\frac{w}{w+b}$.

FUYUAN: That's right.

YOU: Shouldn't this approach always work.

FUYUAN: In a sense it does work, and these are very useful techniques. However in many cases, it requires many more runs to get a good approximation, and it can be harder to guess the exact answer.

Two examples are worth considering: bets where winning and losing do not have equal weight and trying to predict *how many bets* it takes before the gambler is ruined or achieves her goal.

YOU: The first is a simple change to `bet` and add a parameter:

```
pgd> def gamble_with(wealth, goal, win_to_lose=1): # <<- added parameter
...>     assert wealth >= 0 and goal >= 0
...>     bet = either(1, -1, win_to_lose) # <<- only change from earlier
...>
...>     @conditional_kind(codim=1)
...>     def wealth_change(state):
...>         if state == -wealth or state == goal:
...>             return constant(state)
...>         return bet ^ (__ + state)
...>
...>     return wealth_change
```

Then I'll add the win-to-lose ratio to `ruin_or_goal`, another simple change:

```
pgd> def ruin_or_goal(w, b, win_to_lose=1, steps=3000):
...>     s = Cases({-w: -1, b: 1}, 0)
...>     return s(gamblers_walk(gamble_with(w, b, win_to_lose), n_steps=steps))
```

And then run the simulation, for example with a win-to-lose weight ratio of 16/17:

```
pgd> ruin_or_goal(5, 20, '16/17')
,---- 0.90032 ----- -1
<> -+---- 7.3879E-12 ---- 0
    ^---- 0.099679 ----- 1
```

The approximation must be pretty good because the weight on 0 is so small, but I see what you mean though. Not easy to guess what those numbers mean.

ERIN: Nice. And for the second example?

YOU: We have to change the state to keep track of how many steps we have taken but stop counting when we hit either ruin or goal.

Is there a way to avoid rewriting these functions every time we change state?

ERIN: For the first, you are right; try it. For the second, yes to some degree, but it requires a little fancier programming. It's probably best if we don't go down that rabbit hole right now, but an example:

```
def markov_transition(start, next_state):
    def do_steps(n_steps=1):
        current = start
        for _ in range(n_steps):
            current = next_state // current
        return current

    return do_steps
```

Now, calling `markov_transition` with the Kind of the initial state and the conditional Kind of the transition will return a *function* that computes the Kind of the state after any number of steps.

YOU: Thanks, that's encouraging. I'm guessing that works for any system with the Markov property.

ERIN: Yes

YOU: OK, here's a version that tracks the time until ruin or goal.

```
pgd> def gamble_time(wealth, goal, win_to_lose=1):
...>     assert wealth >= 0 and goal >= 0
...>     bet = either(1, -1, win_to_lose)
...>
...>     @conditional_kind
...>     def state_change(state):
...>         delta_wealth, time = state
...>         if delta_wealth == -wealth or delta_wealth == goal:
...>             return constant(state)
...>         return bet ^ (__ + delta_wealth) ^ Fork(Id, time + 1)
...>
...>     return state_change

pgd> go = markov_transition(constant(0,0), gamble_time(10, 15))
pgd> t3000 = Proj[2](go(3000))
```

Oh, that took a little while.

ERIN: There are many paths, so there is a small chance of it taking a while. It can help to use `clean` to eliminate the negligible branches or use `E` to get a simpler prediction.

```
pgd> E(t3000)
149.9999999725631
```

which we can guess should be 150 if exact. This is our best prediction about how long it will take the gambler to reach ruin or goal.

FUYUAN: All this is prologue to looking at strategy (i). It's good to see that we can still compute good answers without an exact solution but it takes a little work and judgment.

To find an exact solution in this problem, we are going to solve *several similar problems simultaneously*. This will give us an equation that we can solve exactly.

ERIN: It will help to see it in action. We'll start with the gambler's ruin problem and a couple other concrete cases, but the equation we derive will work for a huge variety of problems, as we will see.

FUYUAN: We will write $\text{GamblersRuin}\langle w, b, r \rangle$ to denote the gambler's ruin problem with initial wealth w , goal b , and win-to-lose ratio r .

Our goal is to predict the number of bets it takes before the gambler gets to ruin or her goal. At the start, her net change in wealth is 0, i.e., she starts in state 0. But to find this prediction starting at 0, we will consider the versions of the same problem *for every starting node*.

Let T_s be an FRP representing the number of bets ("time") it takes before the gambler gets to ruin or goal when her starting state is $s \in [-w..b]$. We loosely call the number of bets "time" because it's easier and makes sense. Our best prediction of T_s 's value is $\mathbb{E}(T_s)$.

ERIN: Define a function f on $[-w..b]$ by

$$f(s) = \mathbb{E}(T_s). \quad (7.1)$$

We really want to find $f(0)$, but to do that we will solve for the *whole function*.

What do we know about the function f ?

YOU: I suppose that if we start at ruin or goal, we don't have to place any more bets. So, $f(-w) = 0 = f(b)$.

FUYUAN: Good, yes. Now we come to the key idea: what does the problem look like *after one step*.

YOU: What does the problem look like??

FUYUAN: If $-w < s < b$, what states can we reach after one step?

YOU: Either $s - 1$ or $s + 1$ with weights 1 and r .

ERIN: And from either of those states, what is the expected time until goal or ruin. Remember the Markov property.

YOU: Hmm. Once we are in state $s - 1$, our expected time is $f(s - 1)$ and once we are in $s + 1$, our expected time is $f(s + 1)$.

FUYUAN: And it took one step to get there. So what is the *Kind* of the expected time when you start in state s ?

YOU: (*excited*) Wait. We start in state s , and the state we move to is represented by an FRP, call it X_1 . The function $1 + f$ is solving our problem but it's just a function, so we can use it as a *statistic* and compute the Kind of $1 + f(X_1)$. Damn.

This gives us the Kind, in canonical form:

$$\langle \rangle \left[\begin{array}{l} \frac{1}{1+r} \text{ — } \langle 1 + f(s - 1) \rangle \\ \frac{r}{1+r} \text{ — } \langle 1 + f(s + 1) \rangle \end{array} \right] \quad (7.2)$$

ERIN: And this has expectation⁴

$$(1 + f(s - 1)) \frac{1}{1+r} + (1 + f(s + 1)) \frac{r}{1+r} = 1 + \frac{f(s - 1) + rf(s + 1)}{1+r}.$$

FUYUAN: This is our best prediction of the time it takes to reach ruin or goal, but that is also the definition of $f(s)$. So ***the two must be equal!*** That is, for every $s \in (-w \dots b)$ we have⁵

$$f(s) = 1 + \frac{f(s - 1) + rf(s + 1)}{1+r} \quad (7.3)$$

with $f(-w) = 0 = f(b)$.

This gives us $w + b - 1$ equations in $w + b - 1$ unknowns, so we can solve for f and then just read off $f(0)$, the answer to our original problem.

YOU: Whoa.

ERIN and FUYUAN: (*nods*)

⁴A general formula is given Chapter 6, but take this on faith for now.

⁵ $(a \dots b)$ is the set of integers from a to b excluding a and b . See Section 9.2 in Interlude F.

ERIN: For instance, if $r = 1$ (winning and losing equally likely), this can be re-written as the system of equations:

$$f(s+1) - 2f(s) + f(s-1) = -2 \quad \text{for } s \in (-w \dots b) \quad (7.4)$$

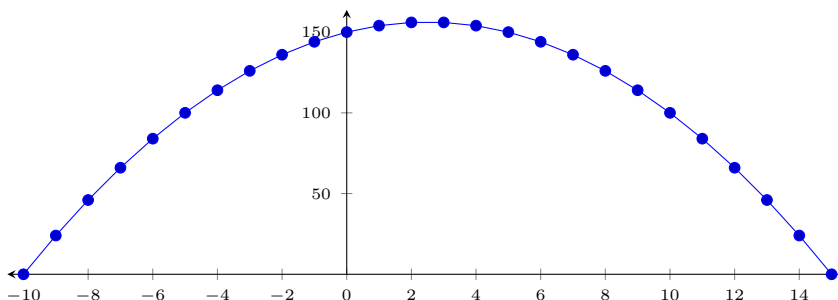
$$f(b) = 0 \quad (7.5)$$

$$f(-w) = 0. \quad (7.6)$$

The expression on the left side of the first equation is called the “second difference” of f , $\Delta^2 f$, at $s - 1$. Differences are a discrete analogue of derivatives,⁶ so we have a “second-order difference equation” for f with values at the boundary specified. We can solve this difference equation to find

$$f(s) = (b-s)(w+s). \quad (7.7)$$

So, in general $f(0) = bw$, and when $b = 15$ and $w = 10$, f looks like



with $f(0) = 150$ as you found earlier.

YOU: That is very cool. I do wonder how broadly applicable this approach is. For instance, earlier we guessed the probability that the gambler is ruined. Could we find it exactly?

FUYUAN: As with anything, there are trade-offs between what we must assume and the tractability of the problem. But the approach is quite general.

Suppose we have a system with the Markov property that visits a finite set of states. Whenever the system is in state s , the Kind of the next state is K_s , and these Kinds can vary from state to state.

Now imagine there are two sets of states $\mathcal{S}_0$ and $\mathcal{S}_1$, and we want to know the chance that the system visits a state in $\mathcal{S}_1$ before any state in $\mathcal{S}_0$.

YOU: So in the $\text{GamblersRuin}\langle w, b, r \rangle$ problem, we can take $\mathcal{S}_1 = \{-w\}$ and $\mathcal{S}_0 = \{b\}$ and the Kind K_s is

⁶The discrete calculus is developed in the “Sequences and Streams” examples and puzzles in Chapter 10 in Interlude F.

$$\langle \rangle \rightarrow \begin{cases} \frac{1}{1+r} \longrightarrow \langle s-1 \rangle \\ \frac{r}{1+r} \longrightarrow \langle s+1 \rangle \end{cases}$$

FUYUAN: Exactly. And similarly to what we did earlier, define $f(s)$ to be the probability of visiting $\mathcal{S}_1$ before $\mathcal{S}_0$ *when we start the system in state s* . Then, using exactly the same logic as earlier, we have

$$\begin{aligned} f(s) &= 0 && \text{if } s \in \mathcal{S}_0 \\ f(s) &= 1 && \text{if } s \in \mathcal{S}_1 \\ f(s) &= \mathbb{E}(f(K_s)) && \text{if } s \notin \mathcal{S}_0 \cup \mathcal{S}_1, \end{aligned} \quad (7.8)$$

where $\mathbb{E}(f(K_s))$ is our best prediction of the value of an FRP with Kind $f(K_s)$.

YOU: But we don't know f .

ERIN: True, but we can write the transformed Kind $f(K_s)$ in terms of f 's value, just like you did in (7.2). And we get one equation and one unknown for each state not in $\mathcal{S}_0 \cup \mathcal{S}_1$.

YOU: So if we try this with $\text{GamblersRuin}\langle w, b, 1 \rangle$ we get

$$\begin{aligned} f(b) &= 0 \\ f(-w) &= 1 \\ f(s) &= \frac{1}{2}f(s-1) + \frac{1}{2}f(s+1) \quad \text{otherwise,} \end{aligned}$$

so

$$f(s) - f(s-1) = f(s+1) - f(s).$$

The differences of f are constant, meaning that f must shrink *linearly* from value 1 at $-w$ to value 0 at b . I think this means

$$f(s) = \frac{b-s}{b+w} \quad (7.9)$$

giving

$$f(0) = \frac{b}{b+w} \quad (7.10)$$

as we guessed earlier.

ERIN: Good. Now, imagine you are in a room that looks like the one in Figure 7.2. You start on a tile in the middle of the room (marked green) and move North, South, East, or West randomly, with equal weights. The red tiles are lava – you don't want


```

pgd> from frplib.examples.dirichlet import *
pgd> f = solve_dirichlet_sparse(lava_room.cKind, states=lava_room.states,
                               fixed=lava_room.fixed, fixed_values=(0, 1))
pgd> f(0, 0)          # starting at (0,0)
0.8178943428510974

```

Use $f(s) = 1 + \mathbb{E}(f(K_s))$ to get the expected time to hit either lava or water:

```

pgd> end_tiles = lava_room.fixed[0].union(lava_room.fixed[1])
pgd> f_time = solve_dirichlet_sparse(lava_room.cKind, alpha=1, states=lava_room.states,
                                     fixed=[end_tiles], fixed_values=[0])
pgd> f_time(0, 0)  # starting at (0, 0)
66.5864958884486

```

ERIN: Here's a puzzle for you to try later. (We have other things to talk about now.)

Puzzle 86. Theseus is trapped in a labyrinth, as is his wont, and a fierce creature is trapped with him. Both start at distinct locations, and each moves randomly from their current juncture with equal weight on all available directions.

If Theseus and the creature ever meet, Theseus will be eaten. But Theseus (not the creature) can escape through a small opening at one specific juncture.

Create a small labyrinth and compute the probability that Theseus escapes.

YOU: Create a labyrinth?

ERIN: A labyrinth is just a graph. Each juncture is a node, and each path from a juncture to its neighbor is an edge.

In fact, the graphs we are working here are actually *directed* graphs with loops. The Kinds K_s can be different for each state, so it is possible that we can get from state s to s' in one step but not s' to s . It is also possible to stay in the same state, which is an edge from s to s . Each branch of the Kind K_s is associated with one (directed) edge that leaves node s in the graph.

We can actually solve a somewhat more general system than Fuyuan showed in (7.8). We have a set of states $\mathcal{S}$ (nodes in our graph) that is partitioned into disjoint sets $\mathcal{S}_0, \mathcal{S}_1, \dots, \mathcal{S}_m$ for some m . We have a function f on $\mathcal{S}$ whose value on each $\mathcal{S}_i$ is

known to be c_i for $i < m$ and that satisfies

$$\begin{aligned} f(s) &= c_i && \text{if } s \in \mathcal{S}_i \text{ for } i \in [0 \dots m) \\ f(s) &= \alpha + \beta \mathbb{E}(f(K_s)) && \text{if } s \in \mathcal{S}_m, \end{aligned} \quad (7.11)$$

for fixed numbers α, β .

Setting $m = 2$, $c_0 = 0$, $c_1 = 1$, $\alpha = 0$, and $\beta = 1$, gives our prediction of whether the system hits $\mathcal{S}_1$ before $\mathcal{S}_0$. Setting $m = 1$, $c_0 = 0$, $\alpha = 1$, and $\beta = 1$, we get our predicted time until the system hits $\mathcal{S}_0$. And so on. Our `solve_dirichlet` will handle all those problems.

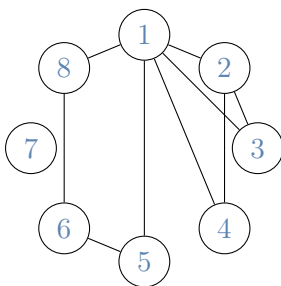
YOU: That's useful to know, thanks. I do have two questions. First, this lets us compute specific predictions (expectations), but can I use this to find the Kind itself. Second, Carlos and Danielle showed me a case where the system seemed to come to an equilibrium, but there's no "hitting" any set of states. Can we handle that?

FUYUAN: Good questions. The general answer to both is yes, but to keep things concise-ish, let's tackle an example of both at once.

Notice that if the system can get *stuck* in two distinct states, like the gambler's ruin, then it can't really get to an equilibrium. It can get stuck in one or the other, we're not sure which. But we've solved problems like that, so let's consider cases where there is a long-run equilibrium.

What was the graph that Carlos and Danielle showed you?

YOU: It was this



FUYUAN: OK, that's a good candidate, but let's ignore node 7. You can't reach it from the others and if you start there, you stay there, which is an equilibrium but a trivial one.

ERIN: Like the Hotel California

FUYUAN: *(smiles)* Imagine we have a system that describes a random walk on this graph. What does it mean for the system to be in "equilibrium"?

YOU: It means that at some level things don't change. Of course, they *change* because if you are moving around the graph, but . . .

ERIN: That's the idea. Think of it this way: when you look at the system at some arbitrary time, the current node is random, and it has some Kind K . Equilibrium means that that Kind doesn't change. The system might move among the nodes, but if the system is in equilibrium, your *prediction* about what node it will be after 100, 1000, 10000 steps will be *the same*.

YOU: That makes sense.

FUYUAN: So we do the same logic that led to our equations (7.11): we think about what happens in a *single step*.

Let S be the conditional Kind of the next node given the current node. If an FRP representing the current node has Kind K , then an FRP representing the next node has Kind $S \parallel K$, and the assumption that the system is in equilibrium means that

$$K = S \parallel K. \quad (7.12)$$

We just solve this equation for K .

We say that K is a **fixed point** of the function $\langle k \rangle \mapsto S \parallel k$, because if we evaluate this function at K , it returns K right back. The function “keeps K fixed.”

ERIN: We can usually find this K by iterating. Start with an arbitrary Kind, like `constant(0)`, and do the following

```
pgd> k0 = constant(0)
pgd> k0 = S // k0
pgd> k0 = S // k0
pgd> . . .
```

or just do it in a loop. This `k0` should convert to K .

But the graph has finitely many nodes, so we can solve for K 's weights directly.

YOU: Solve?

FUYUAN: Sure. We know the values of K ; what we don't know are the weights. Write out equation (7.12) in terms of those weights.

YOU: I see. So, to find the weight on node 1, say, in $S \parallel K$, we need to add up the

contribution from all nodes leading to 1.

$$\begin{aligned}
 w_1 &= \frac{1}{3}w_2 + \frac{1}{2}w_3 + \frac{1}{2}w_4 + \frac{1}{2}w_5 + \frac{1}{2}w_8 \\
 w_2 &= \frac{1}{5}w_1 + \frac{1}{2}w_3 + \frac{1}{2}w_4 \\
 w_3 &= \frac{1}{5}w_1 + \frac{1}{2}w_2 \\
 w_4 &= \frac{1}{5}w_1 + \frac{1}{2}w_2 \\
 w_5 &= \frac{1}{5}w_1 + \frac{1}{2}w_6 \\
 w_6 &= \frac{1}{2}w_5 + \frac{1}{2}w_8 \\
 w_8 &= \frac{1}{5}w_1 + \frac{1}{2}w_6.
 \end{aligned}$$

For instance, to get to 1, we must choose one neighbor out of three from node 2, and one out of two from each of nodes 3, 4, 5, and 8. We can solve these equations, yielding

$$\begin{array}{c}
 \begin{array}{c} \langle \rangle \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \frac{5}{18} \\ \frac{3}{18} \\ \frac{2}{18} \\ \frac{2}{18} \\ \frac{2}{18} \\ \frac{2}{18} \\ \frac{2}{18} \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} \langle 1 \rangle \\ \langle 2 \rangle \\ \langle 3 \rangle \\ \langle 4 \rangle \\ \langle 5 \rangle \\ \langle 6 \rangle \\ \langle 8 \rangle \end{array}
 \end{array}$$

as we saw before.

Let me do this in the playground to check my ideas. The Kind factory `prenormalized` is like `weighted_as` but assumes that weights with symbolic terms add up to 1.

```

pgd> S = conditional_kind({
  1: uniform(2, 3, 4, 5, 8),
  2: uniform(1, 3, 4),
  3: uniform(1, 2),
  4: uniform(1, 2),
  5: uniform(1, 6),
  6: uniform(5, 8),

```

```

            8: uniform(1, 6)
...> })
pgd> w = symbols('w1 w2 w3 w4 w5 w6 w8')
pgd> K = prenormalized(1, 2, ..., 6, 8, weights=w)
pgd> S // K
...output omitted

```

Yes, the same equations. Good.

ERIN: Great work!

YOU: If you cannot find an exact solution and have to resort to your strategies (ii) and (iii) that we discussed earlier, do you have ways to make those more efficient.

ERIN: There is not one general technique, but there are ways to speed things up a lot. The Markov property comes in handy. Here's a nice example.

We have an encrypted English text consisting of n characters $c_1 c_2 \cdots c_n$ that we would like to decrypt. Assume two things for simplicity: (i) the text consists only of the 26 letters A-Z and spaces, and (ii) the encryption is with a *substitution cipher* that permutes the 27 characters and substitutes the true characters with the character permuted to that position. For example, if ABC is permuted as CAB, then every A in the true text will appear as C in the cipher text, every B as A, and every C as B. Our decryption d is thus be a permutation of the same 27 characters; we want the permutation that *inverts* the cipher. So, $d(A) = B$, $d(B) = C$, $d(C) = A$ would invert CAB, replacing every A in the cipher text with a B, every B with a C, and every C with an A.

YOU: The problem is that there are $27! = 10888869450418352160768000000$ possible d 's. That's a lot. Searching them all is practically impossible. Even searching enough to have a chance of finding the right one would take far too long.

FUYUAN: You're right, and that's why we need a bit of cleverness in our simulation. We've computed the frequency of character pairs $b(c, c')$ in a reference corpus⁷ of English text, where we use special characters **start** and **end** to represent the beginning or end of the text.

⁷See comments in the code for citations.

Using those frequencies, we give each d a *score*:

$$L(d) = b(\text{start}, d(c_1)) \prod_{i=2}^n b(d(c_{i-1}), d(c_i)) b(d(c_n), \text{end}).$$

We want to find the d that (at least approximately) *maximizes* the score L .

YOU: That helps you pick good or bad d 's, but don't you still have to search untold numbers of orderings of the characters?

ERIN: We create a system with the Markov property much like your random walk on a graph. Like the random walk, which settled into an equilibrium Kind with weights proportional to the number of neighbors a node has, this system will settle into an equilibrium proportional to the score $L(d)$.

As the system runs, we keep track of the state (d) with the largest score. As the system approaches equilibrium, that gets closer to the maximizer.

YOU: Can you be more specific about how you set up this system?

FUYUAN: The algorithm is simple. First, we choose a starting state, such as the d that is the identity permutation. Initialize the maximum score to $s = L(d)$ and best decryption to $m = d$.

Then we repeat the following steps for as many iterations as we can:

1. If the current state is d with score $L(d)$, generate an FRP that randomly swaps two positions in d (e.g., ABC to CBA). Call the resulting permutation d' .
2. Compute $L(d')$ and let $p = \min(1, L(d')/L(d))$.
3. Generate an *event*⁸ with Kind

$$\langle \rangle \begin{cases} \xrightarrow{1-p} \langle 0 \rangle \\ \xrightarrow{p} \langle 1 \rangle \end{cases}$$

⁸Recall that an event is just an FRP with possible values 0 or 1.

4. If that event occurs, the system moves to state d' ; otherwise it stays in state d .
5. Set s to $\max(s, L(d'))$. If s increases, set m to d' .

YOU: So boiled down to the details, we randomly swap letters. If that increases the score, we move to that state; if not, we *might* move to that state, with a move more likely the bigger its score.

ERIN: That's all there is to it. Do you want to see it in action?

```
pgd> from frplib.examples.markov_decrypt import markov_decrypt, cipher1, clear1
pgd> cipher1
QXYVE WMAVDOCBJVLCKVP TGYFVCHYOVQXYVRUSNVFCI
pgd> decrypted, *_ = markov_decrypt(cipher1, iter=10_000)
pgd> decrypted
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG
pgd> decrypted == clear1
True
```

You can look at the code, too. It's pretty simple.

YOU: That was surprisingly fast, and it worked.

ERIN: This technique works for a range of ciphers and it generalizes to many different kinds of problems.

YOU: Amazing. I really appreciate your time. The possibilities are exciting.

PIZZA PERSON: (*knocks on lab door*) Did someone order a pizza?

FUYUAN: Finally!

Chapter Checkpoints

After reading this Chapter you should be able to:

- Identify situations where FRP sizes grow quickly.
- Explain the useful property of a “monoidal statistic”.
- Show how to compute a transformed Kind with a monoidal statistic quickly even when the original Kind has very large size.
- Use conditional Kinds/FRPs and joins to describe steps in a process.
- For some large Kinds, find an efficient way to answer targeted questions.
- Explain what the *state* of a system means.
- Define the state for some simple systems and use joins and statistics to update the state as the system evolves.
- Explain in words what the Markov property means.
- Simulate a system with the Markov property and if possible, compute the Kind of the state after some number of steps.
- Find simulated and approximate predictions for the time to hit some set of states or whether one set of states will be hit before the other.
- Explain how to construct an equation for an exact solution to such problems.
- Explain how to find the Kind of the state for a system in equilibrium.
- Describe the idea behind the Markov decryption simulation.

Patterns, Predictions, and Practice

8

Chapter

Contents

8.1	Mind the Types	478
8.2	A Pattern Language for Random Systems	482
8.3	Simple Finite Random Processes	488
8.4	Strategies and Representations	505
8.5	Using Observations	514
8.6	Touching Infinity	521

This Chapter offers a synthesis of the ideas of Part I in two dimensions. First, it starts with a perspective on the connection between our description of a random system and our analysis of that system, via the Big 3+1. We can see this as a type of “pattern language” that we can learn to recognize and exploit to tackle more complicated problems. We consider the types underlying the operations of the Big 3+1 and study the parallel between operations on FRPs and on their Kinds.

Second, the Chapter uses `frplib` to tackle a wide variety of example problems that combine the methods we have seen so far. The focus here is on identifying the Big 3+1 operations – transformations, joins, observations, and expectations – and studying how they are combined to solve problems in our “pattern language.” All of the complex analyses we do and will do are built from these basic operations, and the commentary on the examples will attempt to highlight this.

The later sections of the Chapter are loosely divided into sections emphasizing particular techniques. In the example `frplib` code for this section, the prompts (like `pgd>` and `...>`) are sometimes omitted from the display to save space unless there is output to show. You can load each example’s code into the playground using the name in lower case with no punctuation and `_` instead of space, but only up through the first three words, e.g., “`from frplib.examples.six_of_one import *`”. You are encouraged to follow along with the examples in the playground as you read.

8.1 Mind the Types

Every operation we perform acts on entities of specific **types** and produces a result of a specific type. We take the union of sets, like $\{1, 2, 4\}$ and $\{4, 7, 11\}$, not of individual integers, like 4 and 11, and the union of sets always produces a set. In contrast, we can do arithmetic on numbers but not sets,¹ and arithmetic with numbers produces a number. The function `log` accepts only positive numbers but can produce any real number. This is a familiar idea in programming as well. We can index a list or a string (e.g., `x[4]`) but not a floating-point number. Indexing a list produces a value with the type of elements of that list, while indexing a string produces a character.

We can think of types in several useful ways. At its simplest, we define a **type** to be a collection of objects that satisfies specified properties or have some specified structure. This collection will often be a set, but it can also be something larger than a set can hold.² For example, the type “natural numbers” might correspond to the set $[0..)$, and indeed we can name the type $\mathbb{N}$ using the name of that set, though in some contexts we might use a descriptive name like `Natural` instead. Alternatively, we can define a type through the ways we can construct objects of that type, the operations we can perform on such objects, and the laws that those objects must satisfy. For example, the type of natural numbers can be specified inductively by saying that 0 is a natural number and for any natural number n , its successor `succ( $n$ )` (i.e., $n + 1$) is also a natural number.

If x is an object and T is a type, we write $x : T$ to denote that x has type T .

We regularly use various *primitive types*, including natural numbers $\mathbb{N} = [0..)$, integers $\mathbb{Z}$, real numbers $\mathbb{R}$, Booleans $\mathbb{B} = \{\perp, \top\}$, and the *unit type* $\mathbb{U} = \{\langle\rangle\}$, which contains only a single object.³ When convenient, we might also call these by alternate names: `Natural`, `Integer`, `Real`, `Bool`, and `Unit`, respectively.

As described in Interlude F, we define the *function type* $A \rightarrow B$ to be the class of functions that take input of type A and return output of type B . We write $f : A \rightarrow B$ to indicate that function f has type $A \rightarrow B$. A function $c : \mathbb{U} \rightarrow B$ from the unit type to some other type B is for all practical purposes equivalent⁴ to the object $c(\langle\rangle)$ of type B , and we treat them interchangeably.

A binary operator `op` is a function of two arguments, which would give it type `op : A × B → C`, but it is often clearer to write this type in “infix” form as $A \text{ op } B \rightarrow C$ to indicate that the operator takes data of type A on the left and type B on the right and produces a result of type C .

For probability theory, including in `frplib`, we start with four *basic types*:

¹Although operations like Union and Cartesian Product often behave algebraically similarly to familiar arithmetic operations, they are not the same.

²Mathematicians call these *proper classes*.

³See Chapter 9 for more on all these sets.

⁴We call this an isomorphism. See Example 14.3 in Interlude F.

- A **Quantity** is a number or a symbolic value representing a number. (We include the special value **nothing** in this type as well.)
- A **Value** is a tuple whose components are of type **Quantity**. We elide the distinction between a tuple of dimension 1 $\langle s \rangle$ and the scalar quantity s it contains. Booleans are represented as scalars with 0 for false ($\perp$) and 1 for true ($\top$).
- A **Kind** is a tree in canonical form with weights that are positive quantities and leaf nodes of type **Value**, where all leaves are distinct and have the same dimension.
- An **FRP** is a device that produces a single **Value** when activated, a value that is fixed for all time.

For the last three types, we add a subscript (like Value_d) to restrict the type to objects of dimension d . The type Value_0 is just an alias for the unit type $\mathbb{U}$.

We have defined three special types of functions that we use throughout. A statistic is a function that takes and returns a value, i.e., tuple of numbers. A conditional FRP is a function that takes a value and returns an FRP. And a conditional Kind is a function that takes a value and returns a Kind. We can represent these broad classes of functions with three corresponding types:

$\text{Statistic} : \text{Value} \rightarrow \text{Value}$
 $\text{ConditionalFRP} : \text{Value} \rightarrow \text{FRP}$
 $\text{ConditionalKind} : \text{Value} \rightarrow \text{Kind}$

In practice, it is helpful to use more precise versions of these types that specify for each its codimension c and dimension d :

$\text{Statistic}_{c,d} : \text{Value}_c \rightarrow \text{Value}_d$
 $\text{ConditionalFRP}_{c,d} : \text{Value}_c \rightarrow \text{FRP}_d$
 $\text{ConditionalKind}_{c,d} : \text{Value}_c \rightarrow \text{Kind}_d$

Note that the types of particular, statistics, conditional FRPs, and conditional Kinds may be more specific still⁵ if they are only defined on strict subsets of Value_c . Also, we define the type **Condition** as the sub-type of **Statistic** representing statistics that returning Boolean values (or in **frplib** $\langle 0 \rangle$ for $\perp$ and $\langle 1 \rangle$ for $\top$).

Recall our observation that a **Kind** is just a conditional Kind of codimension 0 and similarly an **FRP** is just a conditional FRP of codimension 0. This relates to the comment earlier that a function from the unit type Value_0 to some other type b is

⁵As discussed in Chapter 10 of Interlude F, a function's domain is part of its type, and two functions with different types are not the same, even if they agree on the common parts of their domains.

equivalent to an object of type b . We can see this isomorphism directly in `frplib`. The following function takes a Kind `k` and returns a conditional Kind of codimension 0 whose target equals `k`:

```
def k_to_ck(k : Kind) -> ConditionalKind:
    @conditional_kind(codim=0, dim=dim(k))
    def ck():
        return k

    return ck
```

Conversely, the following function takes a conditional Kind `ck` of codimension 0 and returns its single target Kind:

```
def ck_to_k(ck : ConditionalKind) -> Kind:
    return ck()
```

These two functions are inverses of each other, e.g., `Kind.equal(k, ck_to_k(k_to_ck(k)))` returns `True` and similarly in the other direction. So we treat `Kindd` as equivalent to the type `ConditionalKind0,d` and `FRPd` as equivalent to the type `ConditionalFRP0,d`.

With these basic types in hand, we can look at the types of our Big 3+1 operations.

- *Transforming with Statistics*

$$\begin{aligned} \text{FRP}_m \wedge \text{Statistic}_{m,n} &\longrightarrow \text{FRP}_n \\ \text{Kind}_m \wedge \text{Statistic}_{m,n} &\longrightarrow \text{Kind}_n \\ \text{Statistic}_{m,n}(\text{FRP}_m) &\longrightarrow \text{FRP}_n \\ \text{Statistic}_{m,n}(\text{Kind}_m) &\longrightarrow \text{Kind}_n \\ \text{Statistic}_{m,n} @ \text{FRP}_m &\longrightarrow \text{FRP}_n \\ \text{Statistic}_{m,n} @ \text{Kind}_m &\longrightarrow \text{Kind}_n \end{aligned}$$

All three pairs here have the same meaning, transforming an FRP or Kind with a statistic into a new FRP or Kind. Although we write `stat(k)` or `stat(X)` for a statistic `stat` and a Kind `k` or FRP `X`, think of this as an *operator* (equivalent to $\wedge$), *not* as evaluating the statistic with an argument. A statistic does **not** take a Kind or FRP as input, only a `Value`. The `@` operator is a version of transformation particular to `frplib` that makes it easier to use with observational constraints, as discussed on page 348.

- *Building with Joins*

$$\begin{aligned}
&\text{ConditionalFRP}_{m,n} \triangleright \text{ConditionalFRP}_{n,r} \rightarrow \text{ConditionalFRP}_{m,r} \\
&\text{ConditionalKind}_{m,n} \triangleright \text{ConditionalKind}_{n,r} \rightarrow \text{ConditionalKind}_{m,r} \\
&\text{FRP}_m \star \text{FRP}_n \rightarrow \text{FRP}_{mn} \\
&\text{Kind}_m \star \text{Kind}_n \rightarrow \text{Kind}_{mn} \\
&\text{FRP}_m \star \star \text{Natural}_n \rightarrow \text{FRP}_{m^n} \\
&\text{Kind}_m \star \star \text{Natural}_n \rightarrow \text{Kind}_{m^n} \\
&\text{ConditionalFRP}_{m,n} \parallel \text{FRP}_m \rightarrow \text{FRP}_n \\
&\text{ConditionalKind}_{m,n} \parallel \text{Kind}_m \rightarrow \text{Kind}_n
\end{aligned}$$

The first pair describes general joins and the remaining pairs give the specialized variants: independent joins, independent join powers, and conditioning. In the independent join powers, we use the subscript in Natural_n to name the natural number n for use later in the type. Recall that the conditioning operator ($\parallel$, or $//$ in `frplib`) is just general join followed by projection that drops the input, as can be seen in the dimension of the resulting FRP/Kind. The argument order is reversed from that of general join because it makes the conditional FRP/Kind the focus of attention. Note that as a special case, if we have two conditional FRPs or conditional Kinds with the *same domain*, we can take their independent join, which gives a new conditional FRP/Kind on that domain that returns the independent join of the targets.

$$\begin{aligned}
&\text{ConditionalFRP}_{m,n} \star \text{ConditionalFRP}_{m,r} \rightarrow \text{ConditionalFRP}_{m,nr} \\
&\text{ConditionalKind}_{m,n} \star \text{ConditionalKind}_{m,r} \rightarrow \text{ConditionalKind}_{m,nr}
\end{aligned}$$

- *Constraining with Observations*

$$\begin{aligned}
&\text{FRP}_m \mid \text{Condition}_m \rightarrow \text{FRP}_m \\
&\text{Kind}_m \mid \text{Condition}_m \rightarrow \text{Kind}_m
\end{aligned}$$

- *Predicting with Expectations*

$$\begin{aligned}
&\mathbb{E}: \text{FRP}_m \rightarrow \text{Value}_m \\
&\mathbb{E}: \text{Kind}_m \rightarrow \text{Value}_m \\
&\mathbb{E}: \text{ConditionalFRP}_{m,n} \rightarrow (\text{Value}_m \rightarrow \text{Value}_n) \\
&\mathbb{E}: \text{ConditionalKind}_{m,n} \rightarrow (\text{Value}_m \rightarrow \text{Value}_n)
\end{aligned}$$

The last two types mean that the expectation of a conditional FRP or conditional Kind returns a *function* that takes the same values as input to the conditional FRP/Kind and returns the expectation of the corresponding target.

In addition to the operations on (conditional) FRPs and Kinds, we also have a variety of combinators that produce new statistics.

```
Statistic(Statistic) → Statistic
Statistic ^ Statistic → Statistic
Statistic + Statistic → Statistic
Statistic + Value → Statistic
...
```

and similarly with other arithmetic and comparison operators
(e.g., -, *, /, **, %, ==, <=).

```
Statisticm // Kindm → Valuem
```

The first pair represents the composition of statistics, in syntactic order ($\phi \circ \psi$, “ ϕ after ψ ”) and pipeline order ($\psi \circ \phi$, “ ψ then ϕ ”).⁶ The last item averages the value of a statistic with the weights and values of a Kind, with $\psi // k = \mathbb{E}(\psi(k))$. This makes $\mathbb{E}(z // k) = \mathbb{E}(z) // k$, where z is a conditional Kind.

⁶See Section 13.1 in Interlude F.

With these types in mind, we can consider more fully the relationships among different FRPs and Kinds, and among their conditional counterparts.

Key Take Aways

All of the operations in the Big 3+1 conform to specific *types*, both the types of their inputs and the type of their outputs. Keeping these requirements in mind makes it easier to remember how to use the operations and recognize when the operations are appropriate.

We start with the base types of quantities, *values*, Kinds, and FRPs and build from there.

8.2 A Pattern Language for Random Systems

Explicate the pattern language with examples.

We illustrate all the examples with arrows and `frplib`/mathematical expressions in the pattern language for each one, on both the FRP level and the Kind level.

Specific examples should include an isomorph of the supplier-quality hw problem, cases where we construct $(X \mid Y)$ and $(Y \mid X)$ and Z like above, and more

In the book *A Pattern Language: Towns, Buildings, Construction*,⁷ the architect Christopher Alexander and co-authors described 253 “patterns” that relate to particular problems in architectural design. Their patterns are modular and composable and so combine into a sort of *language* for describing solutions to design problems. This idea has been influential in other areas, including software engineering, where the idea of *design patterns*⁸ give a language for describing software-design problems. Taking inspiration from these ideas, we can see that the Big 3+1 operations – along with various operations derived from them – provide us with a collection of modular and composable patterns that arise in the building and analysis of random systems. The Big 3+1 thus comprise a domain-specific *language* that can be used to concisely describe both the structure of random systems and the computations that we use to analyze those systems.

⁷Alexander et al. 1977

⁸Gamma et al. 1994

Hard problems inevitably require some creativity to solve. Part of the challenge is using the technology we have to make the easy parts of a problem easy and to isolate the parts that demand creativity so we can focus our energies accordingly. Our random system pattern language concisely describes the structure of random systems and specifies the computations that we use to analyze those systems. This helps us recognize common patterns and subproblems that we know how to solve.

The key idea is one we have encountered in each previous chapter: each of the Big 3+1 operations acts *consistently* on both FRPs and Kinds. For instance:

$$\text{kind}(\psi(X)) = \psi(\text{kind}(X)) \quad (8.1)$$

$$\text{kind}(X \star Y) = \text{kind}(X) \star \text{kind}(Y) \quad (8.2)$$

$$\text{kind}(X \triangleright S) = \text{kind}(X) \triangleright \text{kind}(S) \quad (8.3)$$

$$\text{kind}(X \mid \zeta) = \text{kind}(X) \mid \zeta \quad (8.4)$$

$$\mathbb{E}(X) = \mathbb{E}(\text{kind}(X)). \quad (8.5)$$

(The third equation requires a clean join, which is the typical case.) This means that our description of the system and our analysis are parallel to each other.

What “description of the system” mean refers back to our framework in Figure 2.8. We build the Data FRP from the Element FRPs with the Big 3 operations and then in turn derive the Feature FRPs of interest from that with additional operations.

The Kind of both the Data FRP and the Feature FRPs can be expressed in terms of the Kinds of the Element FRPs.

For example, a system evolves for several stages, we then observe some partial information about the system's output at that stage, then the system evolves further. We want to make a prediction about the answer to a question at the end of that evolution. In our pattern language, the Data FRP looks like

$$D = ((X_0 \triangleright T_1 \triangleright T_2 \triangleright \cdots \triangleright T_n) \mid \zeta) \triangleright T_{n+1} \cdots \triangleright T_{n+m},$$

where each of the transitional conditional FRPs T_i are independent. Then, the Kind of the Data FRP is

$$\text{kind}(D) = ((k_0 \triangleright t_1 \triangleright t_2 \triangleright \cdots \triangleright t_n) \mid \zeta) \triangleright t_{n+1} \cdots \triangleright t_{n+m},$$

by the equations above, where $k_0 = \text{kind}(X_0)$ and $t_i = \text{kind}(T_i)$. This describes everything that we know about the Data FRP's value, and from this we can analyze D and any feature FRP. A feature FRP has the form $\psi(D \mid \xi)$: we potentially make additional observations (ξ) about the output of the Data FRP and then extract the answer to a question (ψ). Our knowledge of this Feature FRP is

$$\text{kind}(\psi(D \mid \xi)) = \psi(\text{kind}(D) \mid \xi).$$

Hence, the description of the system – that is, the Data FRP from which we will derive all our results – tells us how to do the analysis.

A related pattern is one where we do not keep the whole history but just the current state of the system at any stage. This looks like

$$\begin{aligned} S_0 &= X_0 \\ S_{i+1} &= T_{i+1} \parallel S_i \end{aligned}$$

for $i \in [0 \dots n)$. Then, we make an observation and observe additional stages

$$\begin{aligned} \tilde{S}_n &= S_n \mid \zeta \\ \tilde{S}_{i+1} &= T_{i+1} \parallel \tilde{S}_i \end{aligned}$$

for $i \in [n \dots n+m)$. And again our description begets the analysis. Let $k_i = \text{kind}(S_i)$

for $i \in [0 \dots n]$ and $\tilde{k}_i = \text{kind}(\tilde{S}_i)$ for $i \in [n \dots n + m]$. Then

$$k_{i+1} = \mathbf{t}_{i+1} // k_i$$

for $i \in [0 \dots n)$. Then, we make an observation and observe additional stages

$$\begin{aligned}\tilde{k}_n &= k_n \mid \zeta \\ \tilde{k}_{i+1} &= \mathbf{t}_{i+1} // \tilde{k}_i,\end{aligned}$$

for $i \in [n \dots n + m)$. This gives us an exact recipe for computing the Kind of the Data FRP and thus for analyzing any question that we have about the system.

We can go on like this, building systems (the Data FRP) and seeing how our analysis (based on the Kind of the Data FRP) explicitly parallels the system's description. The Big 3+1 and its operations and laws give us the terms in our language, and we learn to recognize and use common patterns that occur.

Let's supplement this abstract discussion with a concrete example. Many other examples follow in the remainder of this chapter, and many more fill the previous chapters as well.

Example 8.1. A robot starts at a random point in the plane with integer coordinates within distance 16 of the origin. At each stage it moves uniformly at random north, south, east, or west, except that it will not move in a direction if the move will take it more than 16 units from the origin.

After 100 steps, we observe that its position is $\langle x_{100}, y_{100} \rangle$ and then it takes 10 more steps before stopping. We would like to predict its final position.

Note first that this situation matches exactly the previously described system. We will start by defining the pieces specifically in `frplib`, but rather than settle for raw computation, we will use our recognition of the pattern – and the insight it gives us – to simplify our analysis. We will use the terms defined above S_i , T_i , k_i , $\tilde{k}_i$, and so forth.

```
@flexible_inputs
def is_in_range(pos):
    x, y = pos
    return x * x + y * y <= 16 * 16

# Possible positions for the robot
```

```

positions = [tup(x, y) for x in irange(-16, 16)
              for y in irange(-16, 16)
              if is_in_range(x, y)]

# Kind of the initial state  $S_0 = X_0$ 
k_0 = uniform(positions)

# Conditional Kind of one transition
@conditional_kind(codim=2, target_dim=2)
def t(pos):
    # Move N, S, E, or W unless we move outside the circle
    return uniform(new_pos for delta in [(0, 1), (0, -1), (1, 0), (-1, 0)]
                  if is_in_range(new_pos := pos + tup(delta)))

# The observation is the condition that the state is exactly
# the observed position  $\langle x_0, y_0 \rangle$ 
@condition_factory
@flexible_inputs
def zeta(observed_pos):
    return (__ == observed_pos)

# We want to predict the value of the Data FRP itself in this case
psi = identity

```

If we were to do the calculations brute force, we would do the following, mimicking the pattern given just before the example. We express this as a function to allow $\langle x_{100}, y_{100} \rangle$ to be specified.

```

def brute_force(x_0, y_0):
    # The Kind of the initial state (robot position) of the system
    state = k_0

    # We update the state for 100 steps
    for _ in range(100):
        state = t // state

    # Constrain with the observation

```

```

state = state | zeta(x_0, y_0)

# Now make 10 more steps
for _ in range(10):
    state = t // state

# The Kind of the final position given our observation
return state

```

The code writes itself because it exactly parallels our description of the system.

But we can do better! Recognizing in this common update pattern at each stage, we see what Danielle and Carlos called the **Markov Property** in Chapter 7: once we know where the robot *is* at a particular stage, its movement in the future does not depend on how it got there.

This means that once we observe the robot's position after 100 steps, the future evolution of the system does not depend on the robot's initial path to that position. We can do the analysis as though the robot *started there*. Constraining with the observation eliminates all other paths from consideration. This gives us a much easier calculation.

```

def with_insight(x_0, y_0):
    # We can start our analysis at the known position
    # by the Markov property
    state = constant(x_0, y_0)

    # We update the state for just 10 steps
    for _ in range(10):
        state = t // state

    # The Kind of the final position given our observation
    return state

```

Running both pieces of code will show the same Kind resulting, but the latter be almost instantaneous.

8.3 Simple Finite Random Processes

In this Section, we look at random processes with a fixed, finite number of stages. Sometimes these stages interact, sometimes they do not. As you consider how to model these examples, look for the parts of the process that are easier to understand in isolation. If those parts require some knowledge of earlier stages, that's OK – it's why we have joins. That suggests defining a conditional Kind/FRP. If we can model a part of the process independently of any other stage, then we can combine it with other parts with an independent join. Throughout, we think hard about how to represent the quantities we are modeling. This sometimes involves assigning meaning to arbitrary numbers, and it sometimes requires us to think about how we describe the state of the system. Statistics are useful for building an initial state and for transforming between different representations.

Example 8.2 Doubled Cards

A deck of 100 cards is labeled $1, 2, \dots, 100$. You choose two cards from the deck in succession. The deck is well shuffled, so you can assume that every pair of cards has equal weight to be selected.

Define two FRPs:

- D is the event that the second card's number is exactly twice the first card's.
- T is the event that *either* card's number is twice the other.

Find the expectations $\mathbb{E}(D)$ and $\mathbb{E}(T)$.

First, because D and T are both events, their expectations are the probabilities that the events occur. If D occurs, then by definition, T occurs as well, so the value produced by D is always $\leq$ the value produced by T . We write this as $D \leq T$, and by the ordering property $\mathbb{E}(D) \leq \mathbb{E}(T)$. The event T is more likely to occur.

Second, we can recognize a join structure here: we first draw a card from the deck and then draw a second card from the deck that is missing the first card. At both stages, all cards remaining in the deck are selected with equal weight. We define `draw` as the Kind of the join FRP that represents the pair of card numbers drawn from the deck:

```
first_card = uniform(1, 2, ..., 100)
all_cards = first_card.values
```

```

second_card = conditional_kind({
    first: uniform(all_cards - {first}) for first in all_cards
})
draw = first_card >> second_card
P = frp(draw)

```

Here, `first_card.values` is the *set* of possible values for the first card and `{first}` is the singleton set with just the “current” card; the difference removes the latter from the former. The Kind `draw` has dimension 2 and size 9900; its values are pairs of distinct card numbers. Because `first_card` and each Kind in `second_card` are uniform, the join Kind has the same weight, $1/9900$, on every branch. You can see this by looking at `draw` in the playground, though the large size makes that less than convenient.

`P` is the FRP representing the drawn pair of cards, and D and T are transforms P by two statistics:

```

is_card_doubled = Proj[2] == 2 * Proj[1]
is_either_doubled = Or(is_card_doubled, Proj[1] == 2 * Proj[2])

D = is_card_doubled(P)
T = is_either_doubled(P)

```

The first just checks if the second component of a value is equal to twice the first component, and the second checks that and in the reverse direction.

The Kinds D and T are transforms of `draw` by these statistics:

```

D_kind = is_card_doubled(draw)      # same as kind(D)
T_kind = is_either_doubled(draw)    # same as kind(T)

```

Think about how we find $\text{kind}(D)$ by transforming `draw`. Values with doubled cards map to $\langle 1 \rangle$; the rest map to $\langle 0 \rangle$. The weights in each branch of $\text{kind}(D)$ add up the weights for all branches of `draw` with the corresponding value. Because all the weights of `draw` equal $1/9900$, the weights are $(9900 - b_1)/9900$ and $b_1/9900$ where b_1 is the number of `draw`’s branches for which `is_card_doubled` equals $\langle 1 \rangle$. As D is an event, $\mathbb{E}(D)$ equals the weight on $\langle 1 \rangle$ in $\text{kind}(D)$. A similar approach works for T . Looking at the results:

```
pgd> D_kind
```

```

,---- 0.99495 ----- 0
<> -|
      `---- 0.0050505 ---- 1
pgd> E(D_kind)      # == E(D)
1/198
pgd> E(T_kind)      # == E(T)
1/99
pgd> FRP.sample(10_000, D)
+-----+-----+-----+
| Values | Count | Proportion |
+=====+=====+=====+
| 0      | 9946 | 99.46% |
| 1      | 54  | 0.54% |
+-----+-----+-----+

```

So, for 50 of the possible pairs, `is_card_doubled` is true and for 100 of the pairs, `is_either_doubled` is true. We can reason that if the first card is bigger than 50, the second card cannot double it, and if it is in $[1..50]$ only 1 out of the 99 remaining cards that will make D occur. T occurs for these 50 possibilities and also for the 50 pairs the second card is in $[1..50]$ and the first card doubles the first. The probabilities of these events are thus $1/198$ and $1/99$, and the demo of FRPs with Kind matching D is close to that proportion.

Our questions are answered, but it is worth looking at a few playground variations. First, we defined the conditional Kind `second_card` using a dictionary, but we could also use a function:

```

second_card = conditional_kind(
    lambda first: uniform(all_cards - {first}),
    codim=1
)

```

The `lambda first` defines a Python anonymous function that takes a single argument `first`, a value of `first_card`. The `codim=1` argument to `conditional_kind` tells it to expect a function of a *scalar* argument, so `first` can be a number.

Second, applying our reasoning *in advance*, we could define the conditional Kind `second_card` by

```

conditional_kind(

```

```

    lambda first: either(0, 1, 98) if c <= 50 else constant(0),
    codim=1
)

```

Finally, there is a built-in factory in the playground that creates `draw` directly

```
ordered_samples(2, irange(1, 100))
```

This example illustrates a common pattern: the transformed join. We describe a system in stages, because the stages are usually simpler to describe and specify. We use joins to combine the stages into the outcome of the process. And then we extract an answer to our question from the full outcome through transformation by a statistic. (See Figure 2.8.)

Example 8.3 Hunter's Success

Eight hunters each get one shot at a target moving quickly through the woods. All the hunters are “in the zone,” and so their shots are unaffected by the actions of their friends. The hunters have different levels of experience and skill, so for $i \in [0..8)$, the event H_i that hunter i hits the target has Kind

$$\langle \rangle \begin{cases} 1 - h_i & \langle 0 \rangle \\ h_i & \langle 1 \rangle \end{cases}$$

Let the FRP H represent the number of hunters who hit the target. **Find the Kind of H and $\mathbb{E}(H)$, your prediction of the number who hit the target. Also find $\mathbb{E}(\{H > c\})$ for each $c \in \{0, 2, 4\}$, the probability that more than c hunters get a hit.**

Our solution to this problem takes as input the probabilities h_i for $i \in [0..8)$. We put these in a list `h`, where for $i \in [0..8)$, `h[i]` holds the probability h_i . With that, we define a list of Kinds,

```
hits = [binary(h_i) for h_i in h]
```

where `hits[i]` equals `kind( $H_i$ )`. Since all the hunters take independent shots, the number of hits is just the sum of the independent join of these:

```
all_hits = independent_join(hits)
number_of_hits = Sum(all_hits)
```

The `independent_join` function returns the independent join of all the Kinds in `hits`, which equal to `hits[0] * hits[1] * hits[2] * hits[3] * hits[4] * hits[5] * hits[6] * hits[7]`. Then, `number_of_hits` is the Kind of the FRP H , representing the number of hunters who hit the target.

With `kind(H)` in hand, we can answer our questions:

```
E(number_of_hits)
E(number_of_hits ^ (__ > 4))
E(number_of_hits ^ (__ > 2))
E(number_of_hits ^ (__ > 0))
```

This gives the expected number of hits and the probabilities that the number of hits exceeds the specified threshold.

Here, we will show two approaches. In the first, we pick specific numeric weights for each hunter and solve the problem for that setting of the weights. In the second, we allow the weights to be variables (symbolic quantities) and solve the problem generally, obtaining answers from this solution for various settings of the weights. Once we set the weights in the list `h`, the solutions follow the steps above.

APPROACH #1 SPECIFIC WEIGHTS. We start by assuming one expert hunter, one good hunter, and six novices.

```
h = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.7, 0.9]
```

With this, we form the Kind `number_of_hits`, and compute the expectations of various transforms to answer our questions:

```
pgd> number_of_hits
,---- 0.015943 ----- 0
|---- 0.19132 ----- 1
|---- 0.45822 ----- 2
|---- 0.25710 ----- 3
<> -+---- 0.066995 ----- 4
|---- 0.0096001 ----- 5
|---- 0.00078384 ----- 6
|---- 0.000034360 ----- 7
`---- 6.3000E-7 ----- 8
```

```

pgd> E(number_of_hits)
2.2
pgd> E(number_of_hits ^ (__ > 4))
0.0104
pgd> E(number_of_hits ^ (__ > 2))
0.3345
pgd> E(number_of_hits ^ (__ > 0))
0.9841

```

The probability of at least one hunter getting a hit is very high; the probability of more than half getting a hit is very low. We predict 2.2 hits, though the probability of getting at least that many hits is substantially less than 1/2. This happens because there is a large probability of getting 1, 2, or 3 hits.

APPROACH #2 SYMBOLIC WEIGHTS. Next, we will mimic this analysis but more generally, letting h contain an unspecified symbolic quantity for each hunter’s skill level. We can then define multiple different “profiles” of hunters’ skills (e.g., all the same, one expert) and solve our problem for each profile.

```

h = symbols('h_0 ... h_7')

all_50_50 = substitute_with(dict(h_0=0.5, h_1=0.5, h_2=0.5, h_3=0.5,
                                h_4=0.5, h_5=0.5, h_6=0.5, h_7=0.5))
one_expert = substitute_with(dict(h_0=0.1, h_1=0.1, h_2=0.1, h_3=0.1,
                                h_4=0.1, h_5=0.1, h_6=0.1, h_7=0.9))
expert_plus = substitute_with(dict(h_0=0.1, h_1=0.1, h_2=0.1, h_3=0.1,
                                h_4=0.1, h_5=0.1, h_6=0.7, h_7=0.9))
some_elders = substitute_with(dict(h_0=0.1, h_1=0.1, h_2=0.1, h_3=0.1,
                                h_4=0.1, h_5=0.75, h_6=0.75, h_7=0.75))
random_skill = substitute_with({str(w): random() for w in h})

p = symbol('p')
all_equal = substitute_with(dict(h_0=p, h_1=p, h_2=p, h_3=p,
                                h_4=p, h_5=p, h_6=p, h_7=p))

```

The elements of h are “symbols” $h_0, h_1, \dots, h_7$ which can have numbers substituted for them after we do the calculations. The other variables like

`all_equal` hold *functions* that represent hunter profiles; each function substitute the specific numeric values for the symbols in an expression.

Now, we form the Kind `number_of_hits` as above, and we can compute the answers to our questions symbolically, then substitute each profile after the fact.

```
exp_hits = E(number_of_hits)
gt4_hits = E(number_of_hits ^ (__ > 4))
gt2_hits = E(number_of_hits ^ (__ > 2))
gt0_hits = E(number_of_hits ^ (__ > 0))
```

The first of these gives us particular insight; it shows us

```
pgd> exp_hits
h_0 + h_1 + h_2 + h_3 + h_4 + h_5 + h_6 + h_7.
```

The expected number of hits is just the sum of the probabilities that each hunter hits the target. This holds for every profile by the Additivity property of expectations (6.4) and because the expectation of an independent join is the tuple of the component expectations, equation (6.8):

$$\begin{aligned}\mathbb{E}(H) &= \mathbb{E}(\text{Sum}(H_0 \star H_1 \star \cdots \star H_7)) \\ &= \text{Sum}(\mathbb{E}(H_0 \star H_1 \star \cdots \star H_7)) \\ &= \text{Sum}(\langle \mathbb{E}(H_0), \mathbb{E}(H_1), \dots, \mathbb{E}(H_7) \rangle) \\ &= \mathbb{E}(H_0) + \mathbb{E}(H_1) + \cdots + \mathbb{E}(H_7).\end{aligned}$$

The expressions for `gt4_hits` and so forth are much more complicated and harder to parse. But we can recover what we did about by substituting the values from the profiles defined earlier. For example, the profile `expert_plus` matches what we used in Approach #1.

```
pgd> expert_plus(exp_hits)
11/5
pgd> expert_plus(gt4_hits)
0.01041895
pgd> expert_plus(gt2_hits)
0.33451777
pgd> expert_plus(gt0_hits)
0.98405677
```

And as expected, we get the same answers earlier.

For profile `one_expert`, we get somewhat different results

```
pgd> one_expert(exp_hits)
8/5
pgd> one_expert(gt4_hits)
0.00247285
pgd> one_expert(gt2_hits)
0.13729411
pgd> one_expert(gt0_hits)
0.95217031
```

Try computing the results for the other profiles. The one profile that is different is `all_equal`, which replaces eight *different* symbol for the common symbol 'p'. Look at

```
pgd> all_equal(exp_hits)
8 p
pgd> all_equal(gt4_hits)
56 p^5 + -1.4E+2 p^6 + 1.2E+2 p^7 + -35 p^8
pgd> all_equal(gt2_hits)
56 p^3 + -2.1E+2 p^4 + 336 p^5 + -2.8E+2 p^6 + 1.2E+2 p^7 + -21 p^8
pgd> all_equal(gt0_hits)
8 p + -28 p^2 + 56 p^3 + -7E+1 p^4 + 56 p^5 + -28 p^6 + 8 p^7 + -1 p^8
```

and the results are still complex but give us more insight. In fact, if we compare `exp_hits` and `all_equal(exp_hits)` we see that latter gives $8p$. Had there been n hunters instead of eight, we can surmise that the expected number of hits would be np . The probabilities `gt4_hits` et cetera are also simpler polynomials in p in this case, though it takes a bit of simplification to make them digestible (as we will see later on). Note, however, that $\mathbb{E}(\{H > j\})$ has all powers of p from $j + 1$ to 8.

We can ask one more question. The expected number of hits gives our prediction for the number of hits, but does not tell us much about the *consistency* of these results. Consider the profiles `all_50_50` and

```
hot_and_cold = substitute_with(dict(h_0=0.05, h_1=0.05, h_2=0.05, h_3=0.05,
                                     h_4=0.95, h_5=0.95, h_6=0.95, h_7=0.95))
```


Both profiles have 4 as the expected number of hits, but which would be more consistent if the hunters tried on target after target? We define a statistic to answer that question, which gives us the distance between number of hits and 4:

```
@statistic(codim=1)
def spread(num_hits):
    return abs(num_hits - 4)
all_50_50( E(spread(number_of_hits)) )
hot_and_cold( E(spread(number_of_hits)) )
```

The expectation of `spread(number_of_hits)` just computes the MAD measure of uncertainty from Example 6.4; it predicts how far H will be from $\mathbb{E}(H)$. The first profile gives about 1.09 and the second 0.33. Thus, our prediction is that if we repeat the experiment many times, the number of hits will “typically” be about 4 ± 1.09 if all hunters are 50-50 shots but 4 ± 0.33 if half are excellent shots and the others terrible. The consistency of the outcome is stronger in the second case; the values are more “spread” on repetition in the first case.

The pattern in the previous example is that we collect several independent random outcomes and apply a statistic to answer a question. We took the independent join of all eight hunters’ attempts and asked a question about those outcomes. (How many hit? Did more than 4 hit?) This emphasizes the role that statistics play in our analysis, which is to express questions that we ask of our data. The next example is a similar instance of this pattern, except it does not entail an *independent* join.

Example 8.4 Six of One, Equilateral of the Other

On a regular hexagon with maximum width 2, we choose three distinct vertices randomly so that all subsets of 3 vertices have equal weight.

Define two FRPs:

- A represents the area of the triangle formed by connecting the three vertices.
- Q is the event that the triangle formed by the three vertices is equilateral.

Find the Kind of Q and $\mathbb{E}(Q)$, and find the Kind of A and $\mathbb{E}(A)$.

Imagine that we have labeled the vertices of the hexagon $1, 2, \dots, 6$ in the clockwise direction from one of the vertices (chosen arbitrarily). Picking three distinct vertices means picking a subset of size three from the set $\{1, 2, \dots, 6\}$. There are $\binom{6}{3} = 20$ such subsets.

In the playground, we can use the Kind factory `without_replacement` to

find the Kind of a randomly selected subsets where all subsets have equal weight.

```
vertices = without_replacement(3, irange(1, 6))
```

With `size(vertices)`, we can check that there are 20 possible values. These tuples of indices are in the original order (like $\langle 1, 2, 4 \rangle$, $\langle 2, 5, 6 \rangle$), so we have an equilateral triangle if there is *a gap of exactly 2* between successive numbers in that list. The built-in statistic `Diff` computes the differences between successive components in its input. A condition that tests for gaps of 2 is:

```
is_equilateral = Diff == (2, 2)
```

The Kind of Q is a transform of the Kind of the vertices:

```
equilateral = is_equilateral(vertices)
```

This transformed Kind combines all branches (out of 20) in `vertices` with the same value of the statistic `is_equilateral`, adding their weights. You might find it useful to look at `vertices` and `equilateral` in the playground.

```
pgd> E(equilateral)
1/10
```

yielding $\mathbb{E}(Q) = 1/10$. There are two choices, $\langle 1, 3, 5 \rangle$, $\langle 2, 4, 6 \rangle$, that satisfy the required condition. To see this, enter

```
vertices ~ Fork(Id, is_equilateral)
```

which will show all the values of `vertices` marked with an extra component that is 1 when the condition is true.

To analyze A , it will help to use Heron's formula: the area of a triangle with side lengths a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$ is the "semi-perimeter." This suggests a statistic that maps the three side lengths of the triangle to the triangle's area:

```
@scalar_statistic
def heron(a, b, c):
    "returns the area of a triangle with given side lengths"
    s = (a + b + c) / 2
    return Sqrt(s * (s - a) * (s - b) * (s - c))
```

We will get the side lengths of the triangle from another statistic.

If the hexagon has width 2, it has side length 1. So, if we pick two vertices that are separated by 1 or 5 in our label, the distance is 1; by 2 or 4, the distance is $\sqrt{3}$; and by 3, the distance is 2.

```
vertex_dists = [0, 1, numeric_sqrt(3), 2, numeric_sqrt(3), 1]

@statistic
def side_lengths(triangle):
    "computes triangle side lengths from vertex positions on hexagon"
    return [vertex_dists[numeric_abs(triangle[i] - triangle[(i - 1) % 3])]
            for i in range(3)]
```

where the $(i - 1) \% 3$ picks the last index when $i == 0$. Putting this together, we apply both statistics and add a `clean` operation to eliminate round-off error in the calculations, yielding the Kind of A :

```
pgd> area = clean( vertices ^ side_lengths ^ heron )
      ,---- 0.30 ---- 0.43301
<> -+----- 0.6 ----- 0.86603
      `---- 0.1 ----- 1.2990
```

with exact values $\sqrt{3}/4$, $\sqrt{3}/2$, and $3\sqrt{3}/4$. $E(\text{area})$ matches equation (6.29)

$$\mathbb{E}(A) = 0.3 \frac{\sqrt{3}}{4} + 0.6 \frac{\sqrt{3}}{2} + 0.1 \frac{3\sqrt{3}}{4} = 0.45\sqrt{3},$$

as you can confirm in the playground.

Example 8.5 Tournament

Eight players are ranked and play in an elimination tournament. In the first round, player 1 (top ranked) is matched against player 8 (bottom ranked), 2 against 7, 3 against 6, and 4 against 5. The winner of each match advances to the next round with 1 or 8 against 4 or 5 and 2 or 7 against 3 or 6. And so on until only one player remains.

In any given match, if players of rank $r_1 \leq r_2$ are competing, the better seeded player (of rank r_1) is $1.15^{r_2 - r_1}$ times more likely to win.

Find the Kind of the rank of the player who wins the tournament and its

expectation. Find the probability that a player from the bottom half of the rankings wins the tournament.

We will model this situation by keeping track of the players (by rank) who are still in the tournament. We will arrange their numbers in a tuple such that, at each stage in the tournament, each pair of players who are matched will be adjacent in the tuple. Thus, for the first round, we have

```
first_round = constant(1, 8, 4, 5, 2, 7, 3, 6)
```

We can see that 1 and 8 are matched, as are 4 and 5, 2 and 7, and 3 and 6. Moreover, who ever wins in the first round will be next to the player they are matched against in the next round. So, 1 or 8 will play 4 or 5. And similarly in the final round.

Next, we need a conditional Kind that takes the slate of players in the current round and produces a Kind for the slate of players in the next round. To do this, we move down the input tuple of players in pairs and for each pair r_1, r_2 produce the Kind

$$\langle \rangle \left\{ \begin{array}{l} \text{---} 1 \text{---} \langle r_2 \rangle \\ \text{---} 1.15^{r_2 - r_1} \text{---} \langle r_1 \rangle \end{array} \right.$$

for the winner of that matchup. (This Kind works regardless of which of r_1, r_2 is smaller.) The independent join of these Kinds is then the output.

The conditional Kind for the second round has type $8 \rightarrow 4$, for the third round has type $4 \rightarrow 2$, and for the final round has type $2 \rightarrow 1$. We can implement all of these conditional Kinds in a single function:

```
@conditional_kind
def next_round(players):
    n = len(players) # Always a power of 2 here
    k = Kind.empty
    for i in range(0, n, 2):
        r1, r2 = players[i], players[i + 1]
        odds = as_quantity('1.15') ** (r2 - r1)
        k = k * either(r1, r2, odds)
    return k
```

With this in hand, we can now express the Kind of the next round by *conditioning next_round on the current round*.

```
second_round = next_round // first_round
third_round = next_round // second_round
winner = next_round // third_round
```

Look at these in the playground; they act as we expect. For example, the combination $\langle 1, 4, 2, 3 \rangle$ is most likely to come out of the first round, with $\langle 1, 5, 2, 3 \rangle$ right behind. Similarly, the top ranked players are more likely to win the tournament in the right order. Here's what the Kinds of the third-round matchups and the winner look like:

```
,---- 0.17270 ----- <1, 2>
|---- 0.14158 ----- <1, 3>
|---- 0.075013 ----- <1, 6>
|---- 0.060314 ----- <1, 7>
|---- 0.094949 ----- <4, 2>
|---- 0.077838 ----- <4, 3>
|---- 0.041242 ----- <4, 6>
|---- 0.033160 ----- <4, 7>
<> -|
|---- 0.076683 ----- <5, 2>
|---- 0.062864 ----- <5, 3>
|---- 0.033308 ----- <5, 6>
|---- 0.026781 ----- <5, 7>
|---- 0.039784 ----- <8, 2>
|---- 0.032614 ----- <8, 3>
|---- 0.017280 ----- <8, 6>
`---- 0.013894 ----- <8, 7>

,---- 0.26520 ----- 1
|---- 0.20843 ----- 2
|---- 0.16017 ----- 3
|---- 0.12058 ----- 4
<> -|
|---- 0.090552 ----- 5
```

```
|---- 0.068000 ---- 6
|---- 0.050322 ---- 7
`---- 0.036742 ---- 8
```

We can see for instance that players seeded 1 and 2 have about a 17% probability of meeting in the finals and player 1 has about a 26.5% probability of winning.

To remind yourself how we compute these probabilities, recall that, say, `next_round // third_round` starts by computing the join `third_round >> next_round` and then projects out the last component. Each possible slate of players produced in the next round is derived from some slate in the current round through particular outcomes of those match-ups. We compute the probability of that slate by finding all the combination of current round and match-up outcome that produce it (the join) and then adding up the weights on all the branches with that slate (the projection). Look at the join in the playground and track through the projection to see it.

For example, the winner $\langle 1 \rangle$ can occur from any of the match-ups $\langle 1, 2 \rangle$, $\langle 1, 3 \rangle$, $\langle 1, 6 \rangle$, or $\langle 1, 7 \rangle$ in the current round. Those match-ups happen with probabilities about 0.173, 0.142, 0.075, 0.060. And in those match-ups the player seeded 1 wins with probabilities $\frac{1.15}{1+1.15}$, $\frac{1.15^2}{1+1.15^2}$, $\frac{1.15^5}{1+1.15^5}$, and $\frac{1.15^6}{1+1.15^6}$, which come from the conditional `Kind next_round`. The result is

$$0.173 \cdot \frac{1.15}{1+1.15} + 0.142 \cdot \frac{1.15^2}{1+1.15^2} + 0.075 \cdot \frac{1.15^5}{1+1.15^5} + 0.060 \cdot \frac{1.15^6}{1+1.15^6} \approx 0.265.$$

The same basic story holds at each stage. See page [315](#).

Now, we can answer our questions

```
pgd> E(winner)
3.151837905582001
pgd> E(winner ^ (__ > 4))
0.2456155413215654
pgd> E(winner ^ (__ == 1))
0.2652034184779578
```

which give values of about 3.15 and just under 0.25, respectively. The top-seeded player has about a 26.5% probability of winning the tournament, but the other players – especially the 2 and 3 and 4 seeds – have a non-trivial chance of winning as well. So our prediction for the winning seed is just over 3.

The next example illustrates the important concept of *state* that describes the

configuration of a random system at a given moment. (See Section 7.2.) We need enough information in the state to understand the future evolution of the system at any point but not so much information that it obscures the essentials. Crafting a good description of state for a process is part art and part science, and we will get lots of practice. As you read the next example, think about how you would describe the state of the mouse's process.

Example 8.6 Mouse Escape

A mouse is caught in a room with a cat and wants desperately to escape. The room has been newly refurbished, so the usual escape routes have been plugged. The only hope is for the mouse to climb three steps to safety.

The mouse starts on the floor (step 0) and successively attempts to climb onto the next step. If the mouse fails it tumbles down to the *previous* step (except on step 0); if it succeeds, it moves to the next step and tries again. So for example, if it fails to climb to step 2 from step 1, it falls to step 0; if it succeeds, it moves to step 2.

If the mouse reaches step 3 by the 16th attempt, it escapes; otherwise the cat eats it. On each attempt the Kind of whether it succeeds or fails has twice the weight on failure.

Let M be the event that the mouse escapes. Find $\mathbb{E}(M)$, the probability that the mouse escapes.

The process by which the mouse tries to escape from its first attempt to its eventual escape or capture can be described by a state that evolves from attempt to attempt. What do we need to keep track of to be able to model the evolution of the process and answer the questions we care about?

First, at any point, we need to know which step the mouse is on, 0, 1, 2, or 3. If we know where the mouse is, we can determine the possibilities for its subsequent attempts. Second, we also need to know how many attempts the mouse has made because if it takes too many the cat will catch it.

We have two choices. We can set the state as the step the mouse is on and account for the possibility of capture by analyzing whether it escapes by the 16th attempt, or we can include the number of attempts in the state as well. We will illustrate both approaches here.

If the state is just the step the mouse is on, then the initial state is 0, and the mouse moves up or down (or stays at 0) with twice the weight on down.

```

initial_state = constant(0)
move = conditional_kind({
    0: either(0, 1, 2),
    1: either(0, 2, 2),
    2: either(1, 3, 2),
    3: constant(3)
})

```

The first is the Kind of the initial state, and the second is the conditional Kind of the next state *given* the initial state. For instance, from step 0, the mouse either stays at 0 or moves to 1 with twice the chance of staying as moving up. If the mouse is at step 3, we model it as staying there; it has escaped. If it has reached state 3 by the 16th attempt, we know that it escaped on or before that attempt.

We can compute this by starting in the initial state and iterating for 16 attempts:

```

state = initial_state
for _ in range(16):
    state = move // state
escaped = E(state ^ (__ == 3))

```

The operation `move // state` is *conditioning*, specifically, we are finding the Kind of the next state by conditioning on the current state. The conditional Kind `move` gives the next state's Kind for each value of the current state, but it does not give any information about how likely any value of the state is. We can view the conditioning operation as computing the Kind of the next state as a weighted average of the Kinds `move(s)` over all values of `state` using its canonical weights. (The loop above is implemented by the builtin playground function `evolve` with `evolve(initial_state, move, 16)`.)

Our question is answered by the expectation `escaped`, which is approximately 0.368. This is the probability that the mouse escapes before the cat catches it. In this approach we account for the mouse's time horizon by evolving the system over 16 moves.

For the second approach, we use an expanded definition of state that keeps track of the number of moves and freezes the state when either the mouse escapes or the cat eats it. This approach is slightly more complicated, but it illustrates

how we can include contingent dynamics in our systems.

Now, the state is a tuple $\langle n, s \rangle$, where n is a number of attempts and s is either a step (0, 1, 2, 3) or -1 to indicate that the mouse has been captured. The initial state is $\langle 0, 0 \rangle$, and we have

```
initial_state_alt = constant(0, 0)
```

Our state transitions now depend on how many moves the mouse has made. If the mouse has escaped or been eaten, we simply keep the state unchanged. If it is the 16th attempt, it will be captured if it doesn't make it to step 3. Otherwise, the mouse moves as before and the number of attempts is incremented.

```
@conditional_kind
def move_alt(attempts_and_step):
    n_attempts, step = attempts_and_step

    # If we are at the end, stay there
    if step == 3 or step == -1:
        return constant(attempts_and_step)

    n = n_attempts + 1

    # If the cat is here, last chance
    if n_attempts == 16:
        if step < 2:
            return constant(n_attempts, -1)
        else:
            return either((n, -1), (n, 3), 2)

    # From step 0, we either stay or move up
    if step == 0:
        return either((n, 0), (n, 1), 2)

    # Otherwise, we move up or down
    return either((n, step - 1), (n, step + 1), 2)
```

To answer our questions, we evolve the system and transform the resulting Kind as we did earlier, except the mouse's outcome is in the second component.

```

mouse_outcome = evolve(initial_state_alt, move_alt, 16)
escaped_alt = E(mouse_outcome ^ (Proj[2] == 3))

```

We evolve the system 16 steps because we need not do more, but evolving it for longer would not change the result because the state becomes fixed.

8.4 Strategies and Representations

Despite their power, neither computation nor mathematics are fully “automatic.” We often need to apply some creativity to get useful results, either to make a computation feasible/efficient or to make a mathematical analysis tractable. In this Section, we look at examples where we bring a little flare to select our strategies or data representations.

These examples demonstrate some common patterns in how we approach more complicated systems. The next example illustrates the pattern of optimizing a decision over a menu of strategies for making that decision.

Example 8.7 Assistant Assistance

You are trying to hire an assistant to help you with your work, so you place an ad in the local paper. The next day, exactly n applicants call you to schedule an appointment for an interview, and you schedule them at random in n time slots. Assume that all ordering of the appointments are equally likely, i.e., have the same weight.

You have a very particular set of criteria in mind for the position. At each interview, you evaluate the applicant’s qualifications with respect to these criteria, so after each interview, you can unambiguously rank the applicants you’ve seen up to that point. However, the job market is running hot, and if you do not offer the job to an applicant immediately after the interview, the applicant will take another job and is lost to you.

Our task is to pick a good strategy for deciding whether to offer an applicant the job and to assess how well that strategy performs. We will consider the family of strategies in which you reject the first k applicants and then offer the job to the first applicant after these that ranks better than all of those first k . There are n different strategies here, which we call “After 0”, “After 1”, “After 2”, ..., “After $n - 1$ ”. In the “After 0” strategy you would always choose the first applicant. With “After k ” for $k > 0$, it is possible to make no offers. We will say that our strategy has *succeeded* if we make an offer to the overall best-ranked

applicant.

Find the probability that the “After k ” strategy succeeds for each $k \in [0..n)$. Which of these strategies is most likely to succeed?

We start by reasoning to reduce the problem to a simpler form in two steps. First, we only need to keep track of the positions of two applicants: the “best” applicant who has the best rank overall and the “pre-best” applicant who has the best rank among all the applicants who appear *before* the best applicant. The “After k ” strategy succeeds in finding the best applicant if the best applicant appears in position b with $b > k$ and the pre-best applicant appears in position s with $s \leq k$. If $b = 1$, there is no pre-best applicant, and we set $s = 0$. (If $b \leq k$, the best applicant would be skipped. If $k < s < b$, we would choose the pre-best applicant who has higher rank than all those at positions $[1..k]$.)

Second, because all orderings of the applicants are equally likely,

- (i) the best applicant is equally likely to be in any position, and
- (ii) *given* that the best applicant appears in position $b > 1$, the position of the pre-best applicant is equally likely to be in any position in $[1..b-1]$.

Claim (i) is true because the Kind for the permutation of the n ranks has size $n!$ with all equal weights. So for any of the n positions of the best applicant, there are $(n-1)!$ branches permuting the other weights, so the position of the best applicant has weight $1/n$ on each possible value. Claim (ii) is true by a similar argument; once we fix the position of the best applicant, the remaining branches iterate over all orderings of ranks that fit before the best applicant, all of which have equal weight. Using the code from this example, you can see this in the playground by entering `check_best_position()` to see the Kind of the best position when $n = 7$ and `check_pre_best_position(b)` to see the Kind of the pre-best position (when $n = 7$) for each best position.

The direct but inefficient way to solve this problem would be to construct the Kind of all orderings of the n rankings, transform this with a statistic that extracts the best and pre-best positions, and then transform with a condition that $b > k$ and $s \leq k$. In the playground, for specific values of n and k , this would look like

```
permutations_of(irange(n)) ^ best_pre_best_of ^ is_success(k)
```

using the statistics `best_pre_best_of` and statistic factory `is_success` defined in the example code, e.g.,

```

def is_success(k):
    "Statistic factory that checks if After-k succeeds given <b,s>."
    @condition
    def succeeded(b, s):
        return b > k and s <= k

    return succeeded

```

For small n , this works fine, but the size of the initial Kind grows quickly.

Instead, we will use our earlier reasoning to directly derive the Kind of the best and pre-best positions. To allow this to work for any n , we wrap the whole analysis in a function that takes n as a parameter:

```

def assistant(n):
    "Returns success probabilities of After-k strategies with `n` applicants."
    assert n >= 1, "at least one applicant is required"

    best = uniform(irange(n))

    @conditional_kind(codim=1)
    def pre_best_position(m):
        if m <= 1:
            return constant(0)
        return uniform(irange(1, m - 1))

    best_pre_best = best >> pre_best_position

    success_probs = [0] * n
    for k in range(n):
        success_probs[k] = as_scalar( E(best_pre_best ^ is_success(k)) )

    return as_vec_tuple(success_probs)

```

(The application of `as_scalar` to the expectation unwraps the number from the tuple.) Here is how we use it:

```

pgd> assistant(7)
<0.14285714285714286, 0.35, 0.41428571428571423, 0.4071428571428571,

```

```

0.35238095238095238, 0.2619047619047619, 0.14285714285714286>

# The direct approach gives the same answers, compare k=2 and k=3
pgd> E(permutations_of(irange(7)) ^ best_pre_best_of ^ Fork(is_success(2), is_success(3)))
<0.4142857142857143, 0.4071428571428571>

pgd> best_k(assistant(7))    # Find k that maximizes the probability
<2, 0.4142857142857143>

```

The function `best_k` is defined in the example code; it returns the best k and its success probability for the specified n .

Trying this for various n , we see the optimal After- k strategy, k^* , has $k^* \approx ne^{-1}$ with a probability approaching $e^{-1} \approx 0.36788$ for large n . These are shown for various n in Table 8.1. We will see later where this comes from.

n	k^*	$\lfloor \frac{n}{e} \rfloor$	$\lceil \frac{n}{e} \rceil$	Probability
10	3	3	4	0.39869
20	7	7	8	0.38421
30	11	11	12	0.37865
40	15	14	15	0.37574
50	18	18	19	0.37428
60	22	22	23	0.37321
70	26	25	26	0.37239
80	29	29	30	0.37186
90	33	33	34	0.37142
100	37	36	37	0.37104
250	92	91	92	0.36915
500	184	183	184	0.36851
1000	368	367	368	0.36820

TABLE 8.1. Optimal After- k strategy, k^* for selected n and the probability of success for that strategy, in Example 8.7.

The next example uses a carefully chosen, but somewhat elaborate, representation of the system's state to make it possible – in fact, straightforward – to compute the answer we seek. The choice of representation still needs to keep track of the essential information, but by discarding everything else, computations that would be slow become manageable.

Example 8.8 Elevator Stops

An elevator opens on the ground floor and a random number of passengers enter it. Each passenger selects one floor (above the ground floor), and the elevator proceeds upward, stopping at each selected floor.

- There are n floors above the ground floor.
- The FRP N represents the number of passengers entering the elevator. Its Kind of N is given below. It depends on numeric parameters p and μ and is given below.
- Each passenger's chosen floor is represented by an FRP with Kind `uniform(1, 2, ..., n)`, with all floors equally likely.
- All passenger choices are independent of each other.

If no passengers enter, then the elevator makes zero stops. The FRP S represents the number of stops the elevator makes. For any positive integer p and $\mu > 0$, the Kind of N is

$$\begin{array}{rcl}
 & 1 & \langle 0 \rangle \\
 & \mu & \langle 1 \rangle \\
 & \mu^2/2 & \langle 2 \rangle \\
 & \mu^3/3! & \langle 3 \rangle \\
 \langle \rangle & \mu^4/4! & \langle 4 \rangle \\
 & \dots & \\
 & \mu^{p-1}/(p-1)! & \langle p-1 \rangle \\
 & \mu^p/p! & \langle p \rangle
 \end{array}$$

The parameter p is the maximum number of passengers allowed in the elevator, and μ determines the weights. In the example code, the function `passengers_kind(p, mu)` computes this Kind in canonical form. **Take** $n = 10$, $p = 21$, and $\mu = 5$. **Find** $\mathbb{E}(S)$.

With 20 or more passengers possible, the Kinds we need will be slow to compute because there are 10 possible floors for each passenger. Our first insight is that keeping track of the floor each passenger visits is not really necessary. We only need to keep track of *which floors are visited*, whether they are visited by one or many passengers is irrelevant to our needs.

Thus, the state we use for describing this system is **the set of visited floors**. With $n = 10$, we encode this set as a 10-tuples containing only 0's and 1s; a one in slot i means that floor i is visited by at least one passenger and a 0 means

that floor is not visited by any passengers. The Kind for the set of visited floors with this representation has size ≤ 1024 , which is manageable. (It does grow quickly with n , but we are keeping n fixed here.)

If we are going to use 10-tuples of bits to represent the set of visited floors, we need two operations on these sets: (1) converting a list of requested floors to a set in our representation, and (2) combining two sets into one, taking a 20-tuple that encodes two sets (10 components each) and returning a single 10-tuple representing their union. These operations are given, respectively, by the statistics `visited_floors(10)` and `union_visited(10)`. These are both nice examples of how we can use statistics to convert data from one representation to another. We implement these as *statistic factories* that take the index of the top floor and return the statistic we seek for that building. Here, we will only be using these functions with an argument of 10.

```
def visited_floors(top_floor: int) -> Statistic:
    "Returns a statistic converting a list of floor choices to a set."
    @statistic(name=f'visited_floors<{top_floor}>')
    def visited_set(value):
        "returns the set of unique components in a fixed range, as a bit string"
        bits = [0] * top_floor
        for x in value:          # values are floors in 1, 2, ..., top_floor
            bits[x - 1] = 1      # this floor's button has been pushed
        return as_vec_tuple(bits)
    return visited_set

def union_visited(top_floor: int) -> Statistic:
    "Returns a statistic that unions two `top-floor` sets as bit-strings."
    @statistic(name=f'union<{top_floor}>', monoidal=as_vec_tuple([0] * top_floor))
    def union(value):
        "unions two `top_floor`-length bit strings into one with a bitwise-or"
        return as_vec_tuple(value[i] | value[i + 10] for i in range(top_floor))
    return union

max_floor = 10
as_set = visited_floors(max_floor) # Statistic: convert floors to visited sets
```

```
union = union_visited(max_floor)    # Statistic: Union of two sets as 10-bit strings
```

Now we are ready for our analysis. First, we build a conditional Kind `floors` that maps the number of passengers to the Kind of the visited floor set for that many passengers. We do this iteratively, by mixing in the choices of one new passenger to our previously computed Kinds.

```
passenger_floor = uniform(1, 2, ..., max_floor)
choice = as_set(passenger_floor)    # Kind for each floor choice as set
floors = {0: constant([0] * max_floor), 1: choice, 2: union(choice * choice)}
for i in irange(2, 20):
    floors[i + 1] = union(floors[i] * floors[1])
visited = conditional_kind(floors)   # kind of visited floor set for each # pass.
```

Each element in the loop adds a new entry to the conditional Kind that accounts for an extra passenger's choices in the set of visited floors. You should look at a few of these Kinds in the playground.

If we apply the `Sum` statistic to this conditional Kind, it transforms it to a new conditional Kind that maps the number of passengers to the Kind of the *number of visited floors*. (Make sure you see why.) All that remains is to mix in the number of passengers. We use the conditioning operator `//` because we want to average over the various numbers of passengers without retaining it in the result.

```
number_passengers = passengers_kind(21, 5)
number_visited_floors = Sum(visited) // number_passengers
```

The Kind `number_visited_floors` looks like

```
pgd> number_visited_floors
,---- 0.0067379 ----- 0
|---- 0.043710 ----- 1
|---- 0.12760 ----- 2
|---- 0.22074 ----- 3
|---- 0.25060 ----- 4
<> -+---- 0.19508 ----- 5
|---- 0.10546 ----- 6
|---- 0.039095 ----- 7
|---- 0.0095105 ----- 8
```



```
|---- 0.0013710 ----- 9
`---- 0.000088937 ---- 10
pgd> E(number_visited_floors)
3.934693309902328
```

So, $\mathbb{E}(S) \approx 3.935$. In the example code, this analysis to get the Kind `number_visited_floors` is run by calling `elevator_stops(max_floors=10, p=21, mu=5)`.

There were a lot of moving parts in the last example, but the key feature was changing the representation of the system from a list of floors requested/visited by passengers in the elevator to a fixed-size set indicating which floors have been visited. With this representation, we could account for each added passenger with an independent join followed by a statistic that combines the two sets. This is a complicated example of a common pattern.

Example 8.9 Rig at Risk

A mid-ocean oil rig accumulates damage from severe waves over time. Assume that in a given year, the number of severe waves is given by the value of an FRP N and that the damage caused by severe wave i is given by the output of FRP D_i , where all the D_i 's have the same Kind. Assume that the D_i 's are independent of N . Let T be the FRP representing the total damage that accumulates in a year.

Find $\mathbb{E}(T \mid N = n)$, the expectation of the total damage during the year given that there were n severe waves, and $\mathbb{E}(T)$.

There are two things to specify in this problem: the Kinds of N and D . But we can do the analysis and get some insight by doing the analysis in terms of these unspecified Kinds.

The first point to recognize is that the structure is simple when we know the value of N . Specifically, *given* that there are n waves, the Kind of the total damage can be expressed in two steps:

1. take an independent join of n copies of D_1 's Kind, and
2. apply the `Sum` statistic.

That is,

$$\text{kind}(T \mid N = n) = \text{Sum}(\text{kind}(D_1) \star n).$$

On the left, we have a conditional Kind (given the value n of N) and the right

an exact expression for it. By the Additivity property (6.4) of expectation,

$$\mathbb{E}(\text{Sum}(\text{kind}(D_1) \star \star n)) = \mathbb{E}(D_1) + \mathbb{E}(D_2) + \cdots + \mathbb{E}(D_n),$$

and because all D_i 's have the same Kind and expectation,

$$\mathbb{E}(T \mid N = n) = n \mathbb{E}(D_1).$$

So viewing $\text{kind}(T \mid N = n)$ as a conditional Kind, we can *condition on* N :

$$\text{kind}(T \mid N = n) \text{ // } \text{kind}(N)$$

See page 315.

and take expectations. As we've seen earlier, this operation averages the Kinds $\text{kind}(T \mid N = n)$ over n weighting by $\text{kind}(N)$. The expectation is therefore

$$\mathbb{E}(T) = \sum_n p_n \mathbb{E}(T \mid N = n), \quad (*)$$

where p_n is the weight on n in $\text{kind}(N)$.

The function `rig_at_risk` in the example code implements this in `frplib`.

```
def rig_at_risk(kind_N, kind_D):
    """Computes expectation of total wave damage.
    Kinds `kind_N` and `kind_D` represent the number of waves and damage per wave.

    """
    @conditional_kind(codim=1)
    def damage_given_n(n):
        return fast_join_pow(Sum, kind_D, n)
    return E(damage_given_n // kind_N)
```

For instance:

```
pgd> rig_at_risk(uniform(1, 2, ..., 10), uniform(1, 2, 3))
11
```

You can play with different input Kinds using the example code.

8.5 Using Observations

In general, we build our model and compute our predictions before the random process we are studying begins. But we allow for making decisions or interventions as the process unfolds, so we need the ability to *update our predictions* in light of new information about uncertain quantities. This is the role of observations.

The results of such updates can seem counter-intuitive because they balance multiple possibilities. The easiest way to handle this is to remember that constraining with observations simply eliminates the branches of the Kind that are inconsistent with our observations. When we renormalize into canonical form, the numbers change, but the relative sizes of the remaining branches' weights *do not change*. Put more glibly: probability does not move when we update our predictions.

The next example⁹ is gleefully counter-intuitive. We will do the calculations in `frplib` and then try to understand them.

⁹Adapted directly from (Mlodinow, 2008).

Example 8.10 A Kid Named “Florida”

We consider two questions

- (a) You know that a neighbor’s family has two children, but you cannot remember whether the children are boys or girls. One day, you see that one of the children is a girl.
- (b) Suppose that during the previous experiment we learn that one of the neighbor’s two children is a girl whose name is very rare, for instance “Florida.” (Mlodinow, 2008)

What is the probability that both children are girls in each situation?

It seems strange that the rarity of the names could have much of an effect, but we’ll see that it does. Let’s solve both parts and then regroup to understand the results.

First, we consider the various outcomes that might occur, ignoring the childrens’ names. The question we want to answer and the information that we observe are represented as statistics:

```
at_least_one_girl = Or(Proj[1] == 1, Proj[2] == 1)
both_girls = And(Proj[1] == 1, Proj[2] == 1)
```

We define the Kind for an event that an individual child is a girl:

```
girl = either(0, 1)
```

Then we look at the Kinds for the various outcomes of interest: whether each of two children are girls and whether each of two children are girls given that at least one is.

```
pgd> girl * girl
,---- 1/4 ---- <0, 0>
|---- 1/4 ---- <0, 1>
<> -|
|---- 1/4 ---- <1, 0>
`---- 1/4 ---- <1, 1>
pgd> outcome_no_names = girl * girl | at_least_one_girl
pgd> outcome_no_names
,---- 1/3 ---- <0, 1>
<> -+---- 1/3 ---- <1, 0>
`---- 1/3 ---- <1, 1>
```

The observational constraint eliminates the $\langle 0, 0 \rangle$ branch. Note that the relative size of the weights in the remaining branches *does not change*. Dropping the branch gives three branches with weights $1/4$, and when we re-normalize weights $1/3$. The information that there is at least one girl has simply ruled out the possibility that neither child is a girl. And hence:

```
pgd> both_girls(outcome_no_names)
,---- 2/3 ---- 0
<> -|
`---- 1/3 ---- 1
```

and $E(\text{both\_girls}(\text{outcome\_no\_names})) = 1/3$. is the desired probability in situation (a).

The information we have in the second situation is somewhat different; we know that at least one of the children is a girl with a rare name. So we need, for each child, events that the child is a girl and that the child has a rare name. We encode the latter events with a Kind with a symbolic weight and use arbitrary values 10 and 11 (rather than 0 and 1) to make it easier to identify the components in the values. A rare name means that p is a *small* number.

```
p = symbol('p')
rare = weighted_as(10, 11, weights=[1 - p, p])
neighbors = girl * girl * rare * rare
```

By using an independent join for `neighbors`, we are assuming that the rarity of each child's name is independent of whether the child is a girl, as well as assuming that the two children's outcomes are independent. As above, we use a condition to represent our observation:

```
a_rare_girl = Or(
    And(Proj[1] == 1, Proj[3] == 11),
    And(Proj[2] == 1, Proj[4] == 11)
)
```

The Kind of interest then becomes

```
pgd> outcome = neighbors | a_rare_girl
,---- (-1 + p)/(-4 + p) ---- <0, 1, 10, 11>
|---- -1 p/(-4 + p) ----- <0, 1, 11, 11>
|---- (-1 + p)/(-4 + p) ---- <1, 0, 11, 10>
<> -+---- -1 p/(-4 + p) ----- <1, 0, 11, 11>
|---- (-1 + p)/(-4 + p) ---- <1, 1, 10, 11>
|---- (-1 + p)/(-4 + p) ---- <1, 1, 11, 10>
`---- -1 p/(-4 + p) ----- <1, 1, 11, 11>
pgd> both_girls(outcome)
,---- 1/(2 + -0.5 p) ----- 0
<> -|
`---- (-2 + p)/(-4 + p) ---- 1
```

Taking expectations $E(\text{both\_girls}(\text{outcome}))$, we get $\frac{2-p}{4-p}$. Because the name is rare – that is, p is small – $\frac{2-p}{4-p} \approx \frac{1}{2}$ and substantially bigger than $1/3$.

This is a surprising difference! We can get insight into this result by studying the Kind outcome. Since p is small, we can take it, for this purpose, to be so small that we can ignore it:

```
pgd> clean(substitution(outcome, p = 0))
,---- 1/4 ---- <0, 1, 10, 11>
|---- 1/4 ---- <1, 0, 11, 10>
<> -|
```

```
|---- 1/4 ---- <1, 1, 10, 11>
^---- 1/4 ---- <1, 1, 11, 10>
```

This Kind reveals what has happened. We are *very unlikely* to see *two* rare names, but the one rare name can be for *either* of the two girls. When only one of the children is a girl, the condition can be satisfied in *just one way*. Hence, instead of one out of three possibilities, we have two out of four.

Remember that when we constrain with a conditional, we eliminate branches that are inconsistent with the condition, but the *relative sizes of the weights on the remaining branches does not change*.

Example 8.11 Buckets and Balls

We have two buckets. In the left bucket, there are three red, six blue, and one green ball. In the right bucket, there are four red, four blue, and two green balls. The balls in both buckets are well mixed.

I choose a bucket at random, with equal weights on both, and then from that bucket choose a ball at random, again with equal weights on every ball. You do not see what bucket I chose the ball from, but I show you that I picked a green ball. What is the probability that I chose from the right bucket given this information?

We have a system that is most easily described in two stages. First, I pick a bucket. Second, I pick a ball from the chosen bucket. This is a join.

Let's assign 0 to the left bucket and 1 to the right; and let 0 and 1 stand also for not-green and green.

```
bucket = either(0, 1)
green_given_bucket = conditional_kind({
  0: either(0, 1, 9),
  1: either(0, 1, 4)
})
```

For the latter Kinds, the left bucket has 9 not-green balls and 1 green ball, a ratio of 9, and the right bucket has 8 not-green balls and 2 green balls, a ratio of 4. This explains the third argument to `either` in both cases. The FRP that represents the chosen bucket *given* that we have observed a green ball has Kind

```
which_bucket_g = Proj[1]( bucket >> green_given_bucket | (Proj[2] == 1) )
```

We get this in three stages: 1. use a join to build the combined Kind of bucket and chosen ball, 2. apply the constraint that we observed a green ball with a conditional, and 3. extract the first component (the bucket). We can write this more concisely as

```
which_bucket_g = bayes(observed_y=1, x=bucket, y_given_x=green_given_bucket)
```

We get $E(\text{which\_bucket\_g}) = 2/3$, the probability that we chose the right bucket given that we observe a green ball. Interestingly, we can also find

```
which_bucket_n = bayes(observed_y=0, x=bucket, y_given_x=green_given_bucket)
```

and $E(\text{which\_bucket\_n}) = 8/17$, the probability that we chose the right bucket given that we observe a not-green ball.

The previous example is a demonstration of **Bayes's Rule**, a common and important pattern we also saw earlier. We have two FRPs X and Y of arbitrary dimension. We know the Kind of X , and we know the *conditional* Kind of Y given X . We then *observe* Y and want to *infer* X . In the previous example, X represents to the bucket we choose the ball from, and Y is the event that we pick a green ball. We build our model for this system with a join because it is easier to specify the Kind of ball we pick once we know the bucket.

We know $\text{kind}(X)$, and for any possible value u of X ; we know $\text{kind}(Y \mid X = u)$; and we have observed a value v of Y . Note that the mapping from u to $\text{kind}(Y \mid X = u)$ is a *conditional kind* that we can denote by $\text{kind}(Y \mid X)$.

Bayes's Rule is equivalent to three steps:

1. Build with a join the Kind of the combined outcome $\langle X, Y \rangle$:

$$\text{kind}(\langle X, Y \rangle) = \text{kind}(X) \triangleright \text{kind}(Y \mid X).$$

2. Constrain this with a conditional using the observation that $Y = v$:

$$\text{kind}(\langle X, Y \rangle) \mid Y = v$$

3. Transform with a projection statistic to extract the X components $1, \dots, \text{dim}(X)$:

$$\text{proj}_{1.. \text{dim}(X)} (\text{kind}(\langle X, Y \rangle) \mid Y = v).$$

The first step gives the combined Kind for X and Y ; the second applies the constraint from our observation; and the third isolates the Kind of X , since that is what we

want to know (and we have already observed Y 's value).

In the playground, we can implement these same steps easily. Letting $\mathbf{x}$ and $\mathbf{y\_given\_x}$ stand for $\text{kind}(X)$ and $\text{kind}(Y \mid X)$, the built-in `bayes` method looks like

```
def bayes(observed_y, x, y_given_x):
    i = dim(x) + 1
    return (x >> y_given_x | (Proj[i:] == observed_y)) ^ Proj[1:i]
```

The `bayes` function will work just as well with FRPs as with Kinds; in that case, $\mathbf{x}$ would be the FRP X , and $\mathbf{y\_given\_x}$ would be the conditional FRP $Y_{\text{given } X}$.

Let's use this again, generalizing an earlier example.

Example 8.12 Disease Testing Redux

A disease is prevalent in the population where in any large sample of people, a proportion around d will have the disease. A test has been developed to detect the disease. If a tested patient does *not* have the disease, the test will indicate they are negative with probability n . If a tested patient *does* have the disease, the test will indicate they are positive with probability p .

If a doctor sees that a patient has tested positive, what is the probability that the patient has the disease?

Let D be the event that the patient has the disease and T be the event that they tested positive. First, let's use the information provided to create $\text{kind}(D)$ and $\text{kind}(T \mid D)$, the conditional Kind of T given the observed value of D . These derive directly from the described assumptions.

```
d = symbol('d')
n = symbol('n')
p = symbol('p')

has_disease = weighted_as(0, 1, weights=[1 - d, d])
test_by_status = conditional_kind({
    0: weighted_as(0, 1, weights=[n, 1 - n]),
    1: weighted_as(0, 1, weights=[1 - p, p])
})
```

To find the probability that D occurs, we apply Bayes's Rule and take expectations:


```
pgd> disease_given_positive = bayes(1, has_disease, test_by_status)
pgd> disease_given_negative = bayes(0, has_disease, test_by_status)
pgd> E(disease_given_positive)
```

The result is

$$\frac{pd}{pd + (1 - n)(1 - d)}. \quad (*)$$

We can understand this by looking at the combined Kind of $\langle D, T \rangle$:

```
pgd> has_disease >> test_by_status
,---- n (1 - d) ----- <0, 0>
|---- (1 - d) (1 - n) ----- <0, 1>
<> -|
|---- (1 - p) d ----- <1, 0>
`---- p d ----- <1, 1>
```

The observation that the test is positive eliminates the first and third branch of this Kind, giving (non-canonical) Kind

```
,---- (1 - d) (1 - n) ----- <0, 1>
<> -|
`---- p d ----- <1, 1>
```

The remaining branches represent the possibilities that the patient tests positive and has the disease or that the patient tests positive and does not have the disease. The probability in (*) is just the normalized weight on the second branch.

We can examine these probabilities for various specific values to get a feel for how Bayes's Rule balances the prevalence – or *base rate* – of the disease in the population and the information about the sensitivity and specificity of the diagnostic test.

```
substitution(E(disease_given_positive),
             d='1/1000', n='99/100', p='95/100')
# is 0.0868
substitution(E(disease_given_positive),
             d='1/10_000', n='99/100', p='95/100')
# is 0.0094
substitution(E(disease_given_positive),
```

```

        d='1/10_000', n='999/1000', p='999/1000')
# is 0.0908
substitution(E(disease_given_positive),
        d='1/10_000', n='95/100', p='9/10')
# is 0.0018
substitution(E(disease_given_positive),
        d='1/100', n='95/100', p='9/100')
# is 0.1538
substitution(E(disease_given_positive),
        d='1/100', n='999/1000', p='999/1000')
# is 0.9098

```

The first thing to notice is that, except in the last case, the probability of the patient having the disease is quite low, *even with a positive test* and even when the test is highly accurate. The reason is that in these cases the base rate of the disease is low, and as we saw above, the probability accounts for the two possibilities: patient has the disease and tests positive versus does not have the disease and tests positive. Only when the disease is somewhat common and the tests accurate do we get a high probability.

8.6 Touching Infinity

FRPs are particularly suited as models of *finite* random processes: a finite number of possibilities, finite dimension, and a finite horizon of time and space. But in some cases, we can push beyond the finite and get exact results for more general processes.

Example 8.13 Waiting for Heads

We flip a coin repeatedly up to and including the first flip on which a heads comes up. How many flips will it take?

Let H be the FRP representing the number of flips required up to and including the first heads. We can ask several questions about H :

- What is a typical number of flips, $\mathbb{E}(H)$?
- How likely are we to need more than n flips, $\mathbb{E}(\{H > n\})$?
- How likely are we to need exactly n flips, $\mathbb{E}(\{H = n\})$?

- How likely are all the different possibilities, $\text{kind}(H)$?

As we did earlier, we use $\{H > n\}$ and $\{H = n\}$ to denote the events (0-1-valued FRPs) that H is bigger than n and equal to n .

We model 0 as “tails” and 1 as “heads.” We will assume that the coin has the same chance q of coming up tails on any flip and that the outcome of any flip has no influence on the outcome of any other, i.e., they flips are *independent*.

Think of this as a random system like we discussed in Chapter 7. At any point, we can be in one of two states: we’ve seen a heads or we have not. Call these states HEADS and NO HEADS. The system starts in state NO HEADS. On any flip, if we get a heads, we have made one flip and transition to state HEADS, and if we get a tails, we have made one flip and remain in the same NO HEADS state as before the flip. Here’s the key insight: if we get a tails, the Kind of our *remaining* number of flips until we see a heads is the *same* as $\text{kind}(H)$.

We can make this idea precise with a conditional Kind. Suppose `remaining_flips` is the *Kind* of the number *additional* of flips until a heads *after* the current flip. Then, the Kind of the total number of flips *given* the current flip is

```
conditional_kind({
    0: remaining_flips ^ (__ + 1),
    1: constant(1)
})
```

It takes one flip if we get a heads on the current flip, or one flip – for the current flip – plus the remaining flips required if we get a tails on the current flip.

We write a function `wait_for_heads` that gives the Kind of total number of flips when we have specified the Kind of the remaining number of flips after the current flip.

```
def wait_for_heads(remaining_flips, q=symbol('q')):
    """Returns the Kind of the total number of flips to get a head.

    `remaining_flips` is the kind of the additional number of flips
        required after seeing a tails

    `q` is the probability of getting a tails. [Default: symbol('q')]

    """
```

```

# Make sure q is a symbol or high-precision number
q = as_quantity(q)

# The kind of the current flip
flip = weighted_as(0, 1, weights=[q, 1 - q])

# the kind of the total number of flips given the current flip
flips_given_current = conditional_kind({
    0: remaining_flips ^ (__ + 1),
    1: constant(1)
})

total_flips = flips_given_current // flip
return total_flips

```

We do not know `remaining_flips`, which is what we want to find, but we have a trick up our sleeve. The solution (`kind(H)`) is the Kind `K` that equals `wait_for_heads(K)`. It is a “fixed point” of the function `wait_for_heads`.

We can solve for that fixed point as in Chapter 7, and will below, but first we will compute an approximation. We start with a guess K_0 for `kind(H)` and set $K_1 = \text{wait\_for\_heads}(K_0)$. Then we use K_1 as our next guess, and continue to iterate until the Kinds we get stop changing to numerical precision. The result will be `kind(H)` to good approximation, even though that Kind will have an infinite number of possible values.

In the playground, this looks like

```

guess = constant(1)
guess = wait_for_heads(guess)
guess = wait_for_heads(guess)
guess = wait_for_heads(guess)
guess = wait_for_heads(guess)
guess = wait_for_heads(guess)

```

Each time, we enhance the tree by accounting for an extra possible flip to get what we want, and but only the two branches with the highest values change at each iteration. Compute these one at a time, and look at `guess` at each stage to see how the trees grow.

The sequence of Kinds produced above is what we get using the `iterate` utility in `frplib`. For instance, the third and sixth values of `guess` above equal, respectively, `iterate(wait_for_heads, 2, constant(1))` and `iterate(wait_for_heads, 5, constant(1))`. We can iterate as long as we like: try `iterate(wait_for_heads, 20, constant(1))`. At each iteration, only the largest two branches change.

Using this, we can compute exact probabilities of waiting *more than n flips* and of waiting *exactly n flips*. We iterate a little more than n times, and the part of the tree we need is exact.

```
def wait_more_than(n, q=symbol('q')):
    "Returns probability of waiting more than n flips for heads; q is the weight on tails."
    wait = iterate(wait_for_heads, n + 1, constant(1), q=q)
    probability = E(wait ^ (__ > n))
    return as_scalar( probability )

def wait_exactly(n, q=symbol('q')):
    "Returns probability of waiting more than n flips for heads; q is the weight on tails."
    wait = iterate(wait_for_heads, n + 2, constant(1), q=q)
    probability = E(wait ^ (__ == n))
    return as_scalar(probability)
```

Try calling each of these for a few values, like `wait_more_than(4)`, `wait_more_than(10)`, `wait_exactly(3)`, `wait_exactly(7)` and so forth. You will see that for any n `wait_more_than(n)` returns q^n and `wait_exactly(n)` returns $q^{n-1}(1 - q)$. The latter makes immediate intuitive sense: to first get a heads on the n th flip, we need to get $n - 1$ tails followed by a heads. And because we have an independent join of flips, the probabilities multiply.

When we iterate to find the expectation of the waiting time, it will necessarily be approximate because in principle one can wait an arbitrarily long time to see the first heads. However, the approximation will be very good for reasonable heads because as we've seen, the probability of waiting longer than n flips is q^n which decreases very quickly. For example, `E(iterate(wait_for_heads, 11, constant(1)))` gives

$$1 + q + q^2 + q^3 + q^4 + q^5 + q^6 + q^7 + q^8 + q^9 + q^{10} + q^{11}$$

which is close to $1/(1 - q)$. Indeed, trying it for a few values of n shows the same form suggesting that the exact expectation is $\sum_{j=0}^{\infty} q^j = 1/(1 - q)$, which is 1 over the probability of heads. So if heads come up with probability $1/2$, we expect to wait 2 flips for a heads, if heads has probability $1/1000$, we expect to wait 1000 flips, and so on.

We can find this exactly by solving for the fixed point, the Kind `remaining_flips` that is unchanged by applying `wait_for_heads`. Hence, `remaining_flips` is equal to `wait_for_heads(remaining_flips)`. If we write down the Kind and equate the weights for branches of the same value, we can solve for all the weights. The fact that the Kind has an infinite number of leaves does not cause a problem.

Figures 8.1 and 8.2 show the process. In the first figure: on the left, we specify arbitrary weights for every possible number of flips, and on the right we apply the conditioning operator `wait_for_heads` using the `flips` Kind. The second figure shows the same comparison, reducing the second tree to canonical form. Equating weights for each branch, we get

$$\begin{aligned} p_1 &= 1 - q \\ p_2 &= qp_1 = q(1 - q) \\ p_3 &= qp_2 = q^2(1 - q) \\ &\vdots \quad \vdots \end{aligned}$$

That is, $p_j = q^{j-1}(1 - q)$ for every integer $j \geq 1$, and the Kind we seek is

$$\langle \rangle \begin{array}{l} \dots \dots \dots \\ \text{---} q^5(1 - q) \text{---} \langle 6 \rangle \\ \text{---} q^4(1 - q) \text{---} \langle 5 \rangle \\ \text{---} q^3(1 - q) \text{---} \langle 4 \rangle \\ \text{---} q^2(1 - q) \text{---} \langle 3 \rangle \\ \text{---} q(1 - q) \text{---} \langle 2 \rangle \\ \text{---} 1 - q \text{---} \langle 1 \rangle \end{array}$$

which does indeed have expectation $1/(1 - q)$.

We can take this idea and run with it. The next two examples use the same approach in slightly different situations.

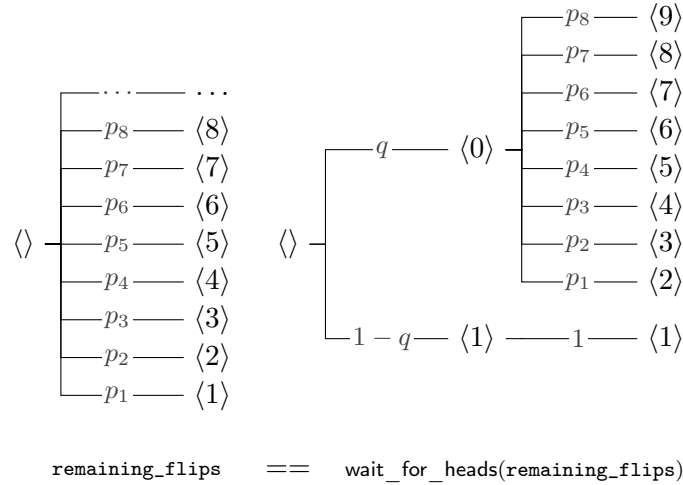


FIGURE 8.1. Solving for `remaining_flips`. On the left is the Kind `remaining_flips` with arbitrary weights assigned to each value. We want to solve for those weights in terms of q . On the right is the value of `wait_for_heads(remaining_flips)` which operates by conditioning on a single flip. (See that function earlier.)

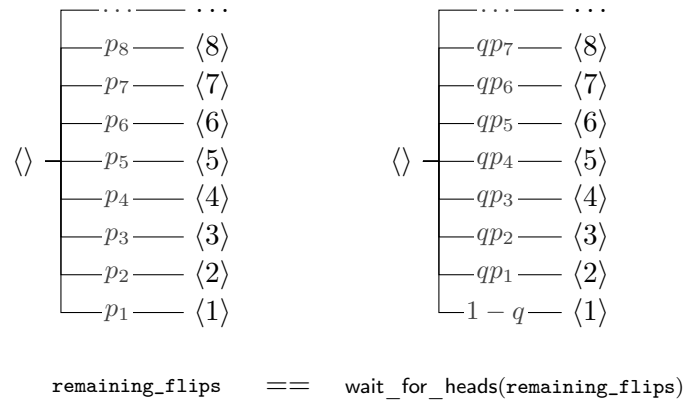


FIGURE 8.2. Solving for `remaining_flips` continued. Here, we reduce the right-hand Kind in Figure 8.2 to canonical form. Making the two Kinds here equal just requires equating the waits on the branches for each value. So, $p_1 = 1 - q$, $p_2 = qp_1$, $p_3 = qp_2$, and so on. We can solve these equations for the p_i 's as shown in the text.

Example 8.14 Double Heads and Other Patterns

What if in the previous example we were not waiting for a single heads to come up but instead waiting for the first appearance of *two consecutive heads*? The approach would be the same, except we have to account for more than two states. In particular, we want to find the sequences that lead us back into the same state we were in at the beginning. That will enable us to solve the equation for the Kind of the total number of flips.

When waiting for two heads, there are *three* possibilities that lead us either to success or back to the state we started in. If we get a heads-heads, then we are done having used two flips. If get a heads-tails, we are back to our original state having used two flips. If we get a tails, we are back to our original state using one flip. Calling these possibilities 0 (tails), 2 (heads-tails), and 3 (heads-heads), the conditional Kind describing this is

```
conditional_kind({
    0: remaining_flips ^ (__ + 1),
    2: remaining_flips ^ (__ + 2),
    3: constant(2)
})
```

which by direct analogy with `wait_for_heads` earlier gives us

```
def wait_for_2heads(remaining_flips, q=symbol('q')):
    """Returns the kind of the total number of flips to get consecutive heads.

    `remaining_flips` is the kind of the additional number of flips
        required after seeing a tails or heads-tails.

    `q` is the probability of getting a tails. [Default: symbol('q')]

    """
    # Make sure q is a symbol or high-precision number
    q = as_quantity(q)

    # The kind of the current prefix flips
    prefix = weighted_as(0, 2, 3, weights=[q, q*(1 - q), (1 - q) * (1 - q)])
```



```

# the kind of the total number of flips given the current flip
flips_given_current = conditional_kind({
    0: remaining_flips ^ (__ + 1),
    2: remaining_flips ^ (__ + 2),
    3: constant(2)
})

total_flips = flips_given_current // prefix
return total_flips

```

Again, we can compute the probability of waiting more than n flips or of waiting exactly n flips. These are

```

E( iterate(wait_for_2heads, n + 1, constant(2)) ^ (__ > n) )
E( iterate(wait_for_2heads, n + 2, constant(2)) ^ (__ == n) )

```

just like before.

Solving the analogous “Kind equation” as before gives us the following.

$$\begin{array}{c}
 \langle \rangle - \left[\begin{array}{l}
 \dots \dots \\
 p_8 \text{---} \langle 8 \rangle \\
 p_7 \text{---} \langle 7 \rangle \\
 p_6 \text{---} \langle 6 \rangle \\
 p_5 \text{---} \langle 5 \rangle \\
 p_4 \text{---} \langle 4 \rangle \\
 p_3 \text{---} \langle 3 \rangle \\
 p_2 \text{---} \langle 2 \rangle
 \end{array} \right. \\
 \mathbf{k}
 \end{array}
 \quad == \quad
 \begin{array}{c}
 \langle \rangle - \left[\begin{array}{l}
 \dots \dots \\
 qp_7 + (1-q)qp_6 \text{---} \langle 8 \rangle \\
 qp_6 + (1-q)qp_5 \text{---} \langle 7 \rangle \\
 qp_5 + (1-q)qp_4 \text{---} \langle 6 \rangle \\
 qp_4 + (1-q)qp_3 \text{---} \langle 5 \rangle \\
 qp_3 + (1-q)qp_2 \text{---} \langle 4 \rangle \\
 qp_2 \text{---} \langle 3 \rangle \\
 (1-q)^2 \text{---} \langle 2 \rangle
 \end{array} \right. \\
 \mathbf{wait\_for\_2heads(k)}
 \end{array}$$

Notice that $p_1 = 0$ has been excluded here as that is not a possible number of flips to get two heads.

Equating weights in these Kinds gives us a recurrence relation that we can solve:

$$\begin{aligned}
 p_2 &= (1-q)^2 \\
 p_3 &= q(1-q)^2 \\
 p_4 &= q^2(1-q)^2 + q(1-q)^3 \\
 &\vdots
 \end{aligned}$$

for all integers $j \geq 2$.

The same idea carries over to other patterns too. For some patterns, like consecutive strings of heads, solving for the Kind reduces a single equation between the Kind. In some cases, like heads-tails-heads-tails, we can express this as a system of “Kind equations.” We will see a general solution in an example in a later chapter.

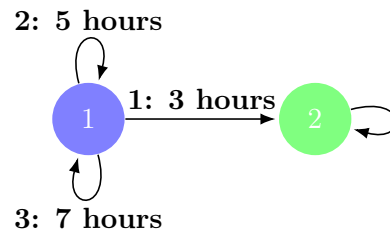


FIGURE 8.3. Graph describing the state machine for the rover.

Example 8.15 Recursive Rover

A robot exploring the Valles Marineris canyon system on Mars finds itself at the end of a canyon with three ancient river channels leading away. If it takes the first channel, it will reach its base site after 3 hours of rocky travel. If it takes the second or third channel, it will travel for 5 or 7 hours respectively only to find itself back where it started. The robot (not a very bright one) always chooses among the channels randomly with equal weights. How long do we predict the rover will take until it reaches safety?

We can think of this as a machine with two states. In state 1, the robot is at the end of the canyon, and in state 2, it has returned successfully to base. From state 1, the robot can take any of the three channels, two return the machine to state 1, one to state 2. These transitions are shown in Figure 8.3, which depicts a graph commonly known as a Finite-State Machine. Each transition from state 1 is labeled with the time the robot requires to make that transition.

Let T and C be FRPs. T represents the # of hours until the robot reaches base, and C 's represents the channel (1, 2, or 3) that the robot chooses *initially*. We want to find $\mathbb{E}(T)$.

We will use the same technique as in the previous two examples. The function `time_to_base` finds the Kind of the robot's total time to base in terms of the Kind of the *remaining time* after the first choice.

```

def time_to_base(t):
    """Returns the conditional kind of time to base.
    Here, t is the *kind* of the remaining time *after the step*.

    """

    base = conditional_kind({1: constant(3),
                             2: t ^ (__ + 5),
                             3: t ^ (__ + 7)})

    channel = uniform(1, 2, 3)

    return base // channel

```

And with this, can approximate $\mathbb{E}(T)$ quite well by iterating on a guess as before:

```

E(iterate(time_to_base, 10, constant(3)))  #== 14.792
E(iterate(time_to_base, 20, constant(3)))  #== 14.996
E(iterate(time_to_base, 30, constant(3)))  #== 14.9999

```

By increasing the number of iterations, we allow for the possibilities of longer sequences by the robot. The expectations converge rather quickly to 15.

We suspect that $\mathbb{E}(T) = 15$. Can we solve this exactly? Yes as before, but here things are even simpler because we only need the expectation. In particular, the Kind of the robot's time to base is the (infinite) Kind $\mathfrak{t}$ such that $\mathfrak{t}$ is *equal to* `time_to_base(t)`, which necessarily means that $\mathbb{E}(\mathfrak{t}) == \mathbb{E}(\text{time\_to\_base}(\mathfrak{t}))$.

But `time_to_base` just does a conditioning operation, conditioning on a `channel` Kind. And $\mathbb{E}(\text{base} // \text{channel})$ is the same as $\mathbb{E}(\text{base}) // \text{channel}$, where $\mathbb{E}(\text{base})$ is a function of the initial channel, which we can view as a conditional Kind that returns a constant Kind for each input. Hence,

$$\begin{aligned}
 \mathbb{E}(t) &= \mathbb{E}(\text{time\_to\_base}(t)) \\
 &= \frac{1}{3} \mathbb{E}(\text{constant}(3)) + \frac{1}{3} (\mathbb{E}(t) + 5) + \frac{1}{3} (\mathbb{E}(t) + 7) \\
 &= 1 + \frac{2}{3} \mathbb{E}(t) + 4 \\
 &= 5 + \frac{2}{3} \mathbb{E}(t).
 \end{aligned}$$

Solving our equation $\mathbb{E}(t) = 5 + \frac{2}{3} \mathbb{E}(t)$ gives us $\mathbb{E}(t) = 15$ as expected.

Note that $E(t \sim (\_ + 5)) = E(t) + 5$ by the Additivity property in equation (6.4) and similarly with 7.

Example 8.16 Central Limit Theorem

For each $n \in [1..)$, let M_n be the FRP with Kind `Mean(either(-1, 1) ** n)`. (We can use `fast_join_pow` to compute this Kind as described earlier, see Puzzle 77.)

We will normalize these FRPs by scaling them in a particular way, defining

$$Z_n = \frac{1}{\sqrt{n}} M_n.$$

Figure 8.4 depicts the Kinds of Z_n for $n \in \{5, 25, 100, 1000\}$. Rather than showing the trees, we show the Kinds more compactly, plotting points $\langle v, w \rangle$ where v is the value of a branch and w is the associated weight.

All of these plots look like the function

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$$

up to a constant factor, and this function is shown in the bottom panel of the Figure for comparison. For even moderate n , the plotted points lie very close to this curve. The Kinds $\text{kind}(Z_n)$ have size growing with n , but the weights are simply described by this curve.

This is a special case of an important result called the **Central Limit Theorem**. The Kind of the normalized means Z_n converges in some sense in a way that lets us approximate the Kind with high accuracy.

Chapter Checkpoints

After reading this Chapter you should be able to:

- Identify the types of the Big 3+1 operations.
- Describe a system and its Kind with parallel expressions.
- Describe Finite Random Processes using joins, statistics, and observations.
- Formulate a strategy for analyzing Finite Random Processes with FRPs and Kinds.

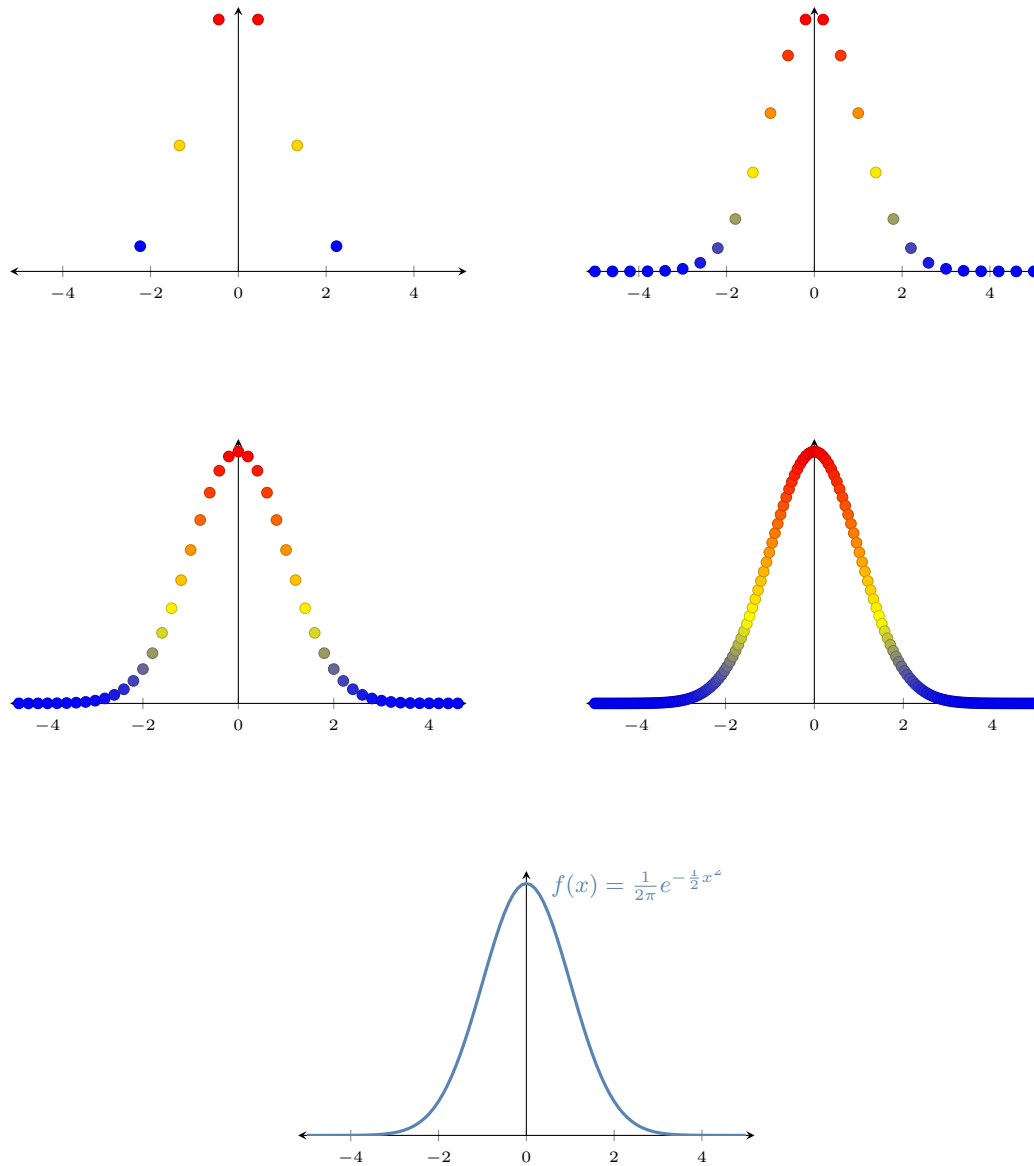


FIGURE 8.4. $\text{kind}(Z_n)$ for $n \in \{5, 25, 100, 1000\}$ in Example 8.16. Each Kind is shown by plotting the weights associated with each value, with one point per branch of the tree. The bottom panel plots the function $f(x)$ for comparison.

- Use the Big 3+1 operations to compute Kinds and predictions.

Interlude F

Functions

Functions are a central mathematical abstraction. We use functions for a huge variety of purposes: to compute, to label, to order, to describe, to summarize, to package, to relate, to probe, and to transform other objects and quantities. They are fundamental and ubiquitous.¹⁰ Here, we take an interlude to understand functions in detail and to develop a language and notation for working with them effectively. More than that, the goal of this interlude is to help you learn to *think* in terms of functions, to use functions as an active tool for framing and solving problems.

The basic idea is simple. A function is a rule that associates each object in a collection of allowed *inputs* to an object in a collection of possible *outputs*, such that ***every input object must map to exactly one corresponding output object***. We say that the function *takes* an allowed input and *returns* an output and that it *maps* an input to its associated output.

Figure 8.5 shows several example functions with simple collections of inputs and outputs. Notice that *an output object may be associated with zero, one, or more than one input objects*. As such, we must distinguish between a function's *possible outputs* and its *actual outputs*. Figure 8.6 shows rules that are *not functions* because some inputs map to more than one or to zero outputs.

We can represent a function in various ways. One way is a two-column table, one each for inputs and outputs, where each input value can appear only once. For example:

Input	Output
1	hello
10	world
100	!

Similarly, we can use a dictionary (aka hash map, associative array) in a programming language like Python: `{1: 'hello', 10: 'world', 100: '!'}`. This table and this dictionary takes 1, 10, or 100 as input and returns the corresponding text strings (hello, world, or !) as output. We can view a function that represents this table or dictionary as a package for the information that the table or dictionary contains, and we can in turn operate on or query that package in a variety of ways.

More often, we represent functions symbolically, with names, operators, and equations. For instance, we can define a function f by $f(x) = 11x + 3$. The variable f names and represents the function itself as a mathematical object. The x is called a *parameter* of the function. It is a local variable that is only defined on the right-hand-side of the $=$ in the defining equation. The name of the parameter is arbitrary,

¹⁰In the words of mathematician Thomas Garrity: "Most of us learned at the feet of our parents the following truth – functions describe the world!" [Garrity2017]

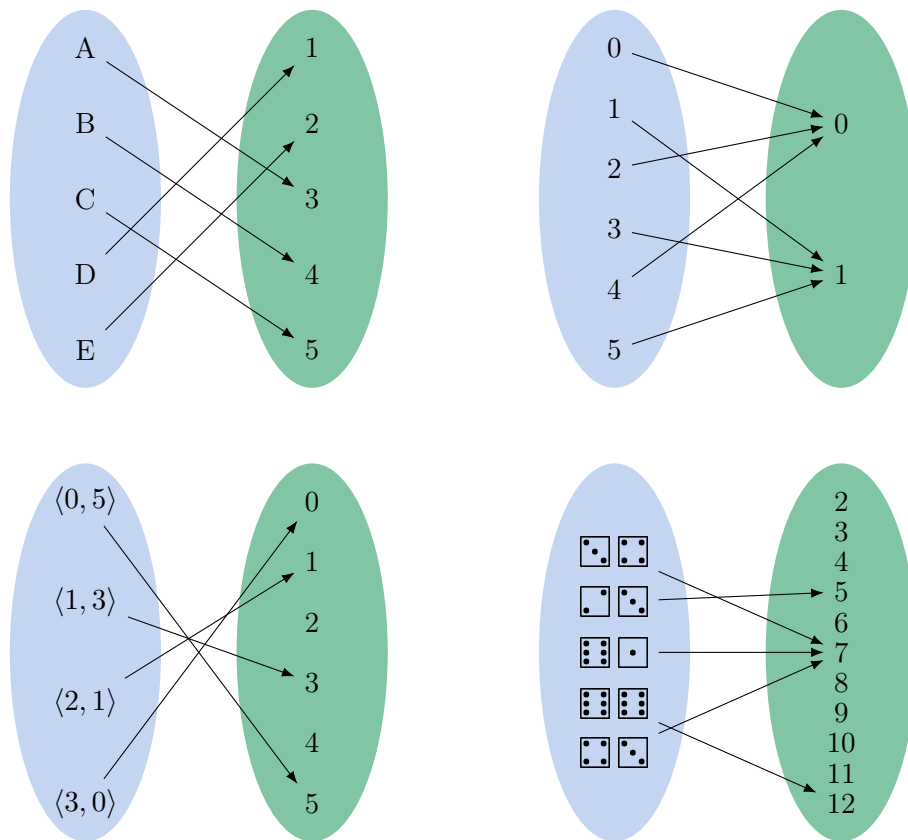


FIGURE 8.5. Examples of simple functions. The arrows connect an input object on the left with its associated output object on the right.

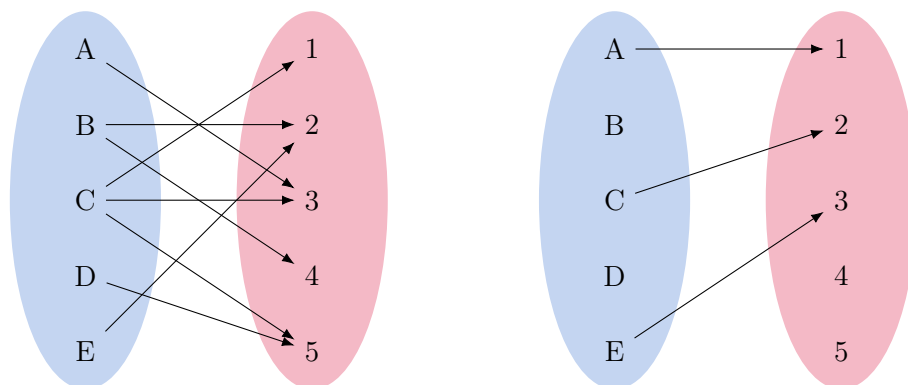


FIGURE 8.6. Mappings that are not functions because some inputs map to more than one output (left) or to no outputs (right). The latter is called a *partial function* because it returns an output for some but not all of the inputs; it can be made into a function defined the a smaller collection of input objects for which it returns a value.

so for instance $f(y) = 11y + 3$ defines the same function. Here, f takes a numeric input, multiplies it by 11, adds 3, and returns the result. When we write $f(4)$, we are *evaluating* the function at the input (aka *argument*) 4, and the definition of the function tells us that $f(4) = 11 \cdot 4 + 3 = 47$.

Most programming languages also have a syntax for defining functions that directly encodes the mapping from input to output. For example, `a_function` defined by

```
def a_function(x: int) -> int:
    return 11 * x + 3
```

takes any valid Python integer (type `int`) as input and returns an `int` (11 times the input `int` plus 3) as output. This function `a_function` is similar to f but not exactly the same, as f can accept any numeric input, not just integers or machine-representable numbers. Defining a function in code can be helpful for understanding how it should be defined or what it means.

Be aware, however, that it is possible to write “functions” in a programming language that do not quite meet our mathematical definition of functions. We require mathematical functions to be *pure*, meaning that they always return the same value given the same input. We also require them to be *total*, meaning that they always return a value. Side effects like printing or random numbers, exceptions or errors, and infinite loops are not allowed. For example, none of the following Python functions represents a mathematical function:

```
def not_function_impure1(x: int) -> int:
    "Has a side effect outside the function, printing the current time."
    print(time.strftime("%H:%M:%S", time.localtime()))
    return 11 * x + 3

def not_function_impure2(x: int) -> int:
    "Returns different outputs for the same input."
    return random.randint(x, x + 11)

def not_function_partial1(x: int):
    raise Exception("Does not return a value")

def not_function_partial2(x: int) -> int:
    "Might never terminate."
    while x > 3:
```

```
pass  
return x
```

The remainder of this interlude formalizes these ideas and dives deeper into examples and applications. We start in Chapter 9 with a brief discussion of sets because sets are the infrastructure on which we build functions. We will see how to define sets and build new sets from other sets, and we will meet some sets that we use frequently.

Chapter 10 presents the foundational ideas and terminology, describes common operations and special functions, and gives lots and lots of examples. Chapters 11 and 12 define two special types of functions and our notation for them: functions without a name – *anonymous functions* – and functions that indicate a binary outcome – *indicator functions*. We use both frequently! And 15 gives conventions and tools that we use for functions that take or return tuples/vectors or multiple arguments, another common case.

Chapters 13 on function composition, 14 on function properties, and 17 on algebraic structure are the conceptual heart of the Interlude. The first highlights an essential recurring theme in our thinking about systems and operations on systems: the power of composability. The second describes key features that functions can have and shows how to recognize and exploit them. The third offers powerful tools with which functions can help us understand the structure of complex objects. In addition, Chapter 16 briefly covers ways to describe relations among objects.

Finally, Chapter 18 is an extended application of all these ideas to counting. It offers a framework that explains answers to many common counting problems in terms of functions.

Sets

9

Chapter

Contents

9.1	Specifying Sets	543
9.2	Increments and Intervals	544
9.3	Making New Sets from Old Ones	545
9.4	To Infinity and Beyond	549

We consider sets briefly in this interlude because they give us the infrastructure on which functions are defined. A function maps elements of one set to elements of another. We will mostly work explicitly with functions rather than sets, but it is nonetheless useful to see how sets are defined and used.

A **set** is a collection of *distinct* objects. Each object in the collection is called an **element** of the set, and the set is said to *contain* its elements. The elements of a set can be any type of object, including numbers, vectors, functions, letters, shapes, . . . , even other sets.¹ A set is determined solely by the objects it contains.

It is often useful to give sets *names* to avoid repeating the definition when we refer to them. Some commonly used sets – such as the empty set or the set of all real numbers ($\mathbb{R}$), integers ($\mathbb{Z}$), natural numbers ($\mathbb{N}$), and Booleans ($\mathbb{B}$) – have special names.² Otherwise, we name sets with *script capital letters* (e.g., $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{L}, \mathcal{S}, \mathcal{X}$), optionally decorated with subscripts ($\mathcal{A}_4$), superscripts ($\mathcal{B}^2$), or other marks ($\tilde{\mathcal{C}}_*$).

Two sets are equal if they both contain the same elements. If $\mathcal{A}$ and $\mathcal{B}$ are sets, we denote equality by $\mathcal{A} = \mathcal{B}$; if they are not equal, we write $\mathcal{A} \neq \mathcal{B}$. The simplest set is the set with no elements, called the **empty set**. This is traditionally denoted by $\emptyset$, though we will favor the more direct notation $\{\}$.

To indicate that an object x is an element of a set $\mathcal{A}$, we write $x \in \mathcal{A}$. The operator $\in$ here is read in any of several equivalent ways: “in”, “element of”, “member of”, or “belongs to.” To indicate the opposite – that y is *not* an element of $\mathcal{A}$ – we write $y \notin \mathcal{A}$.³ The operator $\notin$ is read “not in”, “not an element of”, et cetera. Both operators $\in$ and $\notin$ take an object on the left and a set on the right. We sometimes also use sets to specify *types* of objects. In this case, we write $x : \mathcal{T}$ to indicate that x has “type” $\mathcal{T}$. The just means that x belong to $\mathcal{T}$ but it has a different connotation.

The **cardinality** of a set $\mathcal{A}$ is the number of elements it contains⁴ and is denoted by $\#\mathcal{A}$. For instance, the empty set has $\#\{\} = 0$; the set containing only 1 (which we will denote $\{1\}$) has $\#\{1\} = 1$; and the set containing only 1, 2, 3, and 4 (which we will denote $\{1, 2, 3, 4\}$) has $\#\{1, 2, 3, 4\} = 4$. A basic principle of counting that we will see in Chapter 18 is that two sets have the same cardinality if and only if we can define a particular kind of function between them. The cardinality of a set can be infinite, in multiple ways, which is discussed in Section 9.4.

If every element of a set $\mathcal{A}$ is also an element of a set $\mathcal{B}$, we say that $\mathcal{A}$ is a **subset** of $\mathcal{B}$, which we denote $\mathcal{A} \subseteq \mathcal{B}$. Notice that $\mathcal{A} \subseteq \mathcal{B}$ includes the possibility that $\mathcal{A} = \mathcal{B}$. To show $\mathcal{A} \subseteq \mathcal{B}$, we need only show that an *arbitrary* $x \in \mathcal{A}$ also satisfies $x \in \mathcal{B}$. From the definition of subsets, we have:

- Every set is a subset of itself ($\mathcal{A} \subseteq \mathcal{A}$).

¹There are some limitations. For instance, a set cannot contain itself, so we cannot form the set of all sets. Mathematicians worry about such things.

²See Appendices B and N. The natural numbers $\mathbb{N}$ include 0. $\mathbb{B} = \{\perp, \top\}$, where $\perp$ stands for false and $\top$ for true.

³The slashes through the $\in$ and $=$ mean “no” or “not”, by analogy to, say, a no-left-turn sign.

⁴When someone refers to the “size” of a set, they sometimes mean cardinality. This is fine, but because there are many different notions of a set’s size, precision is sometimes needed.

- The empty set is a subset of every set ($\{\} \subseteq \mathcal{A}$).
- Two sets are equal if and only if each is a subset of the other.

That is, $\mathcal{A} = \mathcal{B}$ is equivalent to $\mathcal{A} \subseteq \mathcal{B}$ and $\mathcal{B} \subseteq \mathcal{A}$.

Conversely, if $\mathcal{A} \subseteq \mathcal{B}$ then $\mathcal{B}$ is a **superset** of $\mathcal{A}$, denoted $\mathcal{B} \supseteq \mathcal{A}$. If $\mathcal{A} \subseteq \mathcal{B}$ but $\mathcal{A} \neq \mathcal{B}$, then $\mathcal{A}$ is a **strict subset**⁵ of $\mathcal{B}$, denoted $\mathcal{A} \subset \mathcal{B}$, and similarly for a strict superset $\mathcal{B} \supset \mathcal{A}$. These subset relations are analogous to $\leq$ and $\geq$ versus $<$ and $>$.

⁵Also known as a *proper subset*

9.1 Specifying Sets

We can *specify* sets in several ways. First, for a small set, we can explicitly list its elements within $\{\}$ delimiters. For example, $\{1\}$ and $\{1, 10, 100\}$ are sets of numbers; $\{\{1\}, \{1, 2, 3\}, \{1, 10, 100\}\}$ is a set of sets; and $\{\text{emu}, \text{frog}, \text{newt}\}$ and $\{?, !, '\}$ are sets of English words and punctuation marks, respectively. The *order* in which we list the elements does not matter, so $\{1, 2, 3\}$ and $\{3, 1, 2\}$ define the same as set. And if we were to repeat an element in the list, the repetition would be redundant, i.e., $\{1, 1, 2, 3\}$ is the same as $\{1, 2, 3\}$.

Second, we can specify a set by the properties of its elements, using what is called “set builder notation.” This follows the template of

$$\{\langle \text{qualified local variable} \rangle \mid \langle \text{condition} \rangle\}. \quad (9.1)$$

Here, $\langle \text{qualified local variable} \rangle$ is either a variable name (like “ u ”) or a variable name with a restriction indicating a containing set (like “ $u \in \mathbb{R}$ ” or “real u ”); $\langle \text{condition} \rangle$ is a Boolean expression⁶ indicating properties of the local variable; and the $\mid$ separator is read as “such that” or “where” or “given”. We use $\wedge$ for logical “and” and $\vee$ for logical “or” in $\langle \text{condition} \rangle$. For example, $\{u \mid u > 0 \wedge u^2 - 3u > 2\}$ is the set of positive real numbers that satisfy the specified quadratic constraint.⁷ Here u is a local variable defined only within the $\{\}$ delimiters. Another example:

$$\{\text{integer } k \mid k \geq 0 \wedge \text{there is an integer } j \text{ such that } k = 3j + 1\},$$

which we can write equivalently as $\{k \in \mathbb{N} \mid k \bmod 3 = 1\}$, is the set of non-negative integers with remainder 1 when divided by 3. The former specification uses two local variables, with different scopes illustrated by boxes in the following

$$\left\{ \text{integer } k \mid \boxed{k \geq 0 \wedge \text{there is an integer } j \text{ such that } \boxed{k = 3j + 1}} \right\},$$

⁶A Boolean expression uses operators $\wedge$ for logical “and” and $\vee$ for logical “or”.

⁷Unless otherwise stated, we take numeric variables to be real numbers, so we need not write “real u ” here.

where k is defined in the outer box and j in the inner box. The set

$$\{\langle a, b, c \rangle \mid a^2 + b^2 + c^2 \leq 1\}$$

is the set of three-dimensional real vectors with length no greater than one.⁸ Here, we “destructure” the set’s elements to define the set in terms of its components. The set

$$\{\mathcal{A} \mid \mathcal{A} \subseteq \mathbb{N} \wedge \#\mathcal{A} \leq 4\}$$

is the set whose elements are sets of natural numbers of cardinality no bigger than 4.

Third, we can specify a set through operations on other sets, such as union and intersection in Section 9.3, or with functions as discussed later in this Interlude. For example, for sets $\mathcal{A}$ and $\mathcal{B}$ and natural number n , $\mathcal{A} \cup \mathcal{B}$ is the set containing the elements of both $\mathcal{A}$ and $\mathcal{B}$, and $\binom{\mathcal{A}}{n}$ is the set of subsets of $\mathcal{A}$ of size n .

⁸See Chapter 15 and Section 17.3 for more on tuples and vectors.

Puzzle 87. We have that $1 \in \{1, 2, 3\}$ and $2 \notin \{\}$. Fill in the blanks in the following with $\in$ or $\notin$:

1. $\pi \_ \{x \mid \sin(x) < 0.1\}$,
2. $1 \_ \{\{1\}, \{1, 2\}, \{1, 2, 3\}\}$,
3. $\{\} \_ \{\{\}\}$.

Explain why $\#\{\{1\}, \{1, 2\}, \{1, 2, 3\}\} = 3$. What is $\#\{\{\}\}$?

Which of the following are subsets of $\{\langle x, y \rangle \mid x^2 + y^2 \leq 1\}$?

1. $\{\langle x, 0 \rangle \mid 0 \leq x \leq 1\}$
2. $\{\langle y, 1 \rangle \mid -1 \leq y \leq 0\}$
3. $\{\langle u, u \rangle \mid -1/\sqrt{2} \leq u \leq 1/\sqrt{2}\}$

Puzzle 88. Let $\mathcal{A} = \{\text{integer } m \mid \text{there is an integer } n \text{ with } m = 2n - 37\}$ and $\mathcal{B} = \{\text{integer } k \mid k \text{ odd}\}$. Argue that $\mathcal{A} = \mathcal{B}$ by showing both $\mathcal{A} \subseteq \mathcal{B}$ and $\mathcal{B} \subseteq \mathcal{A}$. Start with an *arbitrary* $a \in \mathcal{A}$ and show that it is in $\mathcal{B}$ and with an *arbitrary* $b \in \mathcal{B}$ and show that it is in $\mathcal{A}$.

9.2 Increments and Intervals

We often need to refer to numeric ranges, so it is useful to have an easy and meaningful notation for them. We distinguish between contiguous sequences of integers that we call **increments** and contiguous ranges of real numbers that we call **intervals**. The notation is described in Table 9.1. We use $\dots$ to evoke a discrete sequence and $\_$ to evoke a continuous range. We use hard brackets $[]$ to indicate that an endpoint is

included, and parentheses $()$ to indicate that an endpoint is *excluded*. So $[4..8]$ is the set of integers 4, 5, 6, 7, and 8 and $[0\_10)$ is the set of real numbers ≥ 0 and < 10 . If an endpoint is missing on an open end of an increment or interval, then the increment or interval extends over all values in that direction. For example, $[0\_)$ is the interval containing all non-negative real numbers and $(4..)$ is the increment containing all integers greater than 4. It is also acceptable to use $-\infty$ or ∞ as endpoints if needed, such as to show that one of these is included in the set.

Increments

Closed	$[i..j]$	Set of integers n with $i \leq n \leq j$
Open	$(i..j)$	Set of integers n with $i < n < j$
Half-Closed	$[i..j)$	Set of integers n with $i \leq n < j$
	$[i..)$	Set of integers n with $i \leq n$
Half-Open	$(i..j]$	Set of integers n with $i < n \leq j$
	$(..j]$	Set of integers n with $n \leq j$

Intervals

Closed	$[a\_b]$	Set of real numbers x with $a \leq x \leq b$
Open	(a_b)	Set of real numbers x with $a < x < b$
Half-Closed	$[a\_b)$	Set of real numbers x with $a \leq x < b$
	$[a\_)$	Set of real numbers x with $a \leq x$
Half-Open	$(a\_b]$	Set of real numbers x with $a < x \leq b$
	$(\_b]$	Set of real numbers x with $x \leq b$

TABLE 9.1. Notation for increments and intervals. Peruse these as they will be used regularly.

The interval $[0\_1]$ arises often enough that we give it a name – the **unit interval**. The natural numbers $\mathbb{N} = [0..)$ and the positive integers are $[1..)$.

9.3 Making New Sets from Old Ones

UNIONS. The word “union” implies a bringing together, and that is exactly what you should think when considering the union of sets. If $\mathcal{A}$ and $\mathcal{B}$ are sets, then their **union**, denoted $\mathcal{A} \cup \mathcal{B}$, is the set containing the elements that belong to either $\mathcal{A}$ or $\mathcal{B}$:

$$x \in \mathcal{A} \cup \mathcal{B} \text{ if and only if } x \in \mathcal{A} \text{ or } x \in \mathcal{B}. \quad (9.2)$$

The “or” here is a logical **or**; it is true if x is in either or both sets. Because no element of a set is repeated, an object that belongs to both $\mathcal{A}$ and $\mathcal{B}$ still appears only once in the union. Hence, $\{a, b, c\} \cup \{c, r, g\} = \{a, b, c, r, g\}$ and $\{u \mid 0 \leq u \leq 2\} \cup \{u \mid 1 \leq u \leq 4\} = \{u \mid 0 \leq u \leq 4\}$. Seeing “or” in the definition of a set is a clue that the set is built from a union.

We can take the union of any collection of sets, and this operation is both commutative and associative, i.e., $\mathcal{A} \cup \mathcal{B} = \mathcal{B} \cup \mathcal{A}$ and $(\mathcal{A} \cup \mathcal{B}) \cup \mathcal{C} = \mathcal{A} \cup (\mathcal{B} \cup \mathcal{C})$. If we have a collection of sets $\mathcal{A}_i$ indexed by i in a set $\mathcal{I}$, the union of all the sets is written $\bigcup_{i \in \mathcal{I}} \mathcal{A}_i$. When $\mathcal{I}$ is an increment, $[m \dots n]$ or $[m \dots)$, we often write the endpoints above and below like $\bigcup_{i=m}^n \mathcal{A}_i$ or $\bigcup_{i=m}^{\infty} \mathcal{A}_i$, respectively.

Definition 42. Unions. For a collection of sets $\mathcal{A}_i$, indexed by a set $\mathcal{I}$, finite or infinite, the **union** of the sets is defined by

$$x \in \bigcup_{i \in \mathcal{I}} \mathcal{A}_i \text{ if and only if } x \in \mathcal{A}_i \text{ for some } i \in \mathcal{I}. \quad (9.3)$$

For a small, finite collection, we also write a union with $\cup$ as a commutative, associative, infix operator, *e.g.*, $\mathcal{A} \cup \mathcal{B}$ or $\mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$.

Puzzle 89. What is $\bigcup_{i=0}^4 \{2^i, 2^{i+1}, 2^{i+2}\}$? Is $0 \in \bigcup_{j \in [1..)} [\frac{1}{j+1} - 1)$? What is this union?

INTERSECTIONS. At the intersection of two roads is a patch of ground that is common to both of them; this is the image underlying the intersection of sets. If $\mathcal{A}$ and $\mathcal{B}$ are sets, then their **intersection**, denoted $\mathcal{A} \cap \mathcal{B}$, is the set containing those elements that belong to both $\mathcal{A}$ and $\mathcal{B}$:

$$x \in \mathcal{A} \cap \mathcal{B} \text{ if and only if } x \in \mathcal{A} \text{ and } x \in \mathcal{B}. \quad (9.4)$$

The “and” here is a logical **and**; it is true if x is in both sets. Seeing “and” in the definition of a set is a clue that the set is built from an intersection.

As with unions, we can take the intersection of any collection of sets, and the operation is commutative and associative. We use a similar notation $\bigcap_{i \in \mathcal{I}} \mathcal{A}_i$ et cetera, as well. If $\mathcal{A}$ and $\mathcal{B}$ are sets with $\mathcal{A} \cap \mathcal{B} = \{\}$, then the two sets have *no elements in*

common. We say that these sets are *disjoint*.

Definition 43. Intersections. For a collection of sets $\mathcal{A}_i$, indexed by a set $\mathcal{I}$, finite or infinite, the **intersection** of the sets is defined by

$$x \in \bigcap_{i \in \mathcal{I}} \mathcal{A}_i \text{ if and only if } x \in \mathcal{A}_i \text{ for all } i \in \mathcal{I}. \quad (9.5)$$

For a small, finite collection, we also write an intersection with $\cap$ as a commutative, associative, infix operator, *e.g.*, $\mathcal{A} \cap \mathcal{B}$ or $\mathcal{A} \cap \mathcal{B} \cap \mathcal{C}$.

Two sets with empty intersection, i.e., $\mathcal{A} \cap \mathcal{B} = \{\}$, are said to be **disjoint**.

Puzzle 90. What is the intersection of a set with itself? If $\mathcal{A} \subseteq \mathcal{B}$, what is $\mathcal{A} \cap \mathcal{B}$? What is $[a\_]\cap(\_b]$ when $a < b$? Explain why $\bigcap_{j=1}^{\infty} [\frac{1}{j+1}-1) = [\frac{1}{2}-1)$.

Analogously to addition and multiplication, union and intersection *distribute* over each other:

$$\mathcal{A} \cap (\cup_i \mathcal{B}_i) = \cup_i (\mathcal{A} \cap \mathcal{B}_i) \quad (9.6)$$

$$\mathcal{A} \cup (\cap_i \mathcal{B}_i) = \cap_i (\mathcal{A} \cup \mathcal{B}_i). \quad (9.7)$$

CARTESIAN PRODUCTS. A **tuple** is a finite list of objects indexed by their position in the list. We denote tuples with a comma-separated list delimited by $\langle \rangle$. For example, $\langle 0, 1 \rangle$, $\langle 1, 0 \rangle$, and $\langle 1, 1, 8 \rangle$ are tuples of numbers. Note that $\langle 0, 1 \rangle$ and $\langle 1, 0 \rangle$ are not equal as tuples. In contrast to sets, *order matters* and *repetition is allowed*. The entries in the list are called the tuple's **components**, and the number of components is the tuple's **dimension**. In mathematical use, we typically 1-index the components of the tuple, unlike in most programming languages. So, if x is a tuple of dimension d , then x_1 is the first component, x_2 the second, and so forth up to x_d , and we can write $x = \langle x_1, x_2, \dots, x_d \rangle$. A tuple with dimension d is called a d -tuple for short.⁹ See Chapter 15 and Section 17.3 or more on tuples and vectors.

Here, we consider the **Cartesian product** of sets, *product* for short, which is used to construct sets of tuples from the sets containing the components. If $\mathcal{A}$ and $\mathcal{B}$ are sets, then the product

$$\mathcal{A} \times \mathcal{B} = \{ \langle a, b \rangle \mid a \in \mathcal{A} \wedge b \in \mathcal{B} \}$$

⁹2-tuples and 3-tuples are often called pairs and triples, respectively.

is the set of 2-tuples whose first component belongs to $\mathcal{A}$ and whose second component belongs to $\mathcal{B}$. The same idea works for any natural number of sets.

Definition 44. Cartesian Products. The **Cartesian product** of sets $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ is the set of n -tuples defined by:

$$\mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_n = \{ \langle a_1, a_2, \dots, a_n \rangle \mid a_i \in \mathcal{A}_i \text{ for all } i \in [1..n] \}. \quad (9.8)$$

If $\mathcal{A}_1 = \mathcal{A}_2 = \cdots = \mathcal{A}_n = \mathcal{A}$, we use a shorthand for the product, writing

$$\mathcal{A}^n = \overbrace{\mathcal{A} \times \mathcal{A} \times \cdots \times \mathcal{A}}^{n \text{ times}}. \quad (9.9)$$

where the exponent indicates the dimension of the resulting tuples.

We define $\mathcal{A}^0$ for any $\mathcal{A}$ to be the set $\mathbb{U}$ containing only the empty tuple.

We will see an alternative view of Cartesian products via functions in Example 13.6.

DISJOINT UNIONS. A disjoint union¹⁰ aggregates multiple sets into a new set whose elements are the original elements together with a label indicating what constituent set the original element came from. It is like a union, but it keeps elements from different sets distinct even if they are equal. For example, given a set $\mathcal{A}$, the disjoint union $\mathcal{A} \sqcup \mathcal{A}$ joins two distinct copies of $\mathcal{A}$: it has elements of the form $\langle a, 1 \rangle$ and $\langle a, 2 \rangle$ for each $a \in \mathcal{A}$. The first component of the tuple indicates the value and the second component labels which copy that value comes from. The 1 and 2 here are arbitrary; they need merely be distinct. Similarly, if $\mathcal{Z} = \{0\}$ and $\mathcal{S} = \{\text{caribou}, \text{wolf}, \text{bear}\}$, then $\mathcal{Z} \sqcup \mathcal{S} = \{ \langle 0, 1 \rangle, \langle \text{caribou}, 2 \rangle, \langle \text{wolf}, 2 \rangle, \langle \text{bear}, 2 \rangle \}$ is a set that we think of as containing both 0 and the three strings.

¹⁰Also known as a set *coproduct* and as a *discriminated union*.

Definition 45. Disjoint Union. For a collection of sets $\mathcal{A}_i$, indexed by a set $\mathcal{I}$, finite or infinite, the **disjoint union** of the sets is defined by

$$\bigsqcup_{i \in \mathcal{I}} \mathcal{A}_i = \{ \langle a, i \rangle \mid a \in \mathcal{A}_i \wedge i \in \mathcal{I} \}. \quad (9.10)$$

For a small, finite collection, we also write a disjoint union with $\sqcup$ as a commutative, associative, infix operator, *e.g.*, $\mathcal{A} \sqcup \mathcal{B}$ or $\mathcal{A} \sqcup \mathcal{B} \sqcup \mathcal{C}$, using labels 1, 2, $\dots$ as needed.

In practice, when the constituent set of an element can be uniquely identified,

as with $\mathcal{Z}$ and $\mathcal{S}$ above, it is common to treat the label more informally, letting the elements themselves identify their original set rather than using an explicit tuple.

DIFFERENCES AND COMPLEMENTS. Given two sets $\mathcal{A}$ and $\mathcal{B}$, the **difference** $\mathcal{A} - \mathcal{B}$ is the set of elements of $\mathcal{A}$ that are not in $\mathcal{B}$:

$$x \in \mathcal{A} - \mathcal{B} \text{ if and only if } x \in \mathcal{A} \text{ and } x \notin \mathcal{B}. \quad (9.11)$$

Notice that this definition treats $\mathcal{A}$ and $\mathcal{B}$ asymmetrically: just as in a numerical subtraction, the order matters.¹¹

Sometimes in context it is understood that we are working exclusively with subsets of some larger “universe” set. In that case, differences from that universe set correspond to a logical “not” operation, and it is convenient to avoid repeating the universe set again and again. If $\mathcal{U}$ is the current universe set $\mathcal{A} \subseteq \mathcal{U}$, then the complement of $\mathcal{A}$ is just $\mathcal{U} - \mathcal{A}$, denoted by $\mathcal{A}^c$. For example, when dealing with real numbers, $\{u \mid u \geq 0\}^c = \{u \mid u < 0\}$. When working with integers, the complement of the set of even integers is the set of odd integers. It follows that $\mathcal{A} \cap \mathcal{A}^c = \{\}$ and $\mathcal{A} \cup \mathcal{A}^c = \mathcal{U}$. If the universe set is not clear enough from context, we can always just use differences.

Differences interact with unions and intersections according to what are called *DeMorgan’s Laws*:

$$\mathcal{B} - \bigcup_{i \in \mathcal{I}} \mathcal{A}_i = \bigcap_{i \in \mathcal{I}} (\mathcal{B} - \mathcal{A}_i) \quad (9.12)$$

$$\mathcal{B} - \bigcap_{i \in \mathcal{I}} \mathcal{A}_i = \bigcup_{i \in \mathcal{I}} (\mathcal{B} - \mathcal{A}_i). \quad (9.13)$$

To understand these equations, try to show that the set on each side of the equality is a subset of the other.

9.4 To Infinity and Beyond

Many commonly used sets have *infinite* cardinality, like the set of integers and the set of real numbers. With infinite sets, cardinality can be somewhat counter-intuitive. For example, the set of all integers and the set of even integers have the *same cardinality*. This follows from a basic counting principle:¹² if we pair integer k with even integer $2k$, then every integer has exactly one even partner, and vice versa. On the other hand, the set of all integers and the set of all real numbers *do not* have the same

¹¹A related operation for which order does not matter is called the symmetric difference. The symmetric difference contains those points in either but not both of the sets. This is the set equivalent of the logical exclusive-or.

¹²One-to-one correspondence. See Section 14.1 and Chapter 18.

cardinality.¹³ Both sets have infinite cardinality, but they are *different infinities*.

We say that a set is *countable* if it can be put in one-to-one correspondence with a subset of the integers. Put another way, every element of a countable set can be labeled with a distinct integer. A countable set is either finite or *countably infinite*. All finite sets are countable, as are the set of integers and the set of rational numbers.¹⁴ A set that is not countable is called *uncountable*. The set of real numbers is uncountable, as are the set of all real numbers between 0 and 1 and the set of all sets containing only integers.

This distinction between the countable and uncountable will be important in probability theory. For countable sets, we can assign a non-zero probability to every element in the set and have the total probability bounded. But for uncountable sets that is not possible, so we are left to “smear” the probability across a continuum with no probability “on” any one point. Assigning probabilities to countable and uncountable sets is thus often labeled *discrete* or *continuous*, respectively.

¹³This follows from the famous *Cantor diagonal argument*, which also shows that a set's cardinality is strictly less than the cardinality of the set of its subsets. See B.1.

¹⁴Fractions of the form p/q in lowest terms, for integers p and q .

Function Foundations

10

Chapter

Contents

10.1 Common Operations	580
10.2 Special Functions	585
10.3 Operators	585

A function associates to every object in one set of things some object in another set of things. While the reader will have seen functions before, the way they were described in earlier math classes often obscures the possibilities of what functions can represent or how they can be used. We use functions to describe transformations of objects and data, to label objects in a collection, to select objects from a collection, to package disparate information into a unified object that we can manipulate and study, to classify objects in a collection by various attributes, to probe complex relationships that highlight particular features, to summarize complicated data or objects to gain insight, to encode relationships between objects, to represent data structures, and to equip sets of objects with meaningful mathematical structure. For functions are to do all these things, we must let go of some habits from those early math classes.

You have probably been asked to do something like “graph the function $y = x^2$.” For different values x of the horizontal coordinate, you compute the vertical coordinate $y = x^2$ and mark the resulting point. You then draw the curve connecting these points. The graph is the set of points $\langle x, x^2 \rangle$ for all real numbers x . This implicitly maps each real number x to an associated real number x^2 , giving the idea of a *function* but only in a narrow sense.

We will not tie a function’s definition to specific coordinate names (as with $y = x^2$), nor will we restrict our functions to just taking and returning numbers. We get great value from using functions that take and return other kinds of objects, including sets, graphs, and even other functions. So, we will pay attention to a function’s possible inputs and outputs – these define its *type*. We will think of functions as mathematical objects in their own right that we can name (or not) and operate on. For example, we could define the squaring function above as $s: \mathbb{R} \rightarrow \mathbb{R}$ with $s(x) = x^2$, indicating that s is a function that takes and returns real numbers and describing how the return value is determined from the input value. One operation we can use on the function s is to *evaluate it* at some value x , which we usually write as $s(x)$, but note that s and $s(x)$ are different kinds of objects. The graph of a function is also a derived representation, not the object itself.

Before we formalize our language and notation, it is illustrative to consider some examples. We start with examples of functions “in the wild” and then define several functions that we will refer to throughout this section, and in some cases beyond.

Example 10.1 Encryption

Encryption is the process of encoding a document (a string of bits of arbitrary length) into a form that is very difficult to read without access to a secret “key,”

which is usually a fixed-sized string of bits. An encryption algorithm produces a *matched pair of functions for each specific key*, called an encoder and decoder, both of which take a document as input and return a document as output. When the *encoder* function is given an original document (the plaintext), it returns an encrypted version (the ciphertext). When the *decoder* is given a ciphertext document encrypted with the matching encoder, it returns the original plaintext. The algorithm is designed so that it is computationally very difficult to reconstruct the plaintext from the ciphertext unless you know the key.

Set	Meaning	Elements
$\mathcal{T}$	Possible topics	Either a string or a pair of strings $\langle \text{parent}, \text{child} \rangle$
$\mathcal{P}$	Pages in the book	Natural numbers but may include prefatory pages like ix
$\mathcal{A}$	Possible page annotations	Specified strings, includes a none marker for no annotation
$\mathcal{E}$	Example identifiers in the book	Numeric strings, e.g., 10.2
$2^{\mathcal{P}}$	All subsets of $\mathcal{P}$	A set of page numbers

TABLE 10.1. Notation describing the entities in Example 10.2.

Example 10.2 Index in a Book

The index in a book tells you where to look to find discussion of various topics. An index is a *function* that maps a finite set of *topics* to a set of *sets of pages*. The function takes a topic as input and returns a set of pages on which the topic is mentioned. In practice, it is somewhat more complicated as indexes often contain cross-references or other annotations instead of pages for some topics. Topics may also be hierarchically defined.

We can specify the sets of inputs and possible outputs for the index function. The notation for the sets comprising these inputs and outputs is summarized in Table 10.1. Let $\mathcal{T}$ be the set of topics in the book. A topic is either a string t or a 2-tuple of strings $\langle p, t \rangle$ where p is the parent topic and t the specific topic listed under the parent. (Parents can have their own page references.) Let $\mathcal{P}$ be the set of pages in the book and let $2^{\mathcal{P}}$ denote the set of all subsets of $\mathcal{P}$. Then,

we can define the index as a function i that maps a topic t to *either* a subset of page numbers on which the topic is mentioned. Formally, the function i maps elements of $\mathcal{T}$ into the set $2^{\mathcal{P}} \sqcup \mathcal{T}$. The disjoint union here means that for any topic, i returns *either* a subset of pages *or* a cross-reference topic, as labeled. So, we might have, with string `labels` for the disjoint union,

$$\begin{aligned} i(\text{set}) &= \langle \{5\}, \text{pages} \rangle \\ i(\langle \text{set}, \text{disjointunion} \rangle) &= \langle \{12, 97, 254\}, \text{pages} \rangle \\ i(\langle \text{subsets}, \text{all} \rangle) &= \langle \langle \text{set}, \text{power} \rangle, \text{xref} \rangle. \end{aligned}$$

The first component of each tuple is the content of the index, and the second labels what type of entry it is.

The index in *this* book adds one more wrinkle. Some pages in the index are annotated to describe the type of reference, like **bold** for a definition, a * for an important reference, italic for use in a calculation, and an “m” suffix for reference in a marginal note. Example numbers are also allowed as index references. Let $\mathcal{A}$ be the set of possible page annotations, including `def` for definitions and `none` for a non-annotated page reference, and let $\mathcal{E}$ be the set of example identifiers in the book. Then for this book, i maps $\mathcal{T}$ into the set $2^{(\mathcal{P} \times \mathcal{A}) \sqcup \mathcal{E}} \sqcup \mathcal{T}$. For each topic, we get either a set containing a collection of possibly annotated pages and/or example numbers or a cross-reference topic. For this version of i , we might have, with `annotations` and with labels for the `inner` and `outer` disjoint unions,

$$\begin{aligned} i(\langle \text{subsets}, \text{all} \rangle) &= \langle \langle \text{set}, \text{power} \rangle, \text{xref} \rangle \\ i(\langle \text{graph}, \text{directed} \rangle) &= \langle \{ \langle \langle 37, \text{def} \rangle, \text{pages} \rangle, \langle \langle 39, \text{none} \rangle, \text{pages} \rangle, \langle F.2.18, \text{example} \rangle \}, \text{loc} \rangle \end{aligned}$$

where labeled values indicate what they represent. (The last one is a bit tricky to parse, but take it from the inside out.)

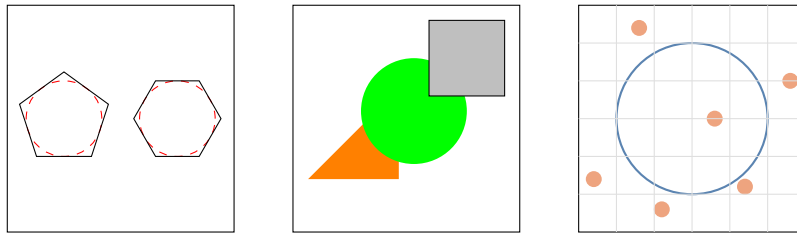


FIGURE 10.1. Several examples of scalable vector graphs from Example 10.3.

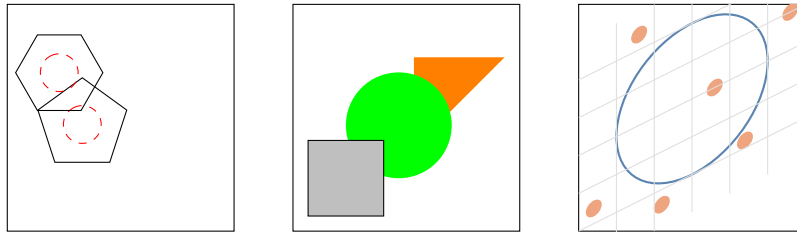


FIGURE 10.2. Transforms applied to the images in Figure 10.1, from Example 10.3.

Example 10.3 Scalable Vector Graphics

Two common types of images are

- raster images (as stored in gif, jpeg, or png files), which are composed of a grid of small tiles called pixels, and
- vector images (as stored in svg, eps, or pdf files), which are composed of shapes and other geometric primitives.

Raster images depend on the resolution of the pixel grid but are efficient for storing highly variable images such as photographs. Vector images are arbitrarily scalable, easy to transform, and can be converted to raster images at a specified resolution.

Figure 10.1 shows three vector images. Each image is represented as a list of shapes painted from bottom to top on an infinite canvas; each shape in the list can itself comprise a list of other, more primitive shapes. Although the canvas is infinite, we see only a finite region because the view is restricted to a rectangular region specified for each image. (We treat the view rectangle as a special shape painted on top that clips the other shapes to the specified region.)

Given a vector image, we can transform it in a variety of ways. Figure 10.2 shows transformations of the images in Figure 10.1. In the left image, the two polygons are translated to new positions, and the two circles are translated and scaled. In the middle image, all the shapes are reflected horizontally and vertically around the center. In the right image, the shapes are sheared by a linear transformation and the view rectangle is shrunk slightly.

Let $\mathcal{S}$ be the set of possible primitive shapes (points, circles, polygons, piecewise linear paths, parametrized curves, ...) on a coordinate system $\mathcal{X}$. Then we can describe an image I as a tuple $I = \langle s_1, s_2, \dots, s_n, v \rangle$, where each s_i is either in $\mathcal{S}$ or is itself a list of shapes and where v is the view rectangle. A transformation T of a vector image $I = \langle s_1, s_2, \dots, s_n, v \rangle$ is a list $T = \langle f_1, f_2, \dots, f_n, g \rangle$ where

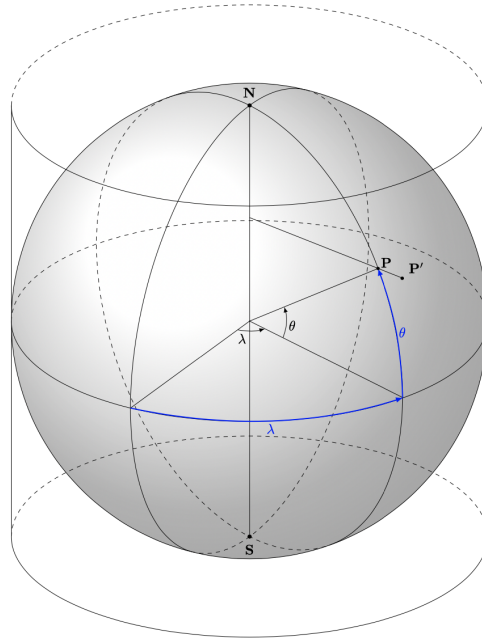


FIGURE 10.3. A cylindrical map projection of a spherical Earth. This particular one is the Lambert equal-area projection. The point $\mathbf{P}$ on the sphere projects to point $\mathbf{P}'$ on the cylinder from the point of equal height on the line from North pole to South.

- (i) if $s_i \in \mathcal{S}$, then $f_i: \mathcal{S} \rightarrow \mathcal{S}$ transforms one primitive shape to another; (ii) if s_i is a list of shapes, f_i is a list of functions $\mathcal{S} \rightarrow \mathcal{S}$ of the same dimension; (iii) g is a function on $\mathcal{X}$ that can only combine translation and scaling and is used to move and resize the view rectangle.

Example 10.4 Map Projections

The Earth is, famously, round. One implication of this is that any map of the Earth on a flat surface necessarily introduces some kind of distortion. A *cartographic map projection* is a function from coordinates on all or part of the Earth's surface (which for this example we approximate by a sphere) to coordinates on the plane (the flat map). Different projections reflect different choices of how the map is distorted.

Figure 10.3 illustrates a *cylindrical* projection in which points on the Earth are mapped to points on a cylinder that is wrapped around the sphere. When the cylinder is unwrapped, we get a flat map. The projection in the Figure is the Lambert equal-area projection. It preserves the area in the sense that the area of

a region on the Earth equals the area of the region it maps to. But the Lambert projection distorts angles, increasingly toward the poles. In contrast, the famous Mercator projection – a different cylindrical projection used on most common maps of the Earth – preserves *angles* but distorts areas. Africa, for instance, is about 14 times the size of Greenland although a Mercator map makes them seem the same size.

If we measure coordinates on the Earth by latitude θ and longitude λ , in degrees, a map projection is a function from $[-90\_90] \times [0\_360)$ to $\mathbb{R} \times \mathbb{R}$ that takes a position on the Earth and returns a position on the flat map. A cylindrical projection P is a map projection with the particular form $P(\theta, \lambda) = \langle \lambda, y(\theta) \rangle$ for some function y taking $[-90\_90] \rightarrow [-\infty\_ \infty]$. For example:

$$P(\theta, \lambda) = \langle \lambda, \sin(\theta) \rangle \quad (\text{Lambert Projection})$$

$$P(\theta, \lambda) = \langle \lambda, \ln(\tan(45 + \theta/2)) \rangle. \quad (\text{Mercator Projection})$$

Figure 10.4 compares the two projections for a map of the Earth. Both projections convert longitude (λ) directly to the horizontal coordinate on the flat map as seen in Figure 10.3 where circles of constant latitude project to horizontal circles on the cylinder. Notice that the vertical coordinate of the Mercator projection grows without bound as we approach either pole.

Given coordinates on the flat map, we can convert them back to the latitude and longitude, with the functions

$$P^{-1}(x, y) = \langle \arcsin(y), x \rangle \quad (\text{Lambert Projection})$$

$$P^{-1}(x, y) = \langle 2 \arctan(e^y) - 90, x \rangle \quad (\text{Mercator Projection})$$

that undo or “invert” the projection.

The properties of the projections are embodied by the function P . For instance, the area distortion at $\langle \theta, \lambda \rangle$ equals $|y'(\theta)|/\cos(\theta)$, giving 1 for Lambert and $1/\cos(\theta)$ for Mercator, where y' is the derivative of the function y . Similarly, P preserves angles if $\cos^2(\theta) |y'(\theta)|^2 = 1$, which holds for Mercator but not for Lambert.

Example 10.5 Voting Patterns. Imagine tallying the results of an election in a single district. Let $\mathcal{R} = [1..m]$ index the set of races/ballot initiatives in the election, in some arbitrary order. For each race r , there is a set $\mathcal{C}_r$ of

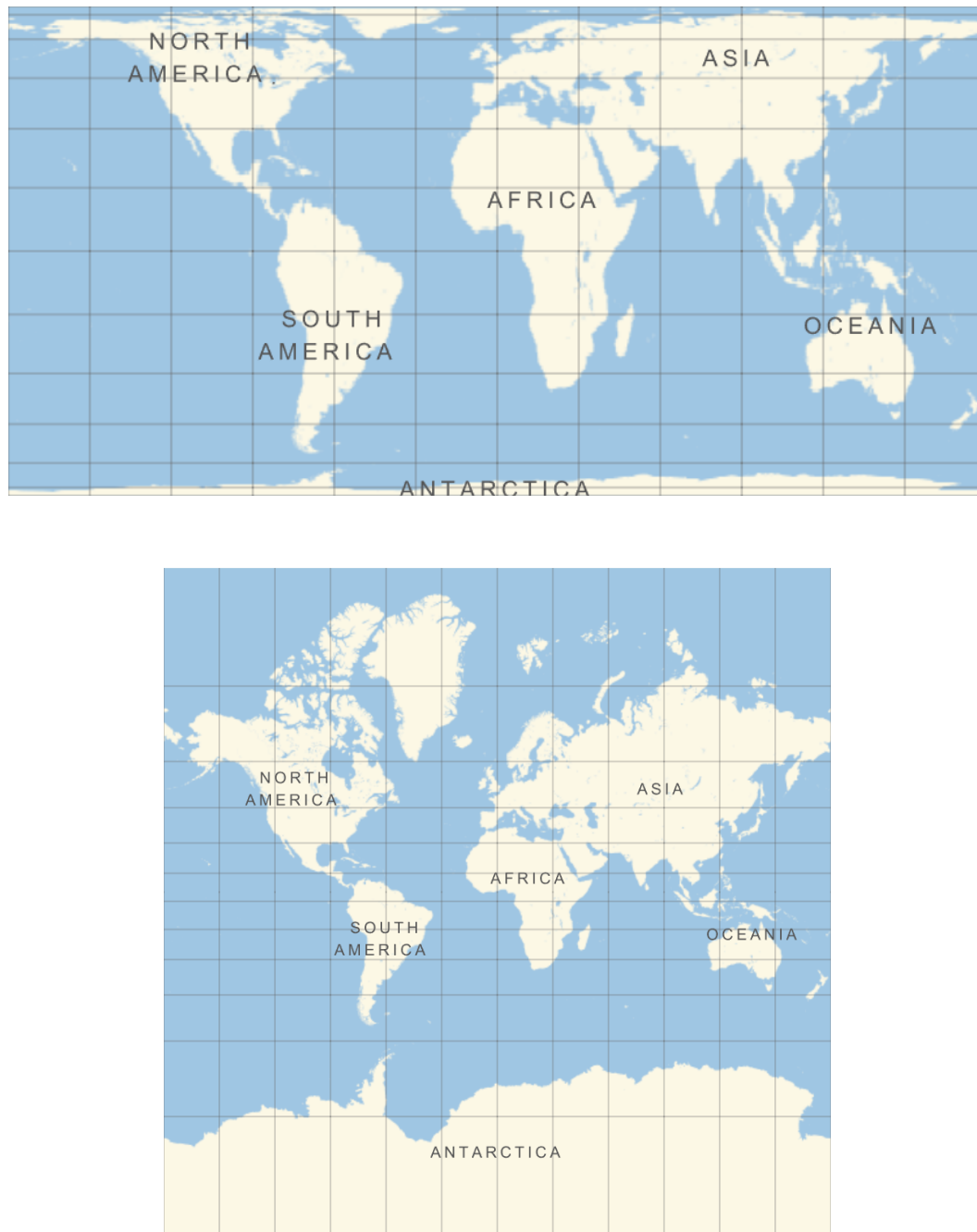


FIGURE 10.4. A comparison of the Lambert (top) and Mercator (bottom) cylindrical projections of the Earth, both aligned North-South.

Set	Meaning
$\mathcal{R}$	Races/Initiatives in the election, labeled $1..m$
$\mathcal{C}_r$	Candidates/options in race r , includes “did not vote”
$\mathcal{V}$	Registered voters in district, where $\#\mathcal{V} = n$
$\mathcal{C}_r \rightarrow [0..n]$	Set of functions giving vote count for each option

TABLE 10.2. Notation describing the entities in Example 10.5.

candidates/options from which the voters can choose, with one option always included to indicate “did not vote”. There is also a set of n registered voters $\mathcal{V}$, and because we are within one district, every voter can record a vote for every race. Table 10.2 summarizes the notation for the relevant sets here.

If we observed each vote directly, we could tally the results in a spreadsheet with mn rows and three columns. The first spreadsheet column would list each of the n voters in $\mathcal{V}$ exactly m times, once per race, and the second column would list all the elements of $\mathcal{R}$ in separate rows for each voter. Taken together the first two columns list all the pairs of voter and race, and there is exactly one row in the spreadsheet for each such pair. The third column would contain the voter’s choice (possibly “did not vote”) for each voter-race pair.

This spreadsheet represents a function that takes as input a voter-race pair and returns that voter’s choice in that race. The set of inputs to this function is precisely $\mathcal{V} \times \mathcal{R}$, the set of pairs. The function’s output needs to associate a choice for race r with every input pair that includes race r , so the set of outputs is a disjoint union, $\bigsqcup_{r \in \mathcal{R}} \mathcal{C}_r$. Thus, the spreadsheet is a function s from $\mathcal{V} \times \mathcal{R}$ to $\bigsqcup_{r \in \mathcal{R}} \mathcal{C}_r$ where $s(v, r)$ is in $\mathcal{C}_r \times \{r\}$; $s(v, r)$ records voter v ’s choice in race r .

Real elections have a “secret ballot” in which voter identities are hidden and only tallies of the votes in each race are recorded. Such a tally is also a function. It takes as input a race and returns a “table” that associates a count to each option for *that* race. How do we represent such a table? As you might have guessed, that table is just a representation of a function that maps $\mathcal{C}_r$ to $[0..n]$, giving a count for each option. We denote the set of such functions by $\mathcal{C}_r \rightarrow [0..n]$. Our tally then is a function t that takes as input a race r and returns a lookup table associating each option in $\mathcal{C}_r$ with its vote count. The function has type

$$t: \mathcal{R} \rightarrow \bigsqcup_{r \in \mathcal{R}} (\mathcal{C}_r \rightarrow [0..n]).$$

The disjoint union reflects that the function returns a table associated with one

In database terms, the first two columns comprise a “primary key” for the table.

Ragged in the sense that different rows can have different numbers of columns.

race only. In spreadsheet terms, the function t is a “ragged” spreadsheet whose first column contains the race, and the remaining columns in row r are the counts for each choice in that race.

Example 10.6 Satellite Navigation

The Global Positioning System is a network of satellites in Earth orbit that enable precise navigation on the Earth’s surface. Each satellite is equipped with a precise atomic clock and is kept in a well-controlled orbit. Here, we consider an idealized version of the system, where the receiver (e.g., your phone) also has a precise atomic clock and simultaneously receives signals from three satellites with their orbital positions at specified times. (We also assume that you are on the surface of the Earth and moving much more slowly than the satellites.)

That is, your receiver knows the time t when the satellite signals’ are received and the position $\langle x_s, y_s, z_s \rangle$ of satellite s at time t_s , for each $s \in \{1, 2, 3\}$. Because the signal from satellite s took $t - t_s$ seconds to reach you, your position $\langle x, y, z \rangle$ is a distance $(t - t_s)/c$ from position $\langle x_s, y_s, z_s \rangle$, where c is the speed of light. For each $s \in \{1, 2, 3\}$, we thus have

$$(x - x_s)^2 + (y - y_s)^2 + (z - z_s)^2 = \frac{(t - t_s)^2}{c^2}.$$

There are *at most two points* $\langle x, y, z \rangle$ that solve these three equations and in general only one is on the surface of the Earth. Solving for this point yields a function that maps the 13-tuple $\langle t, x_1, y_1, z_1, t_1, x_2, y_2, z_2, t_2, x_3, y_3, z_3, t_3 \rangle$ to your position $\langle x, y, z \rangle$.

The real system handles many complications, particularly that your phone does not have an atomic clock. But with some engineering cleverness, the same basic idea defines your approximate position as a function of the signals from (at least) four satellites.

Example 10.7 Projectile Distance

The distance that a projectile will travel is a function of the launch parameters, with various possible functions depending on what we measure. (Here we ignore friction and assume a launch from the ground over a “flat” region on the Earth’s surface.) Measuring the projectile’s mass (m), the angle of launch (θ), and the

energy (K) imparted to the projectile during the launch, gives

$$d_1(m, \theta, K) = \frac{2K}{mg} \sin(2\theta),$$

where g is the gravitational acceleration. Measuring the projectile's mass and velocity (v) at the moment of launch gives

$$d_2(\theta, v) = \frac{v^2}{g} \sin(2\theta).$$

Measuring mass and the force applied to the projectile over a small distance Δ at launch gives

$$d_3(m, \theta, F, \Delta) = \frac{2F\Delta}{mg} \sin(2\theta).$$

We can convert from one form to another by applying a function to the inputs. For instance, $f(m, \theta, k) = \langle \theta, 2K/m \rangle = \langle \theta, v \rangle$ yields $d_1(m, \theta, K) = d_2(f(m, \theta, K))$.

A MENAGERIE OF FUNCTIONS. The following examples illustrate how we define and denote functions and present some specific functions that we will use later in this section. Some of these (like `id`, `const`, `incl`) will be used throughout the book (see Section 10.2); most of the other definitions are *local* to the paragraph in which they are defined.

- i. Perhaps the simplest thing we can do with an input is nothing at all. The **identity function** `id` returns its argument as is, whatever it is: $\text{id}(x) = x$. Although it is simple, we will frequently find `id` useful.
- ii. About as simple a thing to do with an input is to ignore it and return a pre-specified value. The **constant function** `constc` returns the value c for any given argument, $\text{const}_c(x) = c$. We will use this regularly. Notice that we have a different function for each value c .
- iii. Define the function $\ell_{a,b}$ for real numbers a, b by $\ell_{a,b}(x) = a + bx$. This maps real numbers to real numbers. It is called an *affine function*, though colloquially people often call it “linear” as its graph describes a line in the plane.
- iv. Generalizing this, the function $P_{d,a}(x) = a_0 + a_1x + a_2x^2 + \cdots + a_dx^d$ is a degree d **polynomial**, where $d \in [0..)$ is a natural number and $a = \langle a_0, a_1, \dots, a_d \rangle$ is a tuple of real numbers. Here, we think of P as mapping real numbers to real numbers. Notice that we have a different function for every choice of the numbers $d, a_0, \dots, a_d$.
- v. Define functions f and s that take 2-tuples of numbers and returns a number, with $f(\langle a, b \rangle) = a$ and $s(\langle a, b \rangle) = b$. Here, f stands for “first”, and s stands for “second.”
- vi. A function a from the natural numbers¹ $\mathbb{N}$ to the real numbers $\mathbb{R}$ is called a (numeric) **sequence**, representing the list of numbers $a(0), a(1), a(2), a(3), \dots$. For example, if $a(0) = 0$, $a(1) = 1$, and $a(n) = a(n-1) + a(n-2)$ for any $n \in [2..)$, then a represents the famous **Fibonacci numbers**. For sequences, we can write the evaluated function as a_n or $a(n)$, as we prefer.
- vii. Let p map $\mathbb{R}^7$ to $\mathbb{R}^7$ where $p(\langle x_1, x_2, \dots, x_7 \rangle) = \langle x_6, x_7, x_4, x_3, x_1, x_2, x_5 \rangle$. This is an example of a **permutation**. (See page 674.)
- viii. Let m be the function that takes an integer and returns the remainder upon dividing by 5. That is, $m(k) = k \bmod 5 \in [0..4]$. We think of two integers as equivalent “mod 5” if they m maps them to the same value.

¹Recall: The natural numbers include 0.

- ix. Let u be the function that maps a finite set of natural numbers to the set of all its subsets. For instance, $u(\{1, 2\}) = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$.
- x. Let σ (for “shift”) be a function that takes as input a *function* that itself maps $\mathbb{R}$ to $\mathbb{R}$ and returns another such function, where $\sigma(f)(x) = f(x - 1)$. This looks a bit mind-blowing at first, so take it in pieces: f is any function that maps real numbers to real numbers; $\sigma(f)$ is also such a function; the return value of $\sigma(f)$ for input x is $\sigma(f)(x)$;² and $\sigma(f)(x)$ is just the value that f would return given $x - 1$. (Try plotting the graph of f and $\sigma(f)$ when $f(x) = x^2$.)
- xi. Define function $i(f) = \int_{x=0}^1 f(x) dx$ to take non-negative functions (i.e., $f(x) \geq 0$ for all $x \in [0, 1]$) defined on the unit interval and return their integrals, which may be infinite. This is a function that takes a *function* as input.
- xii. Define the function ζ by $\zeta(x) = 1$ if $x \in [0, 1]$ and $\zeta(x) = 0$ if $x \notin [0, 1]$. If we think of 1 as being “true” and 0 as “false,” this is the function that indicates whether its argument is between 0 and 1 (inclusive). We call this an *indicator function*; see Chapter 12.
- xiii. Let $\mathcal{N}$ and $\mathcal{E}$ be the sets of nodes and edges in a directed graph. Define a function ζ that maps from $\mathcal{E}$ to $\mathcal{N} \times \mathcal{N}$. Given an edge $e \in \mathcal{E}$, $\zeta(e) = \langle s, t \rangle$, where s the source node for that edge, and t is the target node.
- xiv. Let ρ (for “reverse”) be a function that maps directed graphs to directed graphs, where $\rho(G)$ is the directed graph obtained from G by reversing the direction of all the edges.
- xv. Any integer is also a real number, so given an integer k , we can write $\text{incl}(k) = k$ where we think of the return value as a real number, a kind of “type conversion”. In general, given $\mathcal{A} \subseteq \mathcal{B}$, an **inclusion** incl maps $\mathcal{A}$ to $\mathcal{B}$ with $\text{incl}(a) = a$. This looks like the identity but is in general a distinct idea.

²If it helps, define $g = \sigma(f)$, then this is just $g(x)$.

FUNCTIONS DEFINED. With these examples in hand, we are ready to formally define functions and describe the language we use to talk about them.

A **function** f from a set $\mathcal{A}$ into a set $\mathcal{B}$ associates to *each* $x \in \mathcal{A}$ exactly one value $f(x) \in \mathcal{B}$. The set $\mathcal{A}$ is called the **domain** of f and $\mathcal{B}$ is called the **codomain**. When needed, we write $\text{dom}(f)$ and $\text{codom}(f)$ for the domain and codomain of a function f .

We write $f: \mathcal{A} \rightarrow \mathcal{B}$ or $\mathcal{A} \xrightarrow{f} \mathcal{B}$ to indicate that f is a function with domain $\mathcal{A}$ and codomain $\mathcal{B}$. In that case, f is said to have *type* $\mathcal{A} \rightarrow \mathcal{B}$, which represents the set of functions³ mapping $\mathcal{A}$ to $\mathcal{B}$.

We say that a function *takes* a domain value as input and *returns* a codomain value as output. The domain of a function is the set of *allowed* inputs to the function, and the codomain is the set of *possible* outputs. If we think of sets of values as *types*, then $\text{dom}(f)$ is the type of inputs the function f accepts and $\text{codom}(f)$ is the type of outputs f returns. This might be familiar from many programming languages where we specify the types of a function's parameters and return values.

It is important to distinguish between a function and a returned value. For $f: \mathcal{X} \rightarrow \mathcal{Y}$, then f is the function itself, but if $x \in \mathcal{X}$, $f(x)$ is a returned *value* in the codomain. So f and $f(x)$ are not the same kind of object.

Indeed, a function is a mathematical object, and we treat the domain and codomain *as part of the object*. So, we treat functions with different domains or codomains as *different functions*. As an example of this, for $\mathcal{A} \subseteq \mathcal{B}$ we can define two distinct functions: the identity function on $\mathcal{A}$, $\text{id}: \mathcal{A} \rightarrow \mathcal{A}$ and the inclusion of $\mathcal{A}$ into $\mathcal{B}$, $\text{incl}: \mathcal{A} \rightarrow \mathcal{B}$. Both functions return a for each input $a \in \mathcal{A}$, but we think of them as different functions because their output *types* (i.e., codomains) are different. Think of incl as a type conversion that “casts” a value a from type $\mathcal{A}$ to type $\mathcal{B}$.

A function must take every value in its domain, but it *need not* return every value in its codomain. That is, the actual outputs of a function need not match the possible outputs. The canonical example of this is a constant function $\text{const}_c: \mathcal{X} \rightarrow \mathcal{C}$ for $c \in \mathcal{C}$ and non-empty $\mathcal{X}$; it returns the same value c for every input value $x \in \mathcal{X}$. For a function $f: \mathcal{A} \rightarrow \mathcal{B}$, the set of actual outputs is called the *range* of f ,

$$\text{range}(f) = \{f(x) \mid x \in \mathcal{A}\} \subseteq \mathcal{B}. \quad (10.1)$$

Similarly, the *graph* of f is the set

$$\text{graph}(f) = \{\langle x, f(x) \rangle \mid x \in \mathcal{A}\} \quad (10.2)$$

that reveals the underlying mapping in its elements.

We often define functions with an *equation* that shows the mapping explicitly. For example, we might define $g: \mathbb{R} \rightarrow \mathbb{R}$ by

$$g(z) = \frac{2z - 3}{1 + z^2}. \quad (*)$$

³Some writers denote the set of functions from $\mathcal{A}$ to $\mathcal{B}$ by $\mathcal{B}^{\mathcal{A}}$ for reasons we will see in Chapter 18.

This equality means that whenever you see $g(\text{something})$, you can replace it with the right side of that equation with *something* substituting in for z . The z in equation (*) is the **parameter** of the function; it is a variable that is *local to the defining equation*. If we change its name throughout the equation, it makes no difference – we define the same function. So, $g(y) = (2y - 3)/(1 + y^2)$ is equivalent to equation (*). When we give a function a value v from the domain to get back a value $g(v)$ in the codomain, we have **evaluated** the function at v . The input value is an **argument** to the function. For example, we can evaluate g from (*) with $g(4), g(1 + a), g(\sqrt{\sin(\pi/17)})$. The arguments here are 4, $1 + a$, and $\sqrt{\sin(\pi/17)}$, where a is a variable defined elsewhere, and the corresponding return values are $5/17$, $\frac{2a-1}{2+2a+a^2}$, and $\frac{2\sqrt{\sin(\pi/17)}-1}{1+\sin^2(\pi/17)}$. Colloquially, we often elide the distinction between parameter and argument, but the key is that the parameter is a local variable specified only in the defining equation, and the argument is a value at which the function is evaluated.

We can define functions with *more than one parameter*, using a shorthand discussed in detail in Chapter 15. For instance, we could define $f(x, y) = \sqrt{x^2 + y^2}$ taking two real numbers, but this is just a convenient way to write $f(\langle x, y \rangle) = \sqrt{x^2 + y^2}$ taking an input in $\mathbb{R}^2$, a 2-tuple of numbers. Similarly, $\psi(r, s, t)$, $w(a, b, c, d)$, $\dots$, in either a defining equation or ensuing evaluation are just shorthands for $f(\langle x, y \rangle)$, $\psi(\langle r, s, t \rangle)$, $w(\langle a, b, c, d \rangle)$, $\dots$. By direct analogy, a function that takes *no parameters* is just a function from $\mathbb{U} \rightarrow \mathcal{C}$ for some codomain $\mathcal{C}$, where $\mathbb{U} = \{\langle \rangle\}$ is called the **unit type**, as it contains only one element. Each such function picks out a single element of $\mathcal{C}$.

The functions in examples (ii), (iii), and (iv) on page 562 depend on extra values (like a, b, c, d) in their definition. We often need to define functions that depend on such values, which we call **exogenous parameters** because they must be specified elsewhere to determine the specific function. When the exogenous parameters are fixed from context (e.g., defined during an earlier part of our calculation), we need not explicitly show the function's dependence on them. Otherwise, as in examples (ii), (iii), and (iv), we do, usually either with sub- or superscripts or listing them in order after a delimiter in the argument list, like $\ell(x \mid a, b)$. For example, consider the function $h: (0_{-}) \rightarrow (0_{-})$ defined by

$$h(t) = c_1 10^{-c_2 t}, \quad (**)$$

which has two exogenous parameters $c_1 > 0$ and $c_2 > 0$. If c_1 and c_2 have been defined already in our work, then we can define h as in equation (**). Otherwise, we

might write the definition to make dependence on c_1, c_2 explicit, like

$$\begin{aligned} h_{c_1, c_2}(t) &= c_1 10^{-c_2 t} \\ h(t \mid c_1, c_2) &= c_1 10^{-c_2 t}, \end{aligned}$$

or something similar. In both cases, the exogenous parameters are specified by position at evaluation time. The collection of functions h_{c_1, c_2} over all the exogenous parameters defines a *family* of functions, an idea we return to below.

We name functions in various ways depending on how we use them. As you can see with `id` and `constc`, commonly used functions are given word-like names. The same goes for many common mathematical functions⁴ like `log`, `sin`, and `min`. A few special functions are named with capital Greek letters, like Γ (the “Gamma function”) and Φ (the Normal CDF defined in Chapter 1). Otherwise, we define functions on the fly when we need them, and we then typically use lowercase roman (e.g., f, g) or Greek letters (e.g., ψ, ϕ). These can be augmented with various subscripts, superscripts, or decorations as needed. Often it is convenient to define and use a function *without naming it*; these are called *anonymous functions* and are discussed in Chapter 11.

⁴Many standard mathematical and special functions are discussed in detail in Appendix B.

The reader may have noticed a slight inconsistency in the foregoing discussion relating to a few of the functions above. Consider the identity function `id` ... well it is not exactly one function is it? For every domain $\mathcal{A}$, we have a unique identity function $\text{id}_{\mathcal{A}}: \mathcal{A} \rightarrow \mathcal{A}$ defined by $\text{id}_{\mathcal{A}}(a) = a$. For different domains, we have different functions. Usually, though, the specific identity function we need is clear from context, so we use the same name `id` for all of these functions.⁵ Mathematicians call this “slight abuse of notation,” computer scientists call it *polymorphism*. For a few functions that act generically on their arguments – e.g., the identity `id`, constant functions `constc`, inclusions `incl` – we use the same name for the generic function of different types when those types are clear from context. For example, for both `constc` and `incl` there is a distinct such function for each pair of domain and codomain, but usually it is clear which function we need. (And if not, we can simply note `constc:  $\mathcal{A} \rightarrow \mathcal{C}$` or `incl:  $\mathcal{X} \rightarrow \mathcal{Y}$` to disambiguate.)

⁵If absolutely needed, we can always indicate the domain with a subscript like we just did.

In light of the preceding ideas, let’s write down the types of selected functions from the earlier list of examples on pages 562 and 562.

- (i) We have seen that for any domain $\mathcal{A}$, we have a version of the identity function $\text{id}: \mathcal{A} \rightarrow \mathcal{A}$.
- (ii) Similarly, for any $\mathcal{X}$ and $\mathcal{C}$ with $c \in \mathcal{C}$, we have `constc:  $\mathcal{X} \rightarrow \mathcal{C}$` . But we can also view `const` itself (no subscript) as a function that takes a value $c \in \mathcal{C}$ and returns the *function* `constc`. This has type `const:  $\mathcal{C} \rightarrow (\mathcal{X} \rightarrow \mathcal{C})$` . The codomain (i.e.,

return type) is the set of functions from $\mathcal{X} \rightarrow \mathcal{C}$.

- (iii) As noted above, $\ell_{a,b}: \mathbb{R} \rightarrow \mathbb{R}$.
- (v) Both $f, s: \mathbb{R}^2 \rightarrow \mathbb{R}$. What would these types be if f and s had domain $\mathcal{A} \times \mathcal{B}$?
- (vi) A sequence a has type $a: \mathbb{N} \rightarrow \mathbb{R}$. The function here serves as a *package* for an infinite list of numbers. A basic thing we can do with a sequence is to find the number at a particular position in the sequence; that's what the function does.
- (viii) $m: \mathbb{Z} \rightarrow [0..4]$. Think of m as assigning a particular property to each integer.
- (x) $\sigma: (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$. It operates on a numeric function, returning a modified numeric function.
- (xi) $i: ([0\_1] \rightarrow [0\_]) \rightarrow [0\_ \infty]$. Look carefully at what the types here say about i 's input function and return value.
- (xii) $\mathbb{1}_{[0\_1]}: \mathbb{R} \rightarrow \{0, 1\}$. What property is $\mathbb{1}_{[0\_1]}$ assigning to each real number?

Puzzle 91. Write an implementation of example functions (i), (ii), (iii), (v), (viii), (x), and (xii) in Python or your favorite programming language.

If your language supports it (as Python, JavaScript, Haskell, R, Ocaml, and many others do), for (ii), write a function `const` that takes a value c and returns the *function* const_c .

See below for solutions.

MORE EXAMPLES. We now consider a range of examples and puzzles relating to functions. Their purpose is to give you practice thinking about what functions are, how we define them, and the various ways we use them. Try the puzzles and study how functions are introduced and applied in the examples. The content in the examples should help ground the abstract operations in meaningful context.

Puzzle 92. In the previous section, we defined a tuple as an element of a product set. Inspired by example (vi), show how we can define a d -dimensional tuple of real numbers as a *function*. (Hint: A basic operation given a tuple is finding the value in a particular component.) What is the type of this function? How is the function defined?

This is nice example of how functions can act as *packages* for organizing other quantities and as tools for extracting information from larger structures.

Example 10.8 Apples in Buckets. Let $\mathcal{A}$ and $\mathcal{B}$ be finite sets. Think of the elements of $\mathcal{A}$ as labels on a collection of apples, and the elements of $\mathcal{B}$ as labels on a group of buckets. Each function $f: \mathcal{A} \rightarrow \mathcal{B}$ describes one way to distribute the apples into buckets.

Putting all the apples in one bucket gives us a constant function, of which there are $\#\mathcal{B}$ choices. If every bucket is the output of f for some apple, we know that every bucket gets at least one apple. If $a_1 \neq a_2$ implies that $f(a_1) \neq f(a_2)$, then every bucket gets at most one apple. If $n = \#\mathcal{A} = \#\mathcal{B}$ and every bucket gets *exactly* one apple, then there are $n!$ such functions.

Example 10.9 Text Embeddings. A critical part of modern Natural Language Processing is to convert structured data like text into a numeric form that is more amenable to analysis. A *text embedding* is a function from a set of texts into $\mathbb{R}^n$ for some $n \in [1..)$. A simple example is the so-called “one-hot encoding.”

See Table 10.3.

Let $\mathcal{W}$ be a set of words and let $\mathbb{2} = \{0, 1\}$. The function $r: \mathcal{W} \rightarrow [1.. \#\mathcal{W}]$ gives an arbitrary ordering of the words (e.g., alphabetical) by mapping each word to a distinct integer. The one-hot embedding $h: \mathcal{W} \rightarrow \mathbb{2}^{\#\mathcal{W}}$ maps word w to an $\#\mathcal{W}$ -tuple $h(w)$ where the $r(w)$ th component is 1 and the rest 0:

$$h(w) = \langle \overbrace{0, \dots, 0}^{r(w) \text{ components}}, 1, 0, \dots, 0 \rangle.$$

Each word maps to a unique numeric vector that can serve as input to analysis.

This approach can be extended to a set of documents, $\mathcal{D}$, where each document is a non-empty list of words from $\mathcal{W}$. We define $b: \mathcal{D} \rightarrow \mathbb{R}^{\#\mathcal{W}}$ to map document d the tuple whose components *count* how many times each word in $\mathcal{W}$ appears in d . $b(\langle w_1, w_2, \dots, w_\ell \rangle) = \sum_{i=1}^{\ell} h(w_i)$, where we add the tuples together component-wise. This embedding represents a document as a *bag of words*.

Notice how we use functions here to represent an ordering (r) and to convert objects to new representations (h, b).

While the one-hot encoding is simple and intuitive, it has a few drawbacks. First, the dimension of encoded tuples equals the size of our vocabulary, which can be unwieldy. Second, it ignores semantic information. For instance, we might want *similar words* to map to tuples that are closer to each other than to unrelated words. Third, the same word always maps to the same tuple regardless of the context in which it is seen (e.g., part of speech, surrounding words). Fourth,

Entity	Meaning
$\mathcal{W}$	Set of words, a vocabulary
r	Arbitrary ordering of words in $\mathcal{W}$
h	One-hot embedding on words
$\mathcal{D}$	A corpus of documents
b	Bag-of-words embedding of $\mathcal{D}$

TABLE 10.3. Notation describing the entities in Example 10.9.

h cannot represent unknown words.

Modern approaches address all these drawbacks by constructing more refined functions whose codomains have a fixed (but reasonably large) dimension and whose domains can include not just the word but also context such as surrounding words. The resulting tuples reflect similarity and other semantic features. Unknown words can be handled by defining the function on parts of words. These kinds of functions are usually constructed algorithmically from large bodies of existing text; that is, they are “learned from data,” often with a neural network.

Example 10.10 Tagging and Destructuring

The disjoint union $\mathcal{X} \sqcup \mathcal{X}$ consists of two copies of the set $\mathcal{X}$. We can define functions $\ell, r: \mathcal{X} \rightarrow \mathcal{X} \sqcup \mathcal{X}$ (short for “left” and “right” respectively) that attach an appropriate *tag* to a value $x \in \mathcal{X}$ and “injects” it into the corresponding copy:

$$\ell(x) = \langle x, 1 \rangle \qquad r(x) = \langle x, 2 \rangle,$$

where we use the tags 1 and 2 by default on the two parts. For a general disjoint union $\bigsqcup_{i \in \mathcal{I}} \mathcal{A}_i$, we can define for each $i \in \mathcal{I}$ a function $\text{inj}_i: \mathcal{A}_i \rightarrow \bigsqcup_{i \in \mathcal{I}} \mathcal{A}_i$ that injects a “tagged” $a \in \mathcal{A}_i$ into the disjoint union, where $\text{inj}_i(a) = \langle a, i \rangle$.

Given functions $f: \mathcal{A} \rightarrow \mathcal{C}$ and $g: \mathcal{B} \rightarrow \mathcal{C}$, define a function $f \sqcup g: \mathcal{A} \sqcup \mathcal{B} \rightarrow \mathcal{C}$ by

$$\begin{aligned} (f \sqcup g)(\langle a, 1 \rangle) &= f(a) \\ (f \sqcup g)(\langle b, 2 \rangle) &= g(b). \end{aligned}$$

This *destructures* an element of the disjoint union and passes its underlying value to the appropriate function. The shape of the operator here is intended to evoke two joined parts.

Example 10.11 Absurdly Unique. If $\mathcal{C}$ is a set, we can define a function from $\{\}$ to $\mathcal{C}$, but because the empty set has no elements, we can never evaluate it. Moreover, any two such functions $\{\} \rightarrow \mathcal{C}$ must be *equal* as they agree for every element of $\{\}$ – for which there are none.

Thus, for every set $\mathcal{C}$ there is a unique function **absurd**: $\{\} \rightarrow \mathcal{C}$.

Example 10.12 Subsets. Let $\mathcal{A}$ be a non-empty set and define $\mathbb{2} = \{0, 1\}$.

For every $\mathcal{B} \subseteq \mathcal{A}$, we can define a function $f: \mathcal{A} \rightarrow \mathbb{2}$, called the *indicator of $\mathcal{B}$* , for which $f(a) = 1$ when $a \in \mathcal{B}$ and $f(a) = 0$ when $a \notin \mathcal{B}$.

Conversely, for every function $f: \mathcal{A} \rightarrow \mathbb{2}$, we can define a subset of $\mathcal{A}$: $\{a \in \mathcal{A} \mid f(a) = 1\}$. Thus every subset of $\mathcal{A}$ corresponds to a unique function $\mathcal{A} \rightarrow \mathbb{2}$ and vice versa.

We will denote the set of all subsets of a set $\mathcal{A}$ by $2^{\mathcal{A}}$. The set $2^{\mathcal{A}}$ is called the **power set** of $\mathcal{A}$. The function $\mathcal{A} \rightarrow 2^{\mathcal{A}}$ mapping $a \in \mathcal{A}$ to the set $\{a\} \in 2^{\mathcal{A}}$ so there are at least as many subsets as elements, $\#\mathcal{A} \leq \#2^{\mathcal{A}}$.

Example 10.13 Subsets as Bitstrings. As in the previous example, let $\mathcal{A}$ be a non-empty set and let $\mathbb{2} = \{0, 1\}$, but assume that $\mathcal{A}$ is a *finite* set with $n = \#\mathcal{A}$.

We can assign a “rank” to each element of $\mathcal{A}$ by a function $r: \mathcal{A} \rightarrow [1..n]$ such that every $r(a)$ is a distinct integer.

Then define $b: 2^{\mathcal{A}} \rightarrow \mathbb{2}^n$ by mapping a subset $\mathcal{C} \subseteq \mathcal{A}$ to the n -tuple for which the components $r(c)$ for $c \in \mathcal{C}$ are set to 1 and the other components are all set to 0. Thus, each subset of $\mathcal{A}$ corresponds to a unique n -bit string and vice versa.

Example 10.14 Multisets, aka Bags

The English-language version of the game *Scrabble* comes with 100 tiles, each representing a letter. There are 12 E’s, 9 A’s, 9 I’s, 8 O’s, 6 N’s, 6 R’s, 6 T’s, 4 L’s, 4 S’s, 4 U’s, 4 D’s, 3 G’s, 2 each of B, C, M, P, F, H, V, W, Y, 1 each of K, J, X, Q, Z, and 2 blank tiles that can stand in for any letter. How do we represent this collection of objects?

A **multiset**, also known as a **bag**, is an object like a set but in which we can hold a particular object multiple times. A multiset is a *function* $m: \mathcal{A} \rightarrow \mathbb{N}$ that counts how many times each $a \in \mathcal{A}$. The set $\mathcal{A} = \text{dom}(m)$ is called the *base set* of the multiset.

For instance, the bag of scrabble tiles s has base set $\{\mathbf{A}, \mathbf{B}, \dots, \mathbf{Z}, \mathbf{\_}\}$ with, e.g., $s(\mathbf{E}) = 12$, $s(\mathbf{T}) = 6$, $s(\mathbf{X}) = 1$, $s(\mathbf{\_}) = 2$.

A classic example is the multiset of prime factors of an integer $n \geq 2$. This is a function $f_n: \text{Primes} \rightarrow \mathbb{N}$, where for instance with $15400 = 2^3 \cdot 5^2 \cdot 7 \cdot 11$, $f_{15400}(2) = 3$, $f_{15400}(3) = 0$, $f_{15400}(5) = 2$, $f_{15400}(7) = 1$, $f_{15400}(11) = 1$ and $f_{15400}(p) = 0$ for prime $p > 11$.

If m_A, m_B are multi-sets with a common base set $\mathcal{U}$, we can define analogues of the common set concepts and operations:

- The *cardinality* of m_A is given by $\#m_A = \sum_{a \in \text{dom}(m_A)} m_A(a)$, representing the total number of objects in the bag.
- The *support* of m_A is $\{u \in \mathcal{U} \mid m_A(u) > 0\}$, which need not equal $\mathcal{U}$.
- m_A is *included* in m_B when $m_A \leq m_B$, analogously to a subset relation.
- The *union* of m_A and m_B is the multiset $m = \max(m_A, m_B)$.
- The *intersection* of m_A and m_B is the multiset $m = \min(m_A, m_B)$. Two multisets are *disjoint* if their intersection is the function const_0 .
- The *sum* of m_A and m_B is the multiset $m = m_A + m_B$, analogous to a disjoint union of sets.
- The *difference* between m_B and m_A is the multiset $m = \max(m_B - m_A, \text{const}_0)$, analogous to the set difference.

So for instance, referring to the functions above, $\#s = 100$ and $f_j + f_k = f_{jk}$.

A **multiset**, or **bag**, on *base set* $\mathcal{A}$ is a function $\mathcal{A} \rightarrow \mathbb{N}$. This represents a collection of objects in $\mathcal{A}$ that can be repeated. If m is a multiset on $\mathcal{A}$ and $a \in \mathcal{A}$, $m(a)$ is the number of times a appears in the multiset.

The cardinality and *support* of a multiset, and the union, intersection, sum, and difference of two multisets are defined in Example 10.14.

Example 10.15 Collections and Families We frequently group collections of objects into *families* indexed by one or more parameters. Earlier, for instance, when discussing collections of sets we wrote “a collection of sets $\mathcal{A}_i$ indexed by

$i \in \mathcal{I}$.” Or we might posit that the amount of a radioactive isotope in a substance decreases with time t like $a2^{-ct}$ for some constants $a, c > 0$; this implicitly specifies a family of functions r_{ac} , one for each pair of exogenous parameters a and c , with $r_{ac}(t) = a2^{-ct}$. Or we can look at a numeric tuple $\langle x_1, \dots, x_n \rangle$ as a *family* of numbers x_i indexed by $i \in [1..n]$.

Underlying this idea of families, however we denote them, are functions. A family of objects indexed by some set $\mathcal{I}$ is a *function* from $\mathcal{I}$ to the collection of possible objects. So for example:

1. A family of sets is a function from $\mathcal{I}$ to the set of all subsets of some set $\mathcal{U}$, i.e., of type $\mathcal{I} \rightarrow 2^{\mathcal{U}}$.
2. A family of functions $\mathcal{A} \rightarrow \mathcal{B}$ with exogenous parameters in a set $\mathcal{P}$ is a function from $\mathcal{P}$ to a set of functions of type $\mathcal{A} \rightarrow \mathcal{B}$, i.e., of type $\mathcal{P} \rightarrow (\mathcal{A} \rightarrow \mathcal{B})$.
3. A tuple of dimension d whose components belong to a set $\mathcal{A}$ is a function from $[1..d]$ to $\mathcal{A}$.
4. A sequence of numbers $a_0, a_1, a_2, \dots$ is a function from $\mathbb{N}$ to $\mathbb{R}$.

We often denote the members of a family with some base symbol and the index in a subscript, so $\mathcal{A}_i$ for a family of sets or x_j for the component of a tuple or f_n for the prime factors of n in the family of functions in Example 10.14. Think of the base symbol as the “family name,” which identifies the family, and the index as the “given name,” which identifies the individual in that family. Alternatively, to give it more room, we often put the index of a function family after a $|$ (read “given”), e.g., $r_{ac}(t)$ above might be written as $r(t | a, c)$.

Example 10.16 Features

A consumer is choosing among a set $\mathcal{P}$ of products and intends to purchase one of them. The consumer is deciding based on which products have (or do not have) the features in a set $\mathcal{F}$.

As in Example 10.12, let $2 = \{0, 1\}$ and use $2^{\mathcal{P}}$ and $2^{\mathcal{F}}$ to denote the set of all subsets of products and features, respectively. For example, each element of $2^{\mathcal{P}}$ is a set of products in $\mathcal{P}$.

The consumer defines two functions $f: 2^{\mathcal{P}} \rightarrow 2^{\mathcal{F}}$ and $p: 2^{\mathcal{F}} \rightarrow 2^{\mathcal{P}}$ where

- $f(\mathcal{A})$ is the set of features that every product in $\mathcal{A}$ has, and
- $p(\mathcal{C})$ is the set of products that have all the features in $\mathcal{C}$.

Notice that $\mathcal{A} \subseteq \mathcal{B}$ implies that $f(\mathcal{B}) \subseteq f(\mathcal{A})$, and $\mathcal{C} \subseteq \mathcal{D}$ implies that $p(\mathcal{D}) \subseteq$

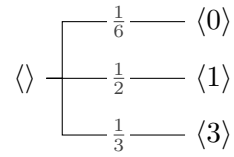
$p(\mathcal{C})$. So both f and p reverse the subset ordering. Each feature in $\mathcal{C}$ is included with every product in $\mathcal{A}$ if and only if every product in $\mathcal{A}$ has all the features in $\mathcal{C}$. That is:

$$\mathcal{C} \subseteq f(\mathcal{A}) \text{ if and only if } \mathcal{A} \subseteq p(\mathcal{C}).$$

Example 10.17 Kinds as Functions

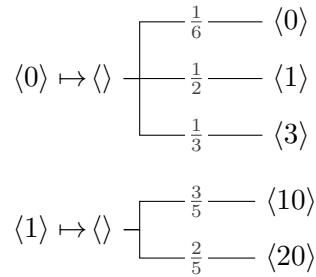
Based on Part I

We depict a *Kind* in Chapter 1 by a single-level complete tree with values on the leaves and weights on the edges. For example,



Kinds have an equivalent representation as a *function* that maps *values to weights*. In the above tree, for instance, we would define the function K with $K(0) = 1/6$, $K(1) = 1/2$, and $K(2) = 1/3$, and $K(x) = 0$ otherwise.

We can go further. A *conditional Kind* is a function from *values to Kinds*, which we often depict as a tree associated with each input value. For instance, the conditional Kind



has input values 0 and 1 and returns the corresponding tree. By representing the returned tree as a function, we can think of this as a *family of functions*, one for input value. We write this family as a function with the input value as an exogenous parameter: for input value y , $K(x | y)$ gives the weight for value x in the tree associated with y , or 0 if no such value is a leaf on that tree. In the above case, we have

$$\begin{array}{lll} K(0 | 0) = 1/6 & K(1 | 0) = 1/2 & K(3 | 0) = 1/3 \\ K(10 | 1) = 3/5 & & K(20 | 1) = 2/5, \end{array}$$

giving a function for the kind for each input value.

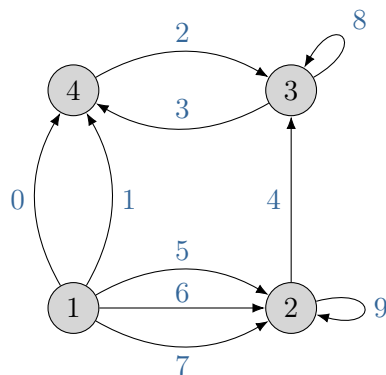


FIGURE 10.5. A directed graph as discussed in Example 10.18.

Example 10.18 Directed Graphs

Graphs are mathematical structures that describe pairwise relationships among various entities. We will consider various types of graphs as we proceed. See Interlude G for more, including further motivation and examples.

A graph consists of *nodes* (the entities) and *edges* (the relationships between pairs of entities). We specify a graph with three pieces of data: a set of nodes $\mathcal{N}$, a set of edges $\mathcal{E}$, and a function φ with domain $\mathcal{E}$ that gives for each edge a *description* in terms of the nodes that are *incident* to the edge (i.e., the nodes the edge relates). The elements of the sets $\mathcal{N}$ and $\mathcal{E}$ can be arbitrary and serve only to uniquely identify each node and edge. The function φ packages the graph's relationships, and its codomain depends on the type of graph. (For special types of graphs, we might add additional data to this specification, such as a function that gives a “label” for each node or a “weight” to each edge.) As a mathematical object, then, a graph is a tuple containing these data, e.g., we might talk about a graph $G = \langle \mathcal{N}, \mathcal{E}, \varphi \rangle$.

We allow the possibilities that there is more than one edge between any pair of nodes and that an edge (called a loop) can have the same node for source and target. If at most one edge is allowed, we have a *simple graph*; otherwise we have a *multigraph*. If loops are forbidden, we have a graph *without loops*; otherwise, we have a graph *with loops*;

In a *directed graph*, or **digraph**, the edge relationships are asymmetric, having one node as a “source” and another as the “target”. (For example, you

might like someone that does not like you or may be able to move from A to B but not vice versa. Similarly, a parent-child relationship is asymmetric.) A directed graph is thus specified by a 3-tuple $\langle \mathcal{N}, \mathcal{E}, \varphi \rangle$ where $\mathcal{N}$ is the set of nodes, $\mathcal{E}$ is the set of edges, and $\varphi: \mathcal{E} \rightarrow \mathcal{N} \times \mathcal{N}$ is the description function. Specifically, for an edge $e \in \mathcal{E}$, $\varphi(e) = \langle s, t \rangle$ where s is e 's source node and t is its target node. That φ returns a tuple encodes the asymmetry/direction of the edges from source to target. A loop is an edge e with $\varphi(e) = \langle n, n \rangle$ for some node n . When we allow loops and multiple edges between a pair of nodes, we have a *directed multigraph with loops*.

For instance, if we set $\mathcal{N} = [1..4]$, $\mathcal{E} = [0..10]$, and

$$\begin{array}{llll} \varphi(0) = \langle 1, 4 \rangle & \varphi(1) = \langle 1, 4 \rangle & \varphi(2) = \langle 4, 3 \rangle & \varphi(3) = \langle 3, 4 \rangle \\ \varphi(4) = \langle 2, 3 \rangle & \varphi(5) = \langle 1, 2 \rangle & \varphi(6) = \langle 1, 2 \rangle & \varphi(7) = \langle 1, 2 \rangle \\ & \varphi(8) = \langle 3, 3 \rangle & \varphi(9) = \langle 2, 2 \rangle, & \end{array}$$

then we get the directed graph in Figure 10.5, with nodes and edges labeled to make the correspondence with φ clear. Compare closely to see how φ represents this graph.

By restricting φ , we can put different constraint on the structure of the graph. For instance, if φ is forbidden from returning $\langle n, n \rangle$ for any node $n \in \mathcal{N}$, we get a graph *without loops*. If edges $e_1 \neq e_2$ implies $\varphi(e_1) \neq \varphi(e_2)$, then there can be at most one edge between any pair of nodes, and we get a *directed simple graph*, with or without loops. For a directed simple graph, we can choose to let $\mathcal{E}$ simply be a subset of $\mathcal{N} \times \mathcal{N}$ and φ the inclusion function incl . We explore these variations further in Chapters 13 and 14. See also upcoming Example 10.19.

Puzzle 93. A *bipartite* (“two part”) directed graph $\langle \mathcal{N}, \mathcal{E}, \varphi \rangle$ is one for which $\mathcal{N} = \mathcal{S} \sqcup \mathcal{T}$, where $\mathcal{S}$ and $\mathcal{T}$ are non-empty sets and $\varphi: \mathcal{E} \rightarrow \mathcal{S} \times \mathcal{T}$. Thus nodes in $\mathcal{S}$ can only be sources and nodes in $\mathcal{T}$ can only be targets.

What conditions on φ are required so that the relationships in this graph represent a *function* from $\mathcal{S}$ to $\mathcal{T}$? Put another way: what kinds of directed, bipartite graphs are models of functions between two finite sets?

Example 10.19 Undirected Graphs

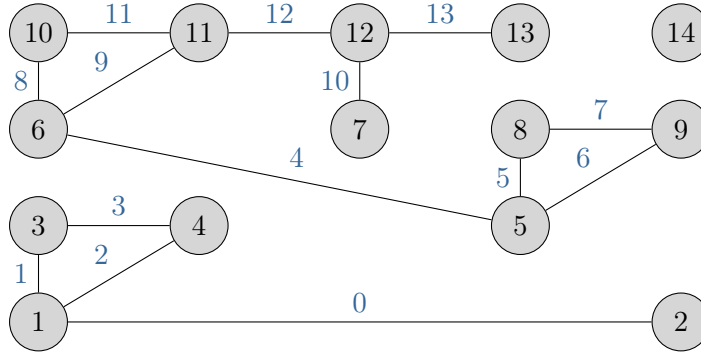


FIGURE 10.6. An undirected graph as discussed in Example 10.19.

An *undirected graph* is specified by general data described in Example 10.18, except in an undirected graph the edges are symmetric, with no preferred direction. (For example, by definition, if you are friends with someone, they are friends with you, and the relationship between students of taking the same class is also symmetric.) As such, rather than describing edges with tuples of nodes, in which order matters, we describe an edge in an undirected graph with a *set* of nodes.

An undirected graph is specified by a similar 3-tuple $\langle \mathcal{N}, \mathcal{E}, \psi \rangle$. Again $\mathcal{N}$ is the set of nodes, $\mathcal{E}$ is the set of edges, and again ψ is a function that converts an edge's unique identity (in $\mathcal{E}$) to a description of its incident nodes. Specifically, if $e \in \mathcal{E}$ is an edge, then either $\psi(e) = \{n_1\}$, giving a loop from node n_1 to itself, or $\psi(e) = \{n_1, n_2\}$ for nodes $n_1 \neq n_2$. So, using $\binom{\mathcal{N}}{k}$ to denote the set of subsets of $\mathcal{N}$ of cardinality exactly k , the type of ψ is $\mathcal{E} \rightarrow \binom{\mathcal{N}}{1} \cup \binom{\mathcal{N}}{2}$. We call this an *undirected multigraph with loops*.

For instance, if we set $\mathcal{N} = [1..14]$, $\mathcal{E} = [0..14]$, and

$$\begin{aligned}
 \psi(0) &= \{1, 2\} & \psi(1) &= \{1, 3\} & \psi(2) &= \{1, 3\} & \psi(3) &= \{3, 4\} \\
 \psi(4) &= \{5, 6\} & \psi(5) &= \{5, 8\} & \psi(6) &= \{5, 9\} & \psi(7) &= \{8, 9\} \\
 \psi(8) &= \{6, 10\} & \psi(9) &= \{6, 11\} & \psi(10) &= \{7, 12\} & \psi(11) &= \{10, 11\} \\
 \psi(12) &= \{11, 12\} & \psi(13) &= \{12, 13\}, & & & &
 \end{aligned}$$

then we get the graph shown in Figure 10.6, with nodes and edges labeled to make the correspondence with φ clear. Compare closely to see how ψ represents this graph. In this example, there are no loops or multiple edges, so this is an *undirected simple graph without loops*.

As with directed graphs, we can restrict ψ to vary the constraints on the

graph's structure. For instance, eliminating loops corresponds to requiring that $\text{range}(\psi) \subseteq \binom{N}{2}$, and eliminating multiple edges between a pair of nodes comes from requiring that for two edges $e_1 \neq e_2$, $\psi(e_1) \neq \psi(e_2)$. Consider carefully how these conditions restrict the graph; how would you change ψ above to add a loop at node 14 or multiple edges from nodes 1 to 2? Example 13.11 below continues this story.

Puzzle 94. Consider the graphs $G_i = \langle \mathbb{N}, \mathcal{E}_i, \varphi_i \rangle$ for $i \in [1..3]$, where $\mathbb{N}$ is the natural numbers and $\mathbb{P} \subseteq \mathbb{N}$ denotes the set of prime numbers.

1. In G_1 , n and m are incident on a common edge if $n = km$ for some $k \in \mathbb{N}$.
2. In G_2 , n and m are incident on a common edge if n and m share any prime factors. Assume that the set of prime factors for 0 and 1 is empty.
3. In G_3 , for every $p \in \mathbb{P}$, there is a distinct edge between n and m if $n \neq m$ and they share p as a prime factor.

Which of G_1, G_2, G_3 are directed graphs and which are undirected graphs? What are the types of each? What might $\mathcal{E}_3$ and φ_3 look like? Sketch a part of one or more of these graphs.

Example 10.20 Sequences and Streams

Example (vi) above defines a (numeric) sequence to be a function $a: \mathbb{N} \rightarrow \mathbb{R}$. (In computer science, these are also called *streams*.) This example begins a series of examples and puzzles that build on this definition; see Examples 10.20, 10.21, 10.22, and 10.23 and Puzzles 95, 96, 100, 101, 102, 103, and 110.

The purpose of this series is three-fold. First, sequences offer a wonderful illustration of how we can use functions to *package* information and how working with the package as a whole gives us more power and more insight than working individually with the pieces within it. Second, these ideas provide a well-equipped playground for practicing and sharpening the skills for working with and reasoning about functions. Finally, the tools we develop in these examples and puzzles will prove surprisingly potent for many problems in probability theory and beyond. So although this series takes us on something of a side road, it is worth taking your time and enjoying the ride.

Sequences are functions of type $\mathbb{N} \rightarrow \mathbb{R}$, and to make this interpretation more salient, we also use $\text{Seq}(\mathbb{R})$ as a short-hand for the set $\mathbb{N} \rightarrow \mathbb{R}$. In this example, we define several utility functions that take or return sequences: three *build*

Start of the “Sequences and Streams” series.

sequences, two *deconstruct* sequences into parts, and two create new sequences from old ones. For each, we describe the function and give its type and definition, where a, b are sequences, r is a real number, and $n \in \mathbb{N}$ is arbitrary.

1. **repeat** takes a number and returns a constant sequence that always returns the given number.
 $\text{repeat}: \mathbb{R} \rightarrow \text{Seq}(\mathbb{R})$ with $\text{repeat}(r) = \text{const}_r$.
2. **cons** takes a number and a sequence and returns a new sequence that makes the number the head and the given sequence the tail.
 $\text{cons}: \mathbb{R} \times \text{Seq}(\mathbb{R}) \rightarrow \text{Seq}(\mathbb{R})$ with $\text{cons}(r, a)(0) = r$ and $\text{cons}(r, a)(n+1) = a(n)$.
3. **lift** takes a number x and returns a sequence with x followed by zeros.
 $\text{lift}: \mathbb{R} \rightarrow \text{Seq}(\mathbb{R})$ with $\text{lift}(r) = \text{cons}(r, \text{repeat}(0))$.
4. **head** takes a sequence and returns the first value in the sequence
 $\text{head}: \text{Seq}(\mathbb{R}) \rightarrow \mathbb{R}$ with $\text{head}(a) = a(0)$.
5. **tail** takes a sequence and returns a new sequence obtained by dropping the first element of the given sequence.
 $\text{tail}: \text{Seq}(\mathbb{R}) \rightarrow \text{Seq}(\mathbb{R})$ with $\text{tail}(a)(n) = a(n+1)$.
6. **add** takes two sequences and returns a new sequence whose value at n is the sum of the values at n of the given sequences.
 $\text{add}: \text{Seq}(\mathbb{R}) \times \text{Seq}(\mathbb{R}) \rightarrow \text{Seq}(\mathbb{R})$ with $\text{add}(a, b)(n) = a(n) + b(n)$.
7. **scale** takes a number and a sequence and returns a sequence whose value at n is the value at n of the given sequence multiplied by the given number.
 $\text{scale}: \mathbb{R} \times \text{Seq}(\mathbb{R}) \rightarrow \text{Seq}(\mathbb{R})$ with $\text{scale}(r, a)(n) = ra(n)$.

Do not let the extra parentheses in some of these expressions confuse you. For instance, $\text{tail}(a)$ is a sequence, and thus a function, so for instance $\text{tail}(a)(0)$ is that function evaluated at 0. Similarly, $\text{repeat}(r)$ is a function also but the definition $\text{repeat}(r) = \text{const}_r$ needs no extra parentheses or evaluation because it is an equality of *functions*.

Think for a moment about what $\text{lift}(r)$ does. Via **cons**, it creates a sequence whose first element is r and whose tail is the sequence $\text{repeat}(0)$ that is all 0s. Thus, **lift** function maps a number to a sequence that is all zero except for the first term; $\text{lift}(r)(0) = r$ and $\text{lift}(r)(n+1) = 0$ for $n \in \mathbb{N}$. To understand **lift**'s

name, fill in an expression for the blanks in the following:

$$\begin{aligned}\text{add}(\text{lift}(x), \text{lift}(y)) &= \text{lift}(\_\_) \\ \text{scale}(r, \text{lift}(x)) &= \text{lift}(\_\_) \\ \text{add}(\text{lift}(x), \text{scale}(-1, \text{lift}(y))) &= \text{lift}(\_\_).\end{aligned}$$

The answers are, respectively, $x + y$, rx , and $x - y$. The range of `lift` is thus a subset of the set of sequences that acts like a copy of the real numbers. All the familiar operations of arithmetic on numbers carry over – are “lifted” – to act on these sequences in identical ways. See Aside “Lifting” on page 590.

Puzzle 95. Following up on Example 10.20, give the types of the following functions and show how to define them:

1. `map` takes a numeric function f and a sequence and returns a new sequence whose value at n is f applied to the value at n of the given sequence.
2. `zip` takes a function g of two numeric arguments and two sequences a and b and returns a new sequence whose value at n is g applied to $\langle a(n), b(n) \rangle$.
3. `convolve` takes two sequences a and b and returns a new sequence whose value at n equals $\sum_{k=0}^n a(k)b(n-k)$.
4. `interleave` takes two sequences and interleaves them starting with the first.
5. `iterate` takes a function $f: \mathcal{X} \rightarrow \mathcal{X}$ for some set $\mathcal{X}$ and a value $x \in \mathcal{X}$ and returns the sequence with values $x, f(x), f(f(x)), f(f(f(x))), \dots$. Note that this is *polymorphic* in the sense described earlier, we get a distinct function for each set $\mathcal{X}$.

Solutions are given later in the section; `sequences.py` is a Python implementation.

The previous puzzle and example illustrate several of the ways that we use functions to build our abstractions. The sequences here use a function to package an infinite list of numbers into a single object. Having such packages, we can then operate on the package as a whole, manipulate them in meaningful ways without having to pay too much attention to the individual values inside. We define functions to build such objects from simple ingredients (e.g., `repeat`, `lift`, `cons`) and functions to pull them apart to get at the insides (e.g., `head`, `tail`). As we develop intuition and facility for working with sequences, we will be able to do more with them. For instance, letting `double:  $\mathbb{R} \rightarrow \mathbb{R}$` be the function that doubles a number,⁶ we can

Part of the “Sequences and Streams” series.

⁶Using the notation in Chapter 11, we will be able to write `double` as $\langle x \rangle \mapsto 2x$, or as $(2\blacksquare)$ for short.

construct meaningful sequences such as

$t = \text{iterate}(\text{double}, 1)$	powers of two
$p = \text{convolve}(\text{repeat}(1), \text{repeat}(1))$	positive integers
$v = \text{cons}(0, \text{map}(\text{double}, p))$	<i>even</i> natural numbers.

From here, we can develop relationships among different sequences at a higher level, subsuming what we know about numbers, polynomials, and vectors in one swoop. For example, there is a unique sequence s that satisfies the equation $s = \text{cons}(1, s)$. What is it? The equation tells us that $s(0) = 1$ and $s(n+1) = s(n)$ for all $n \in \mathbb{N}$, so $s = \text{repeat}(1)$. This is analogous to a differential equation, and similar but more powerful equations will yield an entire discrete calculus for solving counting, probability, calculation, and other problems in elegant way. Our path through Probability Theory will follow an analogous trajectory.

Puzzle 96. Part of the power of our sequence representation is that we can define sequences through equations that describe the behavior of the sequence as a whole (rather than through details of the individual terms). Under minor restrictions, these solutions have unique solutions.

What can you say about the sequences determined by the equations:

$$\begin{aligned} f &= \text{cons}(0, \text{cons}(1, \text{add}(\text{tail}(f), f))) \\ c &= \text{interleave}(\text{repeat}(0), \text{add}(c, \text{repeat}(1)))? \end{aligned}$$

Consider the equations tell us about the first few terms and a general term of the sequence on the left hand side, and relate them to the corresponding terms on the other side. For example, what are $f(0), f(1), f(2)$? (Hint for c : search for “A007814”.)

As a hint for f , in the notation we introduce in Puzzle 100, it will be written

$$f = 0 :: 1 :: (f' + f).$$

Part of the “Sequences and Streams” series.

10.1 Common Operations

A function is a mathematical object in its own right, distinct from the values it returns, and there are various operations we commonly perform on functions.

- *Equality.* We say that two functions f, g are equal, denoted $f = g$, if they have the same domain and codomain and if $f(x) = g(x)$ for every x in their common domain. Otherwise, the functions are not equal, denoted $f \neq g$.

Note in particular, that we can have $f \neq g$ and $f(x) = g(x)$ for some x .

- *Comparison.* If a function's codomain is an ordered set with an order relation $\leq$, we write $f \leq g$ if f and g share the same domain and codomain and if $f(x) \leq g(x)$ for every x in the common domain. An analogous statement holds for order relations like $<$, $\geq$, and $>$.

For example, a function f satisfies $f \geq \text{const}_0$ when $f(x) \geq 0$ for all x . We usually just write this as $f \geq 0$, with slight abuse of notation.

- *Composition.* Composition is when we pass the returned value from one function as input to another. See the next Chapter for details.
- *Image and Inverse Image.* Think back to Example 10.8, where we think of a function $f: \mathcal{A} \rightarrow \mathcal{B}$ as describing a way to distribute (labeled) apples into (labeled) buckets. Given a set of apples $\mathcal{A}_1 \subseteq \mathcal{A}$, we might want to know which buckets received those apples. Given a set of buckets $\mathcal{B}_1 \subseteq \mathcal{B}$, we might want to know which apples were assigned to those buckets. The former is the *image* of $\mathcal{A}_1$ under f , and the latter is the *inverse image* of $\mathcal{B}_1$ under f .

More formally, if $f: \mathcal{A} \rightarrow \mathcal{B}$, then we get two functions:

- the **image** $f^\rightarrow: 2^{\mathcal{A}} \rightarrow 2^{\mathcal{B}}$, and
- the **inverse image** $f^\leftarrow: 2^{\mathcal{B}} \rightarrow 2^{\mathcal{A}}$,

where

$$f^\rightarrow(\mathcal{A}_1) = \{f(a) \mid a \in \mathcal{A}_1\} \quad (10.3)$$

$$f^\leftarrow(\mathcal{B}_1) = \{a \in \mathcal{A} \mid f(a) \in \mathcal{B}_1\}. \quad (10.4)$$

The image gives us the set of return values for a set of inputs, and the inverse image gives the set of inputs whose return values lie in a set of possible outputs. These must satisfy:

$$f^\rightarrow(\mathcal{A}_1) \subseteq \mathcal{B}_1 \text{ if and only if } \mathcal{A}_1 \subseteq f^\leftarrow(\mathcal{B}_1), \quad (10.5)$$

and $f^\rightarrow(\{\}) = \{\} = f^\leftarrow(\{\})$.

- *Arithmetic.* For functions whose codomains support standard arithmetic operators (e.g., addition, multiplication), we “lift” those operators from acting on

elements of the codomain to acting on the functions themselves.⁷

⁷See Aside on “Lifting” on page 590.

For example, with functions $f, g: \mathcal{D} \rightarrow \mathbb{R}$, we can define functions of type $\mathcal{D} \rightarrow \mathbb{R}$ corresponding to the usual arithmetic operations:

$$(f + g)(x) = f(x) + g(x) \quad (10.6)$$

$$(f - g)(x) = f(x) - g(x) \quad (10.7)$$

$$(fg)(x) = f(x)g(x) \quad (10.8)$$

$$(f/g)(x) = f(x)/g(x) \quad (10.9)$$

$$(f^z)(x) = (f(x))^z \quad (10.10)$$

$$|f|(x) = |f(x)|, \quad (10.11)$$

and so forth, when such operations are defined. The same idea (for some operators) works with other codomains (cf., Chapter 15).

- *Function Operators.* There is a variety of common operators that act on functions, like differences, derivatives, sums, and integrals. We will encounter some others as well. (See Section 10.3.) Many important examples are *linear operators* that take numeric or vector-valued functions and satisfy $T(af + bg) = aT(f) + bT(g)$ for operator T , numbers a, b , and functions f, g . This just means that the operator is a linear function on its domain.
- *Linear Combinations.* If we have several measurements of some quantity $q_1, q_2, \dots, q_k$, we might want to take a weighted average of the measurements to get a representative estimate of the quantity. This is a combination $\sum_{i=1}^k w_i q_i$ for some weights $w_1, \dots, w_k$ that add to 1. Lifting this operation to functions just means taking the combination of the functions’ returned values.

For functions $f_1, f_2, \dots, f_k$ on the same domain, a **linear combination** is a function $c_1 f_1 + c_2 f_2 + \dots + c_k f_k$ on their common domain, for any real constants $c_1, c_2, \dots, c_k$, and is defined as you might expect by

$$(c_1 f_1 + c_2 f_2 + \dots + c_k f_k)(x) = c_1 f_1(x) + c_2 f_2(x) + \dots + c_k f_k(x) \quad (10.12)$$

for any valid input x . This assumes that the functions’ codomain supports addition and multiplication by real constants, e.g., when the functions return real numbers or tuples of real numbers. We call the c_i ’s the coefficients or weights of the linear combination.

In the special case where the coefficients satisfy $c_i \geq 0$ for $i \in [1..k]$ and $c_1 + c_2 + \cdots + c_k = 1$, as in a weighted average, the linear combination is called a **convex combination**.

If T is a linear operator on functions, as described in the previous item, then

$$T(c_1f_1 + \cdots + c_kf_k) = c_1T(f_1) + c_2T(f_2) + \cdots + c_kT(f_k),$$

so T maps linear combinations to linear combinations.

- *Restriction.* If $f: \mathcal{A} \rightarrow \mathcal{B}$ and $\mathcal{C}$ is a strict subset of $\mathcal{A}$, then the **restriction** of f to $\mathcal{C}$ is the function $\mathcal{C} \rightarrow \mathcal{B}$ with value $f(x)$ for $x \in \mathcal{C}$. Note that we consider the restriction of f and the original f to be *different functions* because they have different types, even though they agree on the overlap of their domains. If $\text{range}(f) \subseteq \mathcal{D} \subset \mathcal{B}$, we can similarly restrict the output type of f , writing $f: \mathcal{A} \rightarrow \mathcal{D}$.
- *Extension.* Conversely, if $f: \mathcal{A} \rightarrow \mathcal{B}$ and $\mathcal{A}$ is a strict subset of $\mathcal{C}$, it is sometimes possible to extend f to a function on $\mathcal{C}$ that matches f on $\mathcal{A}$.

Puzzle 97. Create small sets of apples and buckets and a function $f: \mathcal{A} \rightarrow \mathcal{B}$. Find $f^+(\mathcal{A}_1)$ for two different subsets of apples and $f^-(\mathcal{B}_1)$ for two different subsets of buckets.

Puzzle 98. Convex combinations are weighted averages. What is the set of all convex combinations of the numbers 0 and 1? Of the numbers a and b with $a < b$?

What is the set of all convex combinations of the three points in $\mathbb{R}^2$: $\langle 0, 0 \rangle$, $\langle 0, 1 \rangle$ and $\langle 1, 0 \rangle$? What is the set of all convex combinations of the four points in $\mathbb{R}^3$: $\langle 0, 0, 0 \rangle$, $\langle 0, 0, 1 \rangle$, $\langle 0, 1, 0 \rangle$, and $\langle 1, 0, 0 \rangle$? Feel free to draw a picture to answer. (Here, we add and scale the points component-wise, e.g., $c\langle a, b \rangle + d\langle x, y \rangle = \langle ca+dx, cb+dy \rangle$, and similarly for points in $\mathbb{R}^3$.)

Describe in a sentence or two the set of all convex combinations of three arbitrary points in $\mathbb{R}^2$?

Fundamentals

Identity	id	Chapter 10, page 562
Constant	const_c	Chapter 10, page 562
Anonymous	$\langle x \rangle \mapsto \dots x \dots; (\dots \blacksquare \dots)$	Chapter 11
Indicator	basic and short-hand	Chapter 12
Image, Inverse Image	$f^{\rightarrow}, f^{\leftarrow}$	Page 581

Common Mathematical

Exponentials and Logarithms	$\exp, \log, \ln, \lg, \log_b$	B.2
Trigonometric	$\sin, \cos, \tan$ $\arcsin, \arccos, \arctan$	B.3
Minima and Maxima	$\max, \vee, \min, \wedge$ $\operatorname{argmax}, \operatorname{argmin}, \sup, \inf$	B.5
Absolute value/magnitude/modulus:	$ x $	B.5, Section 14.2 page 666

Integer and Counting

Floor and Ceiling	$\lfloor x \rfloor$ and $\lceil x \rceil$	B.4
Integer division	$\text{div}, \text{mod}, \backslash, \chi$	
Factorial and Gamma Function	$n!$ and Γ	B.4
Falling powers	$x^{\underline{n}}$	B.4, Ex 10.23
Binomial/Multinomial coefficients	$\binom{r}{k}$ and $\binom{n}{a,b,c}$	Chapter 18, B.4
Multi-choose	$\binom{n}{k}$	Chapter 18
Stirling Numbers	$[n]_k$ and $\{n\}_k$	Chapter 18

Tuple/Vector-related

Projections	$\text{proj}, \overline{\text{proj}}$	Chapter 15
Combining	$\times, \otimes, \boxtimes, ::$	Chapter 15
Fork	$\curlyvee$	Chapter 15
Permutations	permute	Chapter 15

Matrix-related

Determinant	$\det$	Section 17.3
Trace	tr	Section 17.3
Diagonal	diag	Section 17.3

TABLE 10.4. Commonly used special functions and operators, with references to the sections in which they are defined. See also the Help Sheet in Appendix H.

10.2 Special Functions

In practical work, we often define functions as we need them, and discard them just as readily. For instance, in this text, we often define functions in an example or a section or even a paragraph that are intended only to hold during that example or section or paragraph. Like local variables in programming or the local variables we use for parameters, sums, integrals, etc., these definitions are *local* over some extent where they are needed. So, we might at several separated points write “let f be a function ... where ...” for different functions, making that distinction clear from context.

However, there are several “special functions” that we define and name once and for all and use repeatedly. Some of these are familiar mathematical functions like `sin` and `log` and others are functions we have already defined, like `id` and `const`. Table 10.4 lists the special functions we make regular use of throughout this text along with references to where in the text those functions are defined or discussed. The Help Sheet (Appendix H) also has useful information and identities.

10.3 Operators

In our early experiences with arithmetic, we are introduced to operators: addition (+), multiplication ($\cdot$), subtraction ($-$), and division ($\div$). At that point, operators seem fixed and special, but now we see more clearly. Our goal in this Section is to see that most⁸ of the operators we use are just “syntactic sugar” for – you guessed it – *functions*. With this perspective, we will explore how operators tend to be used and useful, and we will see why it is sometimes valuable to define our own operators.

Let’s begin with the humble addition operator $+$ on real numbers. We think of this as having two “holes” to be filled in with numbers, $\blacksquare + \blacksquare$. Because there are two holes to fill, we call $+$ a *binary operator*.⁹ When the holes are filled in with numbers, we get a number back. This seems familiar, right? Indeed, $+$ is just a function $+: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ that takes two numeric arguments¹⁰ and returns a number. We could write $+(a, b)$ like we ordinarily do when evaluating function, but by convention and practice, we use “infix” syntax with the function name ($+$) between its arguments. We can tell a similar story with the other familiar, arithmetic operators, like subtraction, multiplication, division, negation, and exponentiation.

⁸A few operators, like $=$ and $\leq$, are best thought of as *relations*; see Chapter 16.

⁹An operator with one hole is called *unary*. There are a few operators with “arity” bigger than two, but it is not common.

¹⁰Or equivalently, one 2-tuple of numbers

SYNTAX. One reason for having operators – with their special syntax – is making commonly used functions easy to write, read, and manipulate. An expression like $a + b + c + d$ is more convenient than something like $\text{add}(a, \text{add}(b, \text{add}(c, d)))$ and also more clearly reflects properties like associativity.

Binary operators are usually placed between their arguments, though there are exceptions. Exponentiation, for instance, is denoted implicitly by *position* rather than having an explicit operator, with the exponent in a superscript with smaller type, like 2^3 . The underlying function $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ is still there, however. Multiplication is often denoted by juxtaposition ab rather than with the multiplication operator¹¹ $a \cdot b$, especially with variables involved. For instance, we write $4xy$ rather than $4 \cdot x \cdot y$ for the product, though we might write $4 \cdot 12$ with just numbers for clarity. And the binomial coefficient operator, $\binom{r}{k}$,

¹¹Multiplication-like operators such as $\times$ and $*$, familiar from our early arithmetic experience, are typically reserved for other operations.

There is more variation with unary operators. Some, like negation, are applied before their argument; others, like factorial, are applied after; and many, like absolute value $|x|$ or floor $\lfloor x \rfloor$, surround their argument. Some unary operators are represented by accents, superscripts, subscripts, or several at once. Examples include complex conjugate $\bar{z}$, matrix transpose $M^\top$, falling factorial powers $x^{\underline{k}}$, and the image/inverse image operators on page 581. (See Appendix B for more on these operations.)

One downside of using operators in this way is the possibility of syntactic ambiguity. Is $2 + 3 \cdot 6$ equal to 20 or 30? Which is applied first, the addition or multiplication? This problem is typically solved with two overlapping solutions: using delimiters like parentheses for explicit grouping and establishing a *convention* on the order of operations, giving higher precedence to some operators over others. The typical convention is that multiplication, division, and negation bind more tightly than addition and subtraction and that exponentiation binds more tightly on the base than any of them (e.g., $2c^4$ is $2(c^4)$ not $(2c)^4$). Programmers face this issue with operators in their language, and most programming languages have a defined order of operations (precedence) as well.

OPERATORS ON FUNCTIONS. Another place where operators are plentiful is when we transform functions in various ways. For example, for many functions, we can take derivatives, integrals, sums, and differences, sometimes producing a number and other times producing another function. Such operators are familiar enough that it can obscure that these are just *functions*, functions that take a function as input and sometimes also return a function as output.¹²

¹²See examples x and xi earlier.

Consider as an example the derivative operator, which is denoted in various

ways, e.g., f' , Df , or ∇f , depending on the context. If f is a differentiable function on some domain, then its derivative Df is a *function*¹³ that gives a local linear approximation to f . When $f: \mathbb{R} \rightarrow \mathbb{R}$, for instance, we have for t near x that $f(t) \approx f(x) + Df(x)(t - x)$. The key point is that D *itself is a function*; it takes a differentiable function as input and returns a function (not necessarily differentiable) as output. When $f: \mathbb{R} \rightarrow \mathbb{R}$, for instance, $D: \mathcal{D}^1(\mathbb{R}, \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$, where $\mathcal{D}^1(\mathbb{R}, \mathbb{R})$ is the set of differentiable functions $\mathbb{R} \rightarrow \mathbb{R}$.

¹³We write $Df(x)$ for the value of the function at x , meaning $(Df)(x)$.

A function like D that takes and/or returns a function is called a **higher-order function**. Once we recognize that D is a function, we can consider its properties like any other function. We know, for instance, that for numbers α, β and $f, g \in \mathcal{D}^1(\mathbb{R}, \mathbb{R})$:

$$D(\text{const}_\alpha) = 0 \quad (10.13)$$

$$D(\text{id}) = 1 \quad (10.14)$$

$$D(\alpha f + \beta g) = \alpha D(f) + \beta D(g) \quad (10.15)$$

$$D(fg) = fD(g) + D(f)g. \quad (10.16)$$

The first two equations tell us the derivative for constant and identity functions; the third equation says that the derivative is *linear* in its argument; and the last tells us how to compute the derivatives of products. The point here is not about the derivative *per se* but that we can understand a lot about functions – even higher-order functions – by properties like this. Chapter 17 dives deeper into this idea.

Several related operators arise frequently:

- The definite sum operator $\sum_{[a..b]}$ is defined for integers $a \leq b$ by

$$\sum_{[a..b]} f = \sum_{k=a}^{b-1} f(k),$$

which is zero if $a = b$. This is a mathematical version of the following Python `for`-loop:

```
total = 0
for k in range(a, b):
    total += f(k)
```

where the sum is the final value of `total`. The sum operator has many convenient algebraic properties that follow from properties of addition. For instance, because addition commutes, $\sum_{[a..b]} (f + g) = \sum_{[a..b]} f + \sum_{[a..b]} g$ and for $a \leq$

$b \leq c$, $\sum_{[a..b]} f + \sum_{[b..c]} f = \sum_{[a..c]} f$ for any f , so $\sum_{[a..b]} + \sum_{[b..c]} = \sum_{[a..c]}$.

- The definite integral operator $\int_a^b$, for real numbers $a \leq b$, acts much like a sum. For integrable functions on $[a..b]$, we define

$$\int_a^b f = \int_{x=a}^b f(x) \, dx,$$

which satisfies properties analogous to the sum.

- From sums and integrals, we can define weighted average operators. If $w: [a..b] \rightarrow [0..)$ is a non-negative function with w positive for a non-trivial range of inputs, we can define

$$A_{w,a,b}(f) = \frac{\sum_{[a..b]} fw}{\sum_{[a..b]} w}$$

$$M_{w,a,b}(f) = \frac{\int_a^b fw}{\int_a^b w},$$

where the integrals are only defined if fw and w are integrable. Many of the calculations we do in probability have this form.

- More generally, we can use sums and integrals to construct *transforms* that map functions to functions. These look like:

$$S(f)(v) = \sum_{u=a}^{b-1} f(u) L(u, v)$$

$$T(f)(y) = \int_{x=a}^b f(x) K(x, y) \, dx,$$

for some functions L and K . Both $S(f)$ and $T(f)$ are themselves functions, so when we write $T(f)(y)$, say, we are evaluating the function $T(f)$ at the value y .

- A discrete analogue of the derivative is the **difference** operator Δ , where Δf is defined by $(\Delta f)(x) = f(x+1) - f(x)$. We can apply both difference and derivative multiple times (composition, cf. Chapter 13), giving the k th derivative and difference operators D^k and Δ^k ; for instance, $\Delta^2 f(x) = f(x+2) - 2f(x+1) + f(x)$. The difference operator satisfies algebraic laws similar to the derivative and plays an important role in the discrete calculus of sequences and sums we explore in Puzzles 95, 96, 100, 101, and 102. We can apply both difference and derivative multiple times giving the k th derivative and difference

operators. The former are denoted with primes for $k \leq 3$, like f'' and f''' , and with a superscript $f^{(k)}$ otherwise. The latter is denoted Δ^k ; for instance, $\Delta^2 f(x) = f(x+2) - 2f(x+1) + f(x)$. Note that the operator binds more tightly than evaluation, so this is the same as $(\Delta^2 f)(x)$.

Appendix B offers a further discussion of sums, integrals, and derivatives.

Puzzle 99. Define an operators $A, Z: ([0\_1] \rightarrow \mathbb{R}) \rightarrow \mathbb{R}$ by

$$Z(f) = f(0)$$

$$L(f) = \max \{y \in [-\infty\_] \mid y \leq f(x) \text{ for all } x \in [0\_1] \}.$$

1. What can you say about $Z(\text{const}_c)$ and $Z(f+g)$?
2. What can you say about $L(\text{const}_c)$ and $L(f^2)$ when $f(0) = 0$?
3. Suppose $\ell \leq f \leq u$ for real numbers ℓ, u and let $w = \text{const}_1$. (Recall that this means $\ell \leq f \leq u$ for all x in $[0\_1]$.) Can $A_{w,0,1}(f)$ be smaller than ℓ or bigger than u ? What about $M_{w,0,1}(f)$?

OPERATOR OVERLOADING. In many situation, it is useful to define new operators that represent functions that are especially meaningful or commonly used. Table 10.5, for example, defines several operators for manipulating sequences, in the context of that series of examples and puzzles, that make it easier to write and manipulate sequence equations in comparison to named functions. It is not practical to design a new glyph for each new operator we create, so we usually draw from the repository of existing symbols, reasonably large. Nonetheless, we often find ourselves using the same operator symbol to have *different meaning in different contexts*. In programming terms, we have **overloaded** the operators.

There are three ways this happens. First, we appropriate an operator symbol in our context independently of how it may be used elsewhere, usually trying to make the symbol mnemonic or visually consistent with the operation. For instance, in Table 10.5, $a \vee b$ interleaves sequences a and b taking terms alternately from each. The symbol shows two threads merging into one, which reflects what the operator does. A second way we overload is choosing an operator symbol that is often used in a conceptually similar way. For instance, we denote the independent join of two FRPs by $X \star Y$; here $\star$ is often used as a kind of “product” and indeed independent join is conceptually a product operation in an important sense. Finally, we often overload operators that perform a directly operation in a different context. For example, we

use $+$ for the addition of numbers, vectors, matrices, functions, among other things. Not only is this conceptually a “sum” in these cases but it is addition in a stronger sense: the operation satisfies most or all of the usual algebraic laws that we expect from addition. Here the overloading can be helpful in remembering the meaning of the operations.¹⁴

¹⁴However, as we move farther from the original, we must beware the uncanny valley. A $+$ operator that is not commutative might be surprising for instance.

Aside: Lifting. Operator overloading is often a special case of a more general phenomenon where a function on one kind of object produces an analogous function on a composite or more complex kind of object. We think of the original function as operating on a “lower-level” object and the analogous function as operating on a “higher-level” object. We say that the lower-level function has been *lifted* to the higher level.

For example, we *lift* addition from numbers to vectors, matrices, and functions. In Part I, we lift the transformation by statistics, joins, and observational constraints from FRPs to their Kinds.

SYNTHESIS. In the remainder of this Chapter, we pull together many of the ideas we have discussed so far in a deep dive into the “Sequences and Streams” series of examples and puzzles. Answers to selected puzzles from the whole Chapter follows.

Recall that a sequence (aka stream) a is a function $a: \mathbb{N} \rightarrow \mathbb{R}$, and we use $\text{Seq}(\mathbb{R})$ as a short-hand for the set $\mathbb{N} \rightarrow \mathbb{R}$. We write the n th term in $a \in \text{Seq}(\mathbb{R})$ by $a(n)$ or a_n interchangeably. Earlier examples and puzzles defined several functions to create, deconstruct, and manipulate sequences, and Table 10.5 defines a variety of useful operators on sequences.

Example 10.21 Sequences as Power Series

In Puzzles 95 and 96, we defined various functions on sequences. Here, we express some of these functions as *operators* and use them to solve some interesting sequence equations. This and the remaining puzzles in this series do get into the weeds, so these are discretionary – but fun and worthwhile. The story continues below, see e.g., Puzzles 100, 101, 102, and 103.

Table 10.5 defines a variety of operators on sequences that we will use and build on in the remainder of this series. To start, define the special sequence

$$z = 0 :: \hat{1} \quad (10.19)$$

Part of the “Sequences and Streams” series.

Operation	Operator	Definition
lift	$\hat{}$	$\hat{r} = \text{lift}(r)$, as a <i>sequence</i> r is shorthand for $\hat{r}$
repeat	$*$	$r^* = \text{repeat}(r)$
tail	$'$	$a' = \text{tail}(a)$
cons	$::$	$r :: a = \text{cons}(r, a)$
Product	juxtaposition	$ab = \text{convolve}(a, b)$, $(ab)_n = \sum_{k=0}^n a_k b_{n-k}$
Reciprocal	$^{-1}$	if $a(0) \neq 0$, unique a^{-1} exists, $aa^{-1} = \hat{1} = a^{-1}a$
Powers	exponent	$a^0 = \hat{1}$, $a^{n+1} = aa^n$
interleave	Υ	$a \Upsilon b = \text{interleave}(a, b)$
scale	juxtaposition	$(ra)(n) = ra(n)$, also $ra = \hat{r}a$
Negation	juxtaposition	$-a = -1a$
add	$+$	$(a+b)(n) = a(n) + b(n)$, also $a+b = \text{zip}(+, a, b)$
Subtraction	$-$	$(a-b)(n) = a(n) - b(n)$, also $a-b = a+ -b$
Shuffle Product	$\otimes$	$(a \otimes b)_n = \sum_{k=0}^n \binom{n}{k} a_k b_{n-k}$
Shuffle Inverse	$^{[-1]}$	unique $a^{[-1]}$ exists when $a(0) \neq 0$
Shuffle Power	[exponent]	$a^{[0]} = \hat{1}$, $a^{[n+1]} = a \otimes a^{[n]}$
Parallel Product	$\cdot$	$(a \cdot b)(n) = a(n)b(n)$, also $a \cdot b = \text{zip}(\cdot, a, b)$
Difference	Δ	$\Delta a = a' - a$
Partial Sums	$\sum$	$\sum a = 0 :: a1^*$

TABLE 10.5. Sequence operators used in the “Sequences and Streams” series of examples and puzzles, including Examples 10.21, 10.22, and 10.23 and Puzzles 100, 101, 102, and 103.

Here, a, b are sequences and r is a real number. We write the value of a sequence a at n as $a(n)$ or a_n , as preferred in any instance. In a sequence context, treat r as shorthand for $\hat{r}$.

Order of operations is similar to standard, from tightest binding to loosest: $\hat{}/*/'$; powers; products; addition, subtraction, sums, difference; and finally, $::$ and Υ .

The Product and Shuffle Product are commutative, associative, distribute over $+$, have $\hat{1}$ as an identity, and annihilate with $\hat{0}$. Repeated products of both types give powers with natural number exponents. A *unique* reciprocal exists whenever $a(0) \neq 0$, which also allows negative exponents. The Product and Shuffle Product, respectively, uniquely satisfy

$$(ab)_0 = a_0 b_0 \qquad (ab)' = a'b + \hat{a}_0 b', \qquad (10.17)$$

$$(a \otimes b)_0 = a_0 b_0 \qquad (a \otimes b)' = a' \otimes b + a \otimes b'. \qquad (10.18)$$

These equations in fact define the respective products.

Addition, subtraction, and parallel product are the ordinary operations on functions.

which represents $\langle 0, 1, 0, 0, \dots \rangle$, a seemingly innocuous sequence that will be quite useful.

Referring to the table, we can establish a few interesting facts:

1. If $a \in \text{Seq}(\mathbb{R})$, then by the definition of the product, the sequence za has, for $n \in [0..)$,

$$\begin{aligned} (za)(0) &= \sum_{k=0}^0 z_0 a_0 = 0 \\ (za)(n+1) &= \sum_{k=0}^{n+1} z_k a_{n-k} = a_{n-1}, \end{aligned}$$

because $z_1 = 1$ is z 's the only non-zero value. Hence, we have *shifted* the sequence a to the right: $za = 0 :: a$! If a represents $\langle a_0, a_1, a_2, \dots \rangle$, then za represents $\langle 0, a_0, a_1, a_2, \dots \rangle$.

2. It follows that the sequence $z^n a$ for $n \in [0..)$ and $a \in \text{Seq}(\mathbb{R})$ just shifts a to the right n times, prepending n zeros. And in particular, z^n represents the sequence with n zeroes, followed by 1, followed by the all zeroes.
3. For any sequence a , we have the following identity

$$a = \hat{a}_0 + z a'. \quad (10.20)$$

This holds because $\text{head}(a) = a_0 = \text{head}(\hat{a}_0 + z a')$ and because

$$\text{tail}(a) = a' = \text{tail}(0 :: a') = \text{tail}(z a') = \text{tail}(\hat{a}_0 + z a').$$

Let b be the sequence $\langle 1, 2, 3, 4, 0, 0, \dots \rangle$. Apply equation (10.20) repeatedly to get: $b = z^0 + 2z^1 + 3z^2 + 4z^3$. The right-hand side here looks and *acts* like a polynomial, but it is a sequence. Similarly, by repeated application of equation (10.20), we can express any sequence $a \in \text{Seq}(\mathbb{R})$ as

$$a = \sum_{n=0}^{\infty} a_n z^n. \quad (10.21)$$

This sum looks and acts like a power series, but it is a *sequence*. We can manipulate it in all the familiar ways as a purely algebraic object without us needing to establish any notion of convergence. So, we call this a *formal power series*.

The summation operator in equation 10.21 is not the Σ operator from the table, just ordinary summation.

For example, computing the product of $a, b \in \text{Seq}(\mathbb{R})$ by the formula in Table 10.5 and by multiplying out the power series in the usual way give the same result:

$$ab = \sum_{n=0}^{\infty} \left(\sum_{k=0}^n a_k b_{n-k} \right) z^n. \quad (10.22)$$

Similarly, we can supplement Table 10.5 with a “derivative” operator

$$\frac{da}{dz} = (z \otimes a')'. \quad (10.23)$$

Applying this to 10.21 gives us

$$\frac{da}{dz} = \sum_{n=1}^{\infty} n a_n z^{n-1}, \quad (10.24)$$

and from this we get more

$$\frac{dab}{dz} = \frac{da}{dz} b + a \frac{db}{dz} \quad (10.25)$$

$$\frac{da^{-1}}{dz} = -a^{-2} \frac{da}{dz}, \quad (10.26)$$

mimicking the ordinary product and reciprocal rules for derivatives. We use all these results in the next two puzzles, which are worth doing.

These last two results follow from equation 10.22 and rearranging terms:

$$\begin{aligned} \frac{dab}{dz} &= \sum_{n=1}^{\infty} n \left(\sum_{k=0}^n a_k b_{n-k} \right) z^{n-1} \\ &= \sum_{n=1}^{\infty} \left(\sum_{k=0}^n n a_k b_{n-k} \right) z^{n-1} \\ &= \sum_{n=1}^{\infty} \left(\sum_{k=0}^n (k + n - k) a_k b_{n-k} \right) z^{n-1} \\ &= \sum_{n=1}^{\infty} \left(\sum_{k=0}^n k a_k b_{n-k} + \sum_{k=0}^n a_k (n - k) b_{n-k} \right) z^{n-1} \\ &= \sum_{n=1}^{\infty} \left(\sum_{k=0}^n k a_k b_{n-k} \right) z^{n-1} + \sum_{n=1}^{\infty} \left(\sum_{k=0}^n a_k (n - k) b_{n-k} \right) z^{n-1} \\ &= \left(\sum_{k=1}^{\infty} k a_k z^{k-1} \right) \left(\sum_{j=0}^{\infty} b_j z^j \right) + \left(\sum_{j=0}^{\infty} a_j z^j \right) \left(\sum_{j=1}^{\infty} j b_j z^{j-1} \right) \end{aligned}$$

You can skip this derivation if you prefer.

$$= \frac{da}{dz}b + a\frac{db}{dz}.$$

Then, when a^{-1} exists (i.e., $a(0) \neq 0$), take derivatives on both sides of $\widehat{1} = aa^{-1}$:

$$0 = \frac{daa^{-1}}{dz} = \frac{da}{dz}a^{-1} + a\frac{da^{-1}}{dz},$$

and multiplying through by a^{-1} gives equation (10.26).

Puzzle 100. Building on Example 10.21 and Table 10.5:

1. Show that $(1 - z)1^* = \widehat{1}$. (Hint: consider each term in the difference.)
2. Apply the result in the previous item and the definition and uniqueness of the sequence reciprocal to show that

$$(1 - z)^{-1} = 1^* = \sum_{n=0}^{\infty} z^n. \quad (10.27)$$

3. If r is a real number, show that

$$(1 - rz)^{-1} = \sum_{n=0}^{\infty} r^n z^n. \quad (10.28)$$

(Hint: take the product of both sides with $\widehat{1} - rz$ and simplify.)

4. Show, both with and without equation (10.26), that

$$\frac{d}{dz}(1 - z)^{-1} = (1 - z)^{-2}. \quad (10.29)$$

5. If $a \in \text{Seq}(\mathbb{R})$ and $b = (1 - z)^{-1}a$, express b_n in terms of $a_0, \dots, a_n$. (Hint: use the definition of product and #2; collect common terms and add.)
6. What does the sequence $(1 + z)^n$ look like for $n \in [0..)$? Find it explicitly for a few small n to see the pattern.

Recall from the Table that when we write $1 - z$, $1 - rz$ and $1 + z$, these mean $(\widehat{1} - z)$, $\widehat{1} - rz$, and $(\widehat{1} + z)$ because 1 is a short-hand for $\widehat{1}$ in a sequence context.

Part of the “Sequences and Streams” series.

Example 10.22 Sequences and Recurrence Relations

Call $d \in \text{Seq}(\mathbb{R})$ *divisible by* z if $d(0) = 0$. (Recall the definition of z from equation (10.19) in Example 10.21.) If $c, d \in \text{Seq}(\mathbb{R})$ and d is divisible by z , then

Part of the “Sequences and Streams” series.

the sequence equation

$$a = da + c \quad (10.30)$$

has a unique solution $a \in \text{Seq}(\mathbb{R})$ given by

$$a = (1 - d)^{-1}c = c \sum_{n=0}^{\infty} d^n. \quad (10.31)$$

Notice that $(1 - d)^{-1}$ exists because $d(0) = 0$.

As an example, let $d = z + z^2$ and $c = z$. The equation (10.30), written term by term, is

$$\begin{aligned} a_0 &= 0 \\ a_1 &= 1 \\ a_n &= a_{n-2} + a_{n-1} \quad \text{for } n \in [2..). \end{aligned}$$

But this recurrence relation defines the famous [Fibonacci numbers](#) 0, 1, 1, 2, 3, 5, 8, 13, 21, ... Let's look at the solution: $a = (1 - z - z^2)^{-1}z$, which we can write as

$$a = \frac{z}{1 - z - z^2} \quad (10.32)$$

for clarity.

Now define the numbers $\gamma = (\sqrt{5} + 1)/2$ and $\bar{\gamma} = (\sqrt{5} - 1)/2$, with $\gamma\bar{\gamma} = 1$ and $\gamma - \bar{\gamma} = 1$. We have $1 - z - z^2 = (1 - \gamma z)(1 + \bar{\gamma} z)$ by just expanding out the right-hand side. Similarly, we have that

$$\frac{1}{(1 - \gamma z)(1 + \bar{\gamma} z)} = \frac{\gamma/\sqrt{5}}{1 - \gamma z} + \frac{\bar{\gamma}/\sqrt{5}}{1 + \bar{\gamma} z}. \quad (10.33)$$

This is the *partial fractions* expansion of the sequence.

To see this, multiply both sides by $(1 - \gamma z)(1 + \bar{\gamma} z)$ and get the sequence $\hat{1}$; since inverses are unique, the two sides are equal. For instance, on the right we get

$$\gamma(1 + \bar{\gamma}z)/\sqrt{5} + \bar{\gamma}(1 - \gamma z)/\sqrt{5} = \frac{1}{\sqrt{5}}(\gamma + \bar{\gamma})\hat{1} = \hat{1}.$$

It follows that

$$a = \frac{\frac{\gamma}{\sqrt{5}}z}{1 - \gamma z} + \frac{\frac{\bar{\gamma}}{\sqrt{5}}z}{1 + \bar{\gamma} z}$$

$$\begin{aligned}
&= \frac{1}{\sqrt{5}} \sum_{n=1}^{\infty} \gamma^n z^n + \frac{1}{\sqrt{5}} \sum_{n=1}^{\infty} (-1)^{n-1} \bar{\gamma}^n z^n \\
&= \sum_{n=1}^{\infty} \frac{1}{\sqrt{5}} (\gamma^n + (-1)^{n-1} \bar{\gamma}^n) z^n \\
&= \sum_{n=1}^{\infty} \frac{1}{\sqrt{5}} \left(\gamma^n - \left(\frac{-1}{\gamma} \right)^n \right) z^n. \tag{10.34}
\end{aligned}$$

This tells us that the n th **Fibonacci number** is, quite miraculously, given by $a_n = (\gamma^n - (-1/\gamma)^n)/\sqrt{5}$. Try it out for a few small n .

Puzzle 101. For each of the following cases, express the sequence equation (10.30) as a recurrence relation and find the solution as in the previous example:

1. $d = z$ and $c = \hat{r}$ for a real number r .
2. $d = 2z$ and $c = \hat{1}$.
3. $d = (1^*1^*)z$ and $c = \hat{1}$

Express the following recurrence relations and boundary conditions in the form of equation (10.30) and solve it:

1. Tower of Hanoi: $a_0 = 0$, $a_n = 2a_{n-1} + 1$ for $n > 0$.
2. **Triangular Numbers**: $a_0 = 0$, $a_n = a_{n-1} + n$ for $n > 0$.
3. Cycle: $a_0 = 1$, $a_1 = 2$, $a_2 = 5$, $a_n = a_{n-3}$ for $n > 2$.

In each case, give an expression in terms of z like equation (10.32); and if possible, find expressions for a_n for every n like equation (10.34).

Part of the “Sequences and Streams” series.

Example 10.23 A Calculus of Sums and Differences

The summation operator Σ in Table 10.5 computes the partial sums of a sequence including the empty sum that is 0, e.g., $\langle a_0, a_1, a_2, \dots \rangle$ maps to $\langle 0, a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots \rangle$. Its relation with the difference operator Δ , which maps $\langle a_0, a_1, a_2, \dots \rangle$ to $\langle a_1 - a_0, a_2 - a_1, a_3 - a_2, \dots \rangle$, is analogous to the relationship between integral and derivative:

$$\Delta \Sigma a = a \tag{10.35}$$

$$\Sigma \Delta a = a - a_0^*. \tag{10.36}$$

Part of the “Sequences and Streams” series.

The first is analogous to $\left(\int_{t=-\infty}^x f(t) dt\right)' = f(x)$ and the second to the Fundamental Theorem of Calculus $\int_{t=0}^x f'(t) dt = f(x) - f(0)$. Taken together, these equations give us tools for computing sums, and more. In this puzzle, we look at one application and in the next, see try out a more general technique.

The analogy with calculus is deeper still. Defining the *falling factorial powers* by $n^{\underline{k}} = n(n-1)\cdots(n+k-1)$ for $k \in [1..)$, we have

$$\begin{aligned}\Delta(\blacksquare) &= 1^* \\ \Delta(2^{\blacksquare}) &= (2^{\blacksquare}) \\ \Delta\left(\blacksquare^{\underline{k}}\right) &= k\left(\blacksquare^{\underline{k-1}}\right),\end{aligned}$$

where $(\blacksquare)$ is the sequence $\langle 0, 1, 2, \dots \rangle$, $(2^{\blacksquare})$ is the sequence $\langle 1, 2, 2^2, 2^3, \dots \rangle$, and $(\blacksquare^{\underline{k}})$ is the sequence $\langle 0, 1^{\underline{k}}, 2^{\underline{k}}, \dots, n^{\underline{k}}, \dots \rangle$. These three equations are the analogues of the calculus facts that $D \text{id} = 1$, $D(e^{\blacksquare}) = (e^{\blacksquare})$, and $D(\blacksquare^{\underline{k}}) = k(\blacksquare^{\underline{k-1}})$.

We can iterate the difference operator by composing it with itself (see Puzzle 109); this gives us the n th-difference operator Δ^n , where $\Delta^n a = \Delta(\Delta^{n-1} a)$. We write $\Delta^n a(k)$ for $(\Delta^n a)(k)$. Δ^n is the analogue of the n th derivative operator in calculus. Writing $\Delta = \text{tail} - \text{id}$ and so $\text{tail} = \text{id} + \Delta$, we have

$$\Delta^n = (\text{tail} - \text{id})^n = \sum_{k=0}^n \binom{n}{k} (-1)^{n-k} \text{tail}^k \quad (10.37)$$

$$\text{tail}^n = (\text{id} + \Delta)^n = \sum_{k=0}^n \binom{n}{k} \Delta^k, \quad (10.38)$$

by the Binomial Theorem (see Example 17.30) and because composition with the identity does nothing. For instance by (10.37), $\Delta^2 a = a'' - 2a' + a$.

Equation (10.38) is the discrete analogue of the Taylor Series in Calculus. Recall that $\binom{n}{k} = 0$ if $k > n$ and that $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!} \equiv \frac{n^{\underline{k}}}{k!}$. So, we can drop the upper limit on the sum and rewrite equation (10.38) as

$$\text{tail}^n = \sum_{k=0}^{\infty} \frac{n^{\underline{k}}}{k!} \Delta^k.$$

Applying both sides to a sequence a gives

$$a(n) = \sum_{k=0}^{\infty} \frac{\Delta^k a(0)}{k!} n^{\underline{k}}, \quad (10.39)$$

We anticipate the anonymous function short-hand of Chapter 11 where $\blacksquare$ is filled in with the argument of the function and e.g., $(2^{\blacksquare})$ represents the function $f(x) = 2^x$.

with the analogous Taylor series being

$$f(x) = \sum_{k=0}^{\infty} \frac{D^k f(0)}{k!} x^k.$$

We thus have the tools for a *discrete calculus* on sequences.

Puzzle 102. What comes next?

Using the ideas from Example 10.23, we can tackle a popular type of mathematical “brain teaser.” Consider the sequence a that begins

$$\langle 1, 2, 4, 8, 16, 31, 57, 99, 163, ??, \dots \rangle.$$

What comes next?

You are probably thinking that *anything* could come next, and you are right. But assuming that there is some pattern to these numbers, some regularity that we can exploit, can we find a sensible next value? We will make a specific assumption about that pattern: that for *some* $n \in [1..)$, which we do not know *a priori*, $\Delta^n a = 0^*$. (What would the calculus analogue of this assumption mean?)

With this assumption, use the “Discrete Taylor Series” equations (10.38) or (10.39) to fill next term in the sequence. (And with that solution you could fill in any number of terms.)

Hint: Because equation (10.39) involves k th differences of the sequence, it might help to compute a table of differences for the part of the sequence a that we have. It will look like this:

1	2	4	8	16	31	...
1	2	4	8	15	...	
	1	2	4	7	...	
		⋮	⋮	⋮	⋮	

What does our assumption imply about this table and about a ?

Part of the “Sequences and Streams” series.

Puzzle 103. Finding Sums with Differences.

Following up on the previous puzzle, we can use equation (10.36) to compute sums. For a simple example, suppose we want to compute $\sum_{k=0}^n 1$ for $n \in [0..)$.

Part of the “Sequences and Streams” series.

In sequence terms, we compute all these sums at once with

$$\Sigma 1^*,$$

but we already saw that $1^* = \Delta(\blacksquare)$, so by equation (10.36), this sequence is just $\Sigma\Delta(\blacksquare) = (\blacksquare) - 0 \cdot 1^*$ giving $(\Sigma\Delta(\blacksquare))(n) = n$, i.e., $\sum_{k=0}^n 1 = n + 1$. (Remember that Σ prepends a 0 for the empty partial sum.)

Use this technique to find $\sum_{k=0}^n 2^k$ and $\sum_{k=0}^n k^3$ for every n . For the latter, you might want to express k^3 in terms of falling factorial powers first.

Answers to Selected Puzzles.

Puzzle 91.

(i)

```
def id(x):  
    return x
```

(ii) We write a function that takes a constant c and returns the function const_c :

```
def const(c):  
    def const_c(x):  
        return c  
  
    return const_c
```

Then if $g = \text{const}(c)$, $g(x)$ equals c for every x . Alternatively, we could write a function that takes c as an exogenous parameter; the next item shows this approach.

(iii) We write a function that takes a and b as exogenous parameters:

```
def ell(x, a=0, b=1):  
    return a + b * x
```

Alternatively, like the previous item, we could write a function that takes a and b and returns the function $\ell_{a,b}$.

(v)

```
def f(a_b):  
    return a_b[0]
```



```

def s(a_b):
    return a_b[1]

(viii) def m(k):
        return k % 5

(x) def sigma(f):
     def shifted_f(x):
         return f(x - 1)

     return shifted_f

(xii) def indicator0_1(x):
       if x >= 0 and x <= 1:
           return 1
       return 0

```

Puzzle 92. A d -dimensional tuple of numbers v can be viewed as a function $v: [1..d] \rightarrow \mathbb{R}$, where $v(k)$ is the k th component of the tuple.

Puzzle 95.

1. `map` has type $(\mathbb{R} \rightarrow \mathbb{R}) \times \text{Seq}(\mathbb{R}) \rightarrow \text{Seq}(\mathbb{R})$ and is defined by

$$\text{map}(f, a)(n) = f(a(n)).$$

Note that when we write `map( $f, a$ )( $n$ )`, we first obtain the function $g = \text{map}(f, a)$ and then evaluate it at n , $g(n)$.

2. `zip` has type $(\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}) \times \text{Seq}(\mathbb{R}) \times \text{Seq}(\mathbb{R}) \rightarrow \text{Seq}(\mathbb{R})$ and is defined by

$$\text{zip}(g, a, b)(n) = g(a(n), b(n)).$$

3. `convolve` has type $\text{Seq}(\mathbb{R}) \times \text{Seq}(\mathbb{R}) \rightarrow \text{Seq}(\mathbb{R})$ and is defined by

$$\text{convolve}(a, b)(n) = \sum_{k=0}^n a(k)b(n-k).$$

4. `interleave` has type $\text{Seq}(\mathbb{R}) \times \text{Seq}(\mathbb{R}) \rightarrow \text{Seq}(\mathbb{R})$ and is defined by

$$\begin{aligned}\text{interleave}(a, b)(2n) &= a(n) \\ \text{interleave}(a, b)(2n + 1) &= b(n).\end{aligned}$$

Elements at even index in the interleaved sequence come from a and elements at odd index come from b .

5. `iterate` has type $(\mathcal{X} \rightarrow \mathcal{X}) \times \mathcal{X} \rightarrow \text{Seq}(\mathbb{R})$ and can be defined inductively by

$$\begin{aligned}\text{iterate}(f, x)(0) &= x \\ \text{iterate}(f, x)(n) &= f(\text{iterate}(f, x)(n - 1)) \text{ for } n \in [1..).\end{aligned}$$

Puzzle 96. The equation for f tells us that $f_0 = 0$, $f_1 = 1$, and for each $n \geq 2$, $f_n = f_{n-1} + f_{n-2}$. To see this, list out the values of both sides for the first several indices. In particular, the `add(tail(f), f)` part starts two indices back from the left-hand side.

Again, decomposing term by term, the equation for c tells us that for $n \in \mathcal{N}$, $c(2n) = 0$ and $c(2n + 1) = c(n) + 1$. Looking at the first few values, we have 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0. So, $c(2^k - 1) = c(2(2^{k-1} - 1) + 1) = c(2^{k-1} - 1) + 1 = k$ for $k \geq 0$ and $c(2^j + 1) = 1$ for $j \geq 1$. In general, $c(n)$ counts the number of consecutive 1's (from least to most significant bits) in the binary representation of n .

Puzzle 100. We will show the solution to the first few parts of the puzzle.

1. $(1 - z)1^* = 1^* - z1^* = 1^* - 0 :: 1^*$, which has first term $1 - 0 = 1$ and remaining terms $1 - 1 = 0$. Hence, $(1 - z)1^* = \widehat{1}$.
2. Because $(1 - z)_0 \neq 0$ it has a unique reciprocal defined by the equality in the previous item, $(1 - z)1^* = \widehat{1}$. Thus, $(1 - z)^{-1} = 1^*$.

3. We have

$$\begin{aligned}
 (1 - rz) \sum_{n=0}^{\infty} r^n z^n &= \sum_{n=0}^{\infty} r^n z^n - rz \sum_{n=0}^{\infty} r^n z^n \\
 &= \sum_{n=0}^{\infty} r^n z^n - \sum_{n=1}^{\infty} r^n z^n \\
 &= \hat{1},
 \end{aligned}$$

and because $(1 - rz)_0 \neq 0$, this defines its unique reciprocal.

4. Using equation (10.24) and the fact from above that $(1 - z)^{-1} = 1^*$,

$$\begin{aligned}
 \frac{d}{dz}(1 - z)^{-1} &= \frac{d}{dz} 1^* \\
 &= \sum_{n=1}^{\infty} n \cdot 1 z^{n-1} \\
 &= \sum_{n=0}^{\infty} (n + 1) z^n \\
 &= \sum_{n=0}^{\infty} \left(\sum_{k=0}^n 1 \right) z^n \\
 &= 1^* 1^* = (1 - z)^{-2}.
 \end{aligned}$$

Puzzle 101. We will show the solution to the first part of the puzzle.

1. Because $(za)_0 = 0$ and $(za)_n = a(n - 1)$ for $n \in [1..)$, the recurrence relation is

$$\begin{aligned}
 a(0) &= 0 + r \\
 a(n) &= a(n - 1) + 0.
 \end{aligned}$$

This give $a = r^*$. Using equation (10.31), we get the same solution

$$a = \hat{r} \sum_{n=0}^{\infty} z^n = \hat{r} 1^* = r^*,$$

because z^n has a 1 in position n and 0 elsewhere.

2. With $d = 2z$ and $c = \hat{1}$, the recurrence relation is

$$\begin{aligned} a(0) &= 0 + 1 \\ a(n) &= 2a(n-1) + 0, \end{aligned}$$

giving $a(n) = 2^n$. Again, equation (10.31) yields the same result:

$$a = \hat{1} \sum_{n=0} 2^n z^n = \sum_{n=0} 2^n z^n.$$

3. First, $1*1^*$ is the sequence b with $b(n) = \sum_{k=0}^n 1 \cdot 1 = n + 1$. So, $d = bz$ satisfies $d(n) = \sum_{k=0}^n b(k)z(n-k) = b(n-1) = n$.

By $a = da + c$, we have the recurrence relation

$$\begin{aligned} a(0) &= 1 \\ a(n) &= \sum_{k=1}^n ka(n-k), \text{ for } n \in [1..). \end{aligned}$$

Plugging in small values of n , we see

$$\begin{aligned} a(1) &= 1 \\ a(2) &= a(1) + 2a(0) = 3 \\ a(3) &= a(2) + 2a(1) + 3a(0) = 3 + 2 + 3 = 8 \\ a(4) &= a(3) + 2a(2) + 3a(1) + 4a(0) = 8 + 6 + 3 + 4 = 21 \\ a(5) &= a(4) + 2a(3) + 3a(2) + 4a(1) + 5a(0) = 21 + 16 + 9 + 4 + 5 = 55 \\ a(6) &= 55 + 42 + 24 + 12 + 5 + 6 = 144. \end{aligned}$$

These are all [Fibonacci numbers](#) with even index: $a_0 = 1$ and $a_n = f_{2n}$ for $n \in [1..)$, where f is the Fibonacci sequence. This means that $f_{2n+1} = a_n - a_{n-1}$ as well.

To confirm this, we need to establish that for $n \in [3..)$, $a_n = a_{n-1} + a_n - a_{n-1} = a_{n-1} + (a_{n-1} + a_{n-1} - a_{n-2}) = 3a_{n-1} - a_{n-2}$, which can be done (if desired) with a slightly tedious bit of algebra.

Anonymous Functions

11

Chapter

So far in this Interlude alone, we have defined – and named – many, many different functions, and the various f 's, g 's, h 's, and other names are strewn around the grounds like the detritus after an outdoor rock concert. We name functions for the same reason we name other variables, to use and refer to them without repeating their definition. Most often, we want our functions to have names, whether those names are used globally or are local to a particular problem or calculation. Sometimes, however, naming a function is unnecessary, even inconvenient. An **anonymous function** defines a function without giving it a name.

This section describes two notations for defining anonymous functions and gives a variety of examples and use cases. Why *two* notations, you might wonder? Put simply, we have a basic notation that works in all cases, and a short hand that is especially pleasant in a very common case. There is little *conceptually* new here, but it is worth reviewing this to get familiar with the notation, as we will use the anonymous functions throughout.

The basic notation has the form $\langle \text{parameters} \dots \rangle \mapsto \text{return value}$. We specify the list of parameter names in the initial tuple and these are used as local values in the expression for the return value. The symbol $\mapsto$ is read “maps to.” For example, we write $\langle x \rangle \mapsto x^2$ for the function $f(x) = x^2$, $\langle t \rangle \mapsto 2^t - 1$ for the function $g(t) = 2^t - 1$, and $\langle s \rangle \mapsto s \log(s) - s$ for the function $h(s) = s \log(s) - s$. Note that $\langle x \rangle \mapsto x^2$ and $\langle u \rangle \mapsto u^2$ represent the *same* function because the parameter is a local variable only. Multiple input parameters are allowed, e.g., $\langle x, y \rangle \mapsto x^2 + y^2$. The arguments can be of any type (e.g., numbers, vectors), which will typically be clear from context.

The shorthand notation applies in the common but special case where the function takes only a single parameter, and we do not name the parameter, just show a “hole” that is to be filled with the argument value. This hole can be repeated multiple times in the return-value expression, all of which represent the same value. The first three functions above are denoted in this way by $(\blacksquare^2)$, $(2^\blacksquare - 1)$, and $(\blacksquare \log(\blacksquare) - \blacksquare)$. Anonymous functions in the shorthand notation are *always* surrounded by parentheses

to delimit the scope of the hole, *except* when the function is in the argument list of another function (e.g., $D(\blacksquare^2)$ or $G(\log(\blacksquare), \blacksquare + 1)$).¹ In the text, we use $\blacksquare$ for the hole. When writing an anonymous function by hand, use any consistent mark; I suggest either a dash ($-^2 + 1$) or an underscore ($\_{}^2 + 1$).

¹In this case, the scope of the hole is the extent of that argument. This avoids annoying redundant parentheses.

Summary of Anonymous Function Notation.

- Basic notation $\langle \text{parameters} \dots \rangle \mapsto \text{return value}$.

Multiple parameters are allowed; they are local variables that are in scope only in the return value expression. The functions $\langle x \rangle \mapsto f(x)$, $\langle a, b \rangle \mapsto g(a, b)$, $\langle u, v, w \rangle \mapsto h(u, v, w)$, $\dots$ are, respectively, equal to $f, g, h, \dots$

Examples:

- $\langle t \rangle \mapsto \langle \cos(2\pi t), \sin(2\pi t) \rangle$
- $\langle s \rangle \mapsto \frac{2^s}{s!} e^{-2}$
- $\langle r, \theta, \lambda \rangle \mapsto \langle r \cos(\theta) \cos(\lambda), r \sin(\theta) \cos(\lambda), r \sin(\lambda) \rangle$

- Short-hand notation (return value with hole $\blacksquare$).

Used only for functions that accept one argument. Always wrapped in parentheses to delimit scope, except for an anonymous function in the argument list of another function. The function $(f(\blacksquare))$ equals f .

Examples:

- $\left(\langle \cos(2\pi \blacksquare), \sin(2\pi \blacksquare) \rangle \right)$
- $\left(\frac{2^{\blacksquare}}{\blacksquare!} e^{-2} \right)$
- $\left(\begin{bmatrix} 1 & \blacksquare \\ 0 & 2 \end{bmatrix} \right)$

Anonymous functions are available in many mainstream programming languages. For instance in Python, the `lambda` form defines an anonymous function. The examples in Figure 11.1 demonstrate our notation alongside their definition as Python `lambda`'s and corresponding named functions.

Puzzle 104. Describe each of the following functions, either in words, in code, or by defining named mathematical functions:

1. $\langle a, b \rangle \mapsto |a - b|$
2. $(\blacksquare(\blacksquare - 1)(\blacksquare + 1))$
3. $\langle t \rangle \mapsto \langle \cos(t), \sin(t), t \rangle$
4. $\left(\frac{(\blacksquare + 2)!}{\blacksquare!} \right)$
5. $\langle \ell, m, n \rangle \mapsto \min(\ell, m, n)$

$\langle x \rangle \mapsto 2x + 1$ or $(2\blacksquare + 1)$ for short	$\langle x \rangle \mapsto e^{-2x^2}$ or $(e^{-2\blacksquare^2})$ for short
<code>lambda x: 2 * x + 1</code>	<code>lambda x: math.exp(-2 * x * x)</code>
<code>def no_name(x):</code> <code>return 2 * x + 1</code>	<code>def no_name(x):</code> <code>return math.exp(-2 * x * x)</code>
$\langle x, y \rangle \mapsto x + y - xy$	$\langle k \rangle \mapsto \binom{10}{k} 2^{-k}$ or $(\binom{10}{\blacksquare} 2^{-\blacksquare})$
<code>lambda x, y: x + y - x * y</code>	<code>lambda k: scipy.special.binom(10,k) * (2 ** (-k))</code>
<code>def no_name(x, y):</code> <code>return x + y - x * y</code>	<code>def no_name(k):</code> <code>return scipy.special.binom(10,k) * (2 ** (-k))</code>
$\langle u, v, w \rangle \mapsto \sqrt{u^2 + v^2 + w^2}$	$\langle v \rangle \mapsto v ^2$
<code>lambda a, b, c: math.sqrt(a*a + b*b + c*c)</code>	<code>lambda v: sum(vi * vi for vi in v)</code>
<code>def no_name(x, y, z):</code> <code>return math.sqrt(x*x + y*y + z*z)</code>	<code>def no_name(v):</code> <code>norm = 0</code> <code>for vi in v:</code> <code>norm += vi ** 2</code> <code>return norm</code>

FIGURE 11.1. Various functions defined with anonymous function notation, as Python lambda's, and as Python named functions.

Indicator Functions

12

Chapter

In computer programming, the humble **if-then-else** plays a vital role because it lets us make choices that are contingent on the input data.

```
def a_procedure(object):  
    if has_some_property(object):  
        do_something(object)  
    else:  
        do_other_thing(object)  
    return object
```

In many languages, the **if-then-else** form can be used in an *expression*:

- Python: `x + 1 if even(x) else x`
- JavaScript/C/C++: `even(x) ? (x + 1) : x`
- Haskell: `if even x then x + 1 else x`
- Rust: `if even(x) { x + 1 } else { x }`
- Clojure: `(if (even x) (+ x 1) x)`

This ability to create contingent expressions is useful and efficient in mathematical work too. We use *indicator* functions as a mathematical **if-then-else**. This section describes indicator functions and introduces two notations for them that we will use extensively, a basic notation and a short-hand convenient for common cases.

An example application for an **if-then-else** expression is to simplify the definition of functions that act very differently on various parts of their domains. Consider the functions $f, g, h: \mathbb{R} \rightarrow \mathbb{R}$ given by

$$f(x) = \begin{cases} 1 & \text{if } x > 2 \\ 2 & \text{if } x > 1 \\ 4 & \text{if } x > -1 \\ 0 & \text{otherwise,} \end{cases} \quad g(z) = \begin{cases} z + 4 & \text{if } z \leq -4 \\ \left(\frac{z+4}{4}\right)^2 & \text{if } -4 < z < 4 \\ z & \text{otherwise.} \end{cases} \quad h(k) = \begin{cases} \frac{2^k}{(k-2)!} & \text{if } k \in [2..) \\ 0 & \text{otherwise.} \end{cases}$$

While the “big brace” case-by-case definitions are clear enough, they are rather bulky, and they become inconvenient when we want to operate on f or g . Defining $r(x, y) = f(x)f(y)$, for instance, would proliferate the cases into an even larger “big brace.” The definition of h also gets across the idea, but it requires verbal qualifications that are *separate* from the defining equation, leaking local variables and our attention. Sometimes such steps are needed, but usually indicators can make such definitions nicer. With indicators, we would write instead:

$$\begin{aligned} f(x) &= 4 \{-1 < x \leq 1\} + 2 \{1 < x \leq 2\} + \{x > 2\}, \quad \text{or} \\ f &= 4 (-1\_1] + 2 (1\_2] + (2\_), \\ g(z) &= (z + 4) \{z \leq -4\} + \left(\frac{z + 4}{4}\right)^2 \{-4 < z < 4\} + z \{z \geq 4\} \quad \text{or} \\ g &= (\blacksquare + 4) (\_ -4] + \left(\frac{\blacksquare + 4}{4}\right)^2 (-4\_4) + (\blacksquare) (\_4], \\ h(u) &= \frac{2^u}{(u - 2)!} \mathbb{N}(u) \{u \geq 2\}, \quad \text{or} \\ h &= \frac{2^\blacksquare}{(\blacksquare - 2)!} [2..\_), \end{aligned}$$

solving both problems. We will describe the notation below, though the meaning of these expressions is likely already clear. For instance, f returns 4 for inputs in $(-1\_1]$, 2 for inputs in $(1\_2]$, 1 for inputs > 2 , and 0 otherwise.

If $\mathcal{R}$ is a set, Example 10.12 showed¹ that every subset $\mathcal{A} \subseteq \mathcal{R}$ corresponds to a unique function² $\mathcal{R} \rightarrow \mathbb{2}$, called the **indicator** of $\mathcal{A}$, that *returns 1 if its input belongs to $\mathcal{A}$ and 0 if not*.

¹See also Example 14.4.

²Recall that $\mathbb{2} = \{0, 1\}$.

Written as a Python function, the indicator of a set V would look like

```
def indicator_of_V(v):
    if v in V:
        return 1
    return 0
```

showing how indicator functions acts as a mathematical if-then-else. Most often, but not always, our indicators will have some $\mathbb{R}^n$ as their domain, and usually just $\mathbb{R}$.

Given a set $\mathcal{A}$, we need a name for its indicator, and in light of the aforementioned correspondence, it is natural to use the *set itself* as the function. This is our basic indicator notation, exemplified below.³

³A Python programmer might think of this as attaching a `__call__` method to objects of type `set`.

If $\mathcal{V}$ is a set, the **indicator of $\mathcal{V}$** is the *function* denoted by the set itself $\mathcal{V}$ that returns 1 (true) if its input argument lies in $\mathcal{V}$ and 0 (false) if it does not. That is,

$$\mathcal{V}(v) = \begin{cases} 1 & \text{if } v \in \mathcal{V} \\ 0 & \text{otherwise.} \end{cases} \quad (12.1)$$

In practice, the domain of this function is usually clear from context, but ambiguity here does no real harm, as the function returns 0 for any input outside $\mathcal{V}$.

Any function f that returns only 0 or 1 is the indicator of some set; the set in question is just $f^{-1}(\{1\})$. So we call any function with codomain $\mathbb{2}$ an *indicator*. Because they return 0 or 1, *Indicators convert logical statements about their sets into arithmetic*. The indicator $\mathcal{U} \cdot \mathcal{V}$ returns 1 when given a value in *both* of $\mathcal{U}$ and $\mathcal{V}$, so it is the indicator of the intersection $\mathcal{U} \cap \mathcal{V}$. And if $\mathcal{U}$ and $\mathcal{V}$ are disjoint, then $\mathcal{U} + \mathcal{V}$ is also an indicator because an input value can belong to at most one of the two sets. It returns 1 when a value belongs to $\mathcal{U}$ or $\mathcal{V}$, and thus is the indicator of the union. More generally, for any family of sets $\mathcal{W}_i$ for i in an index set $\mathcal{I}$, the function $\sum_{i \in \mathcal{I}} \mathcal{W}_i$ counts the number of $\mathcal{W}_i$'s to which its input belongs. The indicator $1 - \mathcal{V}$ returns 1 when its inputs is *not* in $\mathcal{V}$ and 0 otherwise; it is thus the indicator of the complement of $\mathcal{V}$. More generally, for two sets $\mathcal{U}$ and $\mathcal{V}$, $(\mathcal{U} - \mathcal{V})^2$ is the indicator of the difference $\mathcal{U} - \mathcal{V}$; when $\mathcal{V} \subseteq \mathcal{U}$, this simplifies to $\mathcal{U} - \mathcal{V}$. Table 12.1 summarizes the relationships of this “Indicator Logic”.

Puzzle 105. If f is a function returning a number, then f^2 denotes the function $f^2(x) = (f(x))^2$. If f is an indicator, what can you say about f^2 ?

The basic notation works for any specification of sets. For example, as indicators: $\{-1, 0, 1\}$ returns 1 when its argument equals -1, 0, or 1 and otherwise returns 0; $[0\_1]$ returns 1 when its argument is a real number $0 \leq x \leq 1$ and otherwise returns 0; and $[0\_.)$ returns 1 when its argument is a natural number. We *evaluate* these as functions just as we do any other function, by passing the argument list in parentheses. This can take a moment to get used to, but it works. For instance, $\{-1, 0, 1\}(2) = 0$, $[0\_1](1/2) = 1$, and $\{0\}(0) = 1$. Taking it one step further, we can pass expressions as arguments to indicators:

- $\{0\}(x - y)$ equals 1 when $x = y$ or else equals 0;
- $\{0\}(x \bmod 2)$ equals 1 when x is an even integer or else equals 0; and
- $(-1\_1)(x - y)$ equals 1 when $|x - y| < 1$ or else equals 0.

Indicator	Logical Operation	Set Operation	Constraints/Notes
$\mathcal{U} \cdot \mathcal{V}$	and ($\wedge$)	$\mathcal{U} \cap \mathcal{V}$	none
$\mathcal{U} + \mathcal{V}$	or ($\vee$)	$\mathcal{U} \cup \mathcal{V}$	$\mathcal{U} \cap \mathcal{V} = \{\}$
$\mathcal{U} + \mathcal{V} - \mathcal{U} \cdot \mathcal{V}$	or ($\vee$)	$\mathcal{U} \cup \mathcal{V}$	none
$1 - \mathcal{V}$	not (!)	$\mathcal{V}^c$	none
$\mathcal{U} - \mathcal{V}$	$u \wedge !v$	$\mathcal{U} - \mathcal{V}$	$\mathcal{V} \subseteq \mathcal{U}$
$\mathcal{U}(1 - \mathcal{V})$	$u \wedge !v$	$\mathcal{U} - \mathcal{V}$	none
$\langle u, v \rangle \mapsto \mathcal{U}(u) \cdot \mathcal{V}(v)$	pairing	$\mathcal{U} \times \mathcal{V}$	none, denoted $\mathcal{U} \otimes \mathcal{V}$, cf. Chapter 15
$\sum_{i \in \mathcal{I}} \mathcal{W}_i$	count		counts number of $\mathcal{W}_i$'s that contain the input value

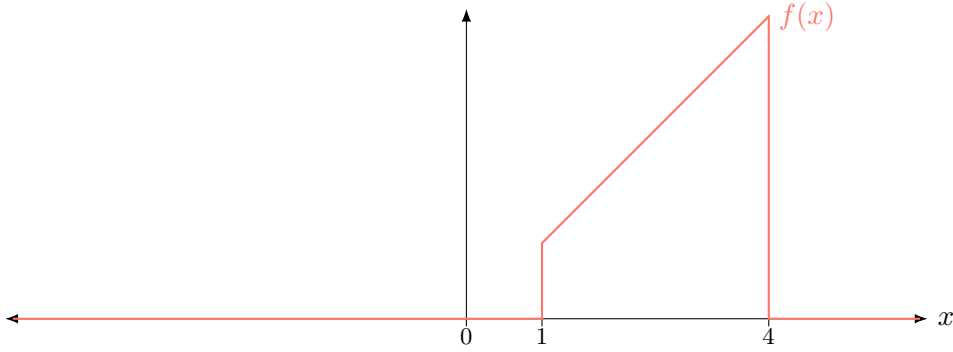
TABLE 12.1. Indicator Logic. The correspondence between indicator arithmetic and logical operations on the underlying sets. Here, $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}_i$ are sets for all i in an index set $\mathcal{I}$.

We can also operate on indicators as we would any other functions; for example, the linear combination $3\{0\} + 7\{1\} - 10\{2\}$ returns 3 when given 0, 7 when given 1, -10 when given 2, and otherwise 0. In code, this would be expressed as

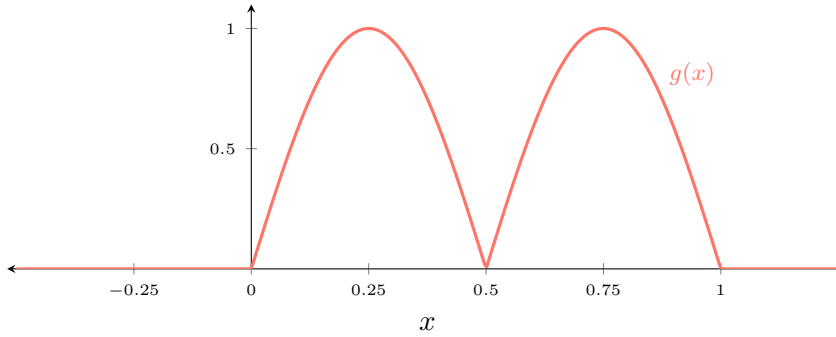
```
def f(x):
    if x == 0:
        return 4
    elif x == 1:
        return 7
    elif x == 2:
        return -10
    else:
        return 0
```

(Evaluating this function at x looks like $3\{0\}(x) + 7\{1\}(x) - 10\{2\}(x)$.) Similarly, the function on $\mathbb{R}^{100}$ defined by $\langle x \rangle \mapsto \{0\}(x_1) + \{0\}(x_2) + \cdots + \{0\}(x_{100})$ *counts* the number of components in its argument $\langle x_1, x_2, \dots, x_{100} \rangle$ that are equal to zero.

Writing functions with indicators is typically unambiguous, but when a juxtaposition is less than clear, we can insert an explicit operator to make it clear. For example, the function $f: \mathbb{R} \rightarrow \mathbb{R}$ given by $f = \text{id} \cdot [1\_4]$ looks like



where we use the $\cdot$ to indicate the product, $f(x) = \text{id}(x) \cdot [1\_4](x)$. This works well too with our anonymous function notation. For instance, $g = (\sin(2\pi \blacksquare)) \cdot ([0\_ \frac{1}{2}] - [\frac{1}{2} \_ 1])$ looks like



Puzzle 106. Describe the following functions of x or of x and y :

1. $\text{id} \cdot [0\_] - \text{id} \cdot (\_ 0)$
2. $\frac{1}{8} [1 \dots 8]$
3. $[0\_] - [1\_ 2] - [3\_ 4] - [7\_ 8]$
4. $(\blacksquare^2) \cdot [-2\_ 2]$
5. $\langle x, y \rangle \mapsto (x + y) [1 \dots 8](x) [1 \dots 8](y)$
6. $\langle x, y \rangle \mapsto [0\_ r](\sqrt{x^2 + y^2})$ for some $r > 0$

In cases where a set has a complicated expression, it often helps to express the indicator as a combination of indicators of simpler pieces using Table 12.1. For example, if we want the indicator of tuples $\langle u, v \rangle$ in the set $[0 \dots 4] \times [1 \dots 3]$, we can write the basic notation directly, as $[0 \dots 4] \times [1 \dots 3](u, v)$, but it is clearer and easier to decompose the product as $[0 \dots 4](u) [1 \dots 3](v)$. Similarly, the indicator of $(\_ - 1] \cup [1\_)$ can be written as the sum of the pieces, $(\_ - 1] + [1\_)$, because $(\_ - 1]$ and $[1\_)$ are disjoint.

While the basic notation is typically declarative and unambiguous, in many common cases – such as defining functions by their inputs – we can write our indicators even more clearly and efficiently with a *shorthand notation*, using what we call **Iverson braces**. We write a Boolean expression in $\{\}$'s, and the result is 1 when that expression is true, 0 when it is false. So, $\{y > 3\}$ is 1 when y is bigger than 3, else 0, and $\{a^2 + b^2 \leq 4\}$ is 1 when $a^2 + b^2$ is at most 4, else 0.

The idea hearkens back to set-builder notation (equation 9.1) where we specify a set as the points of some type that satisfy a predicate. In Iverson braces, we *evaluate* an indicator by writing the predicate in braces with the input argument in the predicate expression. This will be easier to see with some examples:

- The function $h(x, y) = \{x = y\}$ returns 1 when its arguments are equal, else 0. In basic notation, we would write this as $h(x, y) = \{0\}(x - y)$ or as $h = \{\langle x, y \rangle \mid x = y\}$, which is a bit harder to parse.
- For a constant a , $\langle x \rangle \mapsto \{x = a\}$ is the indicator of $\{a\}$ in the shorthand notation, with $\{\blacksquare = a\}$ even shorter. This is just $\{a\}$ in basic notation, which is $\{a\}(y)$ when evaluated at some y .
- In basic notation, $3\{0\} + 7\{1\} - 10\{2\}$ is a combination of indicators. Evaluating this function at x looks like $3\{0\}(x) + 7\{1\}(x) - 10\{2\}(x)$, which is trickier to parse. In shorthand notation, we express this value more clearly as

$$3\{x = 3\} + 7\{x = 1\} - 10\{x = 2\},$$

Note that the Iverson braces in this expression are *actual values* in terms of a specific value x . For an anonymous function, just replace x with a hole $\blacksquare$.

- The indicator $(\_ - 1] + [1 \_)$ is 1 for numbers with absolute value at least 1. Evaluating it is nicely expressed with Iverson:

$$\{-1 \leq t \wedge t \geq 1\} \equiv (\_ - 1](t) + [1 \_)(t).$$

We can use Iverson braces to represent an anonymous function by using a $\blacksquare$ in place of t .

In general, for an indicator in basic notation of the form $\{u \mid p(u)\}$, for some predicate p , the Iverson braces are defined by $\{p(x)\}$ which equals $\{u \mid p(u)\}(x)$. Note again that this is the indicator *evaluated* at a given argument. Because the Iverson braces give the evaluated function, we can get expressions may look surprising at first. For example, the evaluating the indicator $\{0\}$ at various values we get $\{0\}(x)$, $\{0\}(1)$, $\{0\}(0)$, which with Iverson braces look like $\{x = 0\}$, $\{1 = 0\}$, $\{0 = 0\}$. The last two of these are, respectively, just 0 and 1. Any false statement in Iverson

braces gives 0; any true statement gives 1. We can also use the hole notation for anonymous functions to represent the indicator function itself with Iverson braces. Then, $\{p(\blacksquare)\} \equiv \{u \mid p(u)\}$, which can sometimes be clearer and cleaner.

There seem to be a lot of choices here, but worry not. Our guiding principle is to write as simply and clearly as possible. The flexibility means that indicators will almost always allow us to express contingent cases clearly and concisely.

Summary of Indicator Notation

Basic Notation. We use the set itself to stand for the indicator function of that set. So, $\mathcal{V}(x)$ is 1 if $x \in \mathcal{V}$ else 0.

Examples: $\mathbb{N}$ is the indicator of the natural numbers, with $\mathbb{N}(u)$ giving 1 when $u \in [0..)$, else 0. $[0\_1]$ is the indicator of the unit interval, with $[0\_1](0.6) = 1$ and $[0\_1](-2.4) = 0$. $\{0, 1\}$ returns 1 when given 0 or 1, else 0. $[0..)(n/4)$ is 1 when n is a natural number that is divisible by 4, else 0.

Short-hand Notation (Iverson braces). We evaluate an indicator by writing a set's defining predicate in braces *evaluated at the input argument*. So, the indicator of $\{u \mid p(u)\}$ evaluated at argument x is written $\{p(x)\}$, which is 1 when $p(x)$ is true else 0. Here, x , unlike u , is not local to the scope of the braces.

We can use the hole notation with Iverson braces to represent the indicator function itself, e.g., $\{p(\blacksquare)\}$.

Examples: $\{x^2 + y^2 \leq 1\}$ is 1 when the point $\langle x, y \rangle$ belongs to the unit disk else 0. $\{\cos(t) > 1/\sqrt{2}\}$ when the angle t is within $\pi/4$ radians of East. $\{-1 \leq \blacksquare \leq 1\} \equiv [-1\_1]$. $\{1 = 0\}$ is 0 and $\{1 = 1\}$ is 1.

Like the **if-then-else** in programming, indicators have many uses, which we will see as we proceed. A common use, described at the beginning of this section, is to specify functions that treat disparate parts of their domains differently. Consider the equation above defining $h: \mathbb{R} \rightarrow \mathbb{R}$:

$$h = \left(\frac{2^{\blacksquare}}{(\blacksquare - 2)!} \right) [2..).$$

This tells us that h returns 0 for any input that is not in $[2..)$. The term multiplying the indicator is not well defined otherwise. When used like this in a defining equation, we give the indicators precedence in that we then as evaluated before any terms multiplying it; if the indicator returns 0, we ignore those terms and return 0. Indicators are also useful for combining different terms or functions with constraints using

the indicator logic of Table 12.1. We often use indicators for counts with the indicators giving the condition and for sums and integrals to select a region of summation/integration. We will also make extensive use of indicators for describing *events*, binary outcomes that can occur (or not) in observing a random process.

Example 12.1 Piecewise Constant Functions

A **piecewise-constant function** returns 0 except over a collection of disjoint intervals, on each of which it returns (potentially differing) constant values. Formally, a piecewise constant function can be written as

$$\sum_{k=0}^n a_k [u_k \_ v_k], \quad (12.2)$$

where the intervals in the sum can also be of the form $[u_k \_ v_k)$, $(u_k \_ v_k]$, or $(u_k \_ v_k)$ if desired. Here, n is either a natural number or ∞ , the a_k 's are constants, and where for all k , $u_k \leq v_k$ and $v_k < u_{k+1}$. This function is 0 except on one of the intervals $[u_k \_ v_k]$ over which it is constant.

As an example, the piecewise-constant function

$$p = 3[-4 \_ -3.5] - 2[-2 \_ -1] + [0.5 \_ 1.5] - 3[4 \_ 5]$$

has the graph shown in Figure 12.1.

Piecewise-constant functions have a variety of uses, including giving a simple approximation to other functions, histograms for data visualization, representing signals, and describing simple probability distributions. As an example, we can construct a piecewise-constant approximation to the function $\text{id}: \mathbb{R} \rightarrow \mathbb{R}$. We will define functions q_m for $m \in [1 \dots)$ satisfying $|q_m(x) - x| \leq 2^{-m}$ for all real x . Thus as m grows, piecewise-constant q_m gets closer and closer to id .

Equation (12.2) requires that we *can* write the function in that particular form, but there are often more convenient ways to index the pieces. We will define q_m for each integer by making a piecewise-constant varying around that

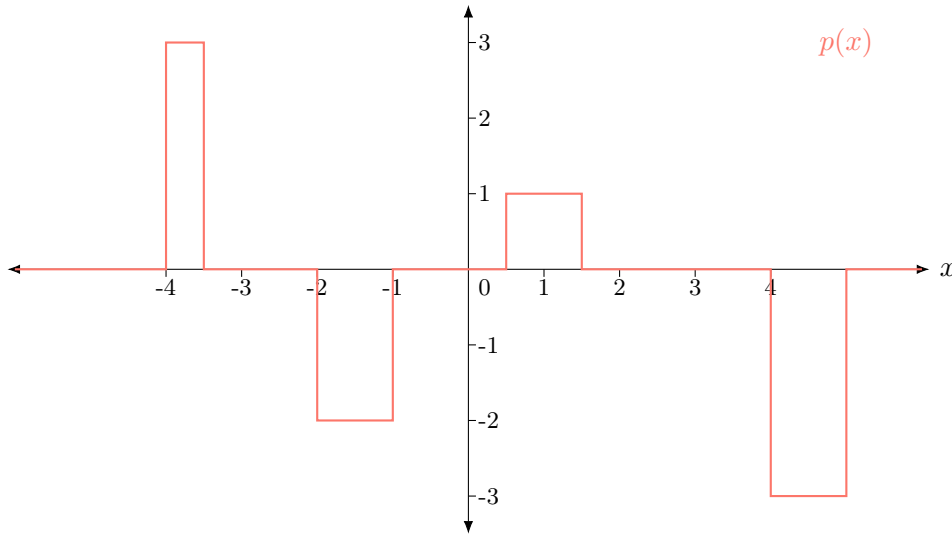


FIGURE 12.1. A piecewise constant function from Example 12.1.

integer with pieces of width 2^{-m} :

$$\begin{aligned}
 q_m = & \sum_{k=0}^{\infty} \sum_{j=0}^{2^m-1} (k + j2^{-m}) \cdot [k + j2^{-m} - k + (j+1)2^{-m}) \\
 & - \sum_{k=1}^{\infty} \sum_{j=0}^{2^m-1} (-k + j2^{-m}) \cdot [-k + j2^{-m} - -k + (j+1)2^{-m}). \quad (*)
 \end{aligned}$$

Each term in (*) inside the sums over k describes a sequence of steps of width 2^{-m} moving by increments of 2^{-m} from an integer $n = \pm k$ to $n + 1$. While the sum in (*) looks more complicated than equation (12.2), we can put it in that form by re-indexing.

Answers to Selected Puzzles.

Puzzle 105. The square of an indicator is the same indicator because $0^2 = 0$ and $1^2 = 1$: $f^2 = f$.

Puzzle 106.

1. This is just $|x|$.
2. A function that returns a weight $1/8$ for integers from 1 to 8 and 0 elsewhere.
3. The function is 1 for all non-negative real numbers except those between 1 and 2, 3 and 4, and 7 and 8, returning 0 elsewhere.

4. This is the parabola $\langle x \rangle \mapsto x^2$ between -2 and 2 and 0 outside that range.
5. On the 64 points $\langle x, y \rangle$ with integer components in $[1 \dots 8]$ returns the sum of the components.
6. This is 1 inside the disk of radius r centered on the origin, otherwise 0.

Composition

13

Chapter

Contents

13.1 The Operation of Composition	621
13.2 Commutative Diagrams	627
13.3 Inverse Functions	636
13.4 Describing Structure	647

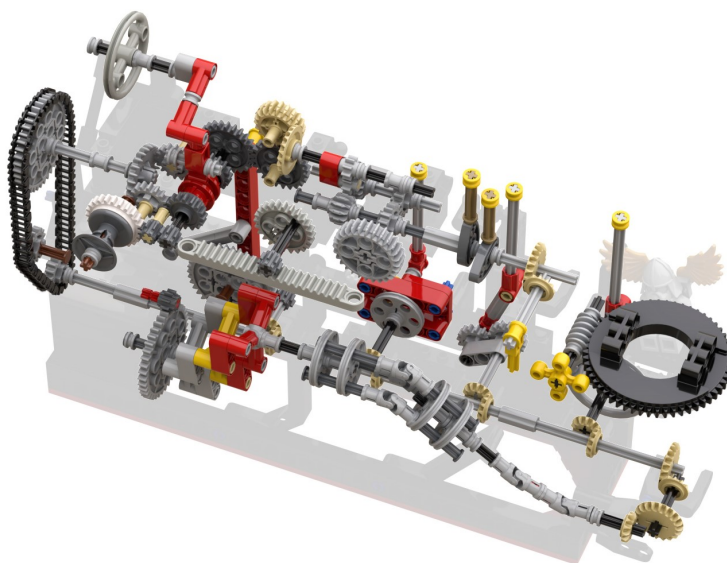


FIGURE 13.1. A wonderful Lego design that builds a complex machine by composing simpler moving pieces. Design and image credit: [Brick Experiment Channel](#).

“Compositionality” is the property that allows systems, actions, relationships, or operations to be combined into new systems, actions, relationships, or operations by connecting the output or result of one to the input or context of another. We build electrical circuits and machines (e.g., Figure 13.1) by joining simpler circuits and machines, output to input, in a larger structure. We build software with an architecture of modules connected by passing data and messages among them. When we navigate in an unfamiliar city by landmarks, we join familiar paths end to end to

get around: hotel to museum, museum to restaurant, restaurant to subway, subway to hotel, ... Making coffee combines actions in sequence, some of which must be done before others: heat water, steep coffee, add sugar, add cream, stir, drink. Cleaning our home for arriving guests combines some independent steps: clean the living room, clean the kitchen, fix up the guest room. Directing the hose on a firetruck turret pairs directional and azimuthal rotation. Analyzing data is often expressed as a pipeline of simpler analyses, from cleaning to summaries to visualizations, each stage receiving the output of the previous one. *Composition is the essence of modularity.*

Functions are the archetypal input-output system and are especially amenable to being combined by composition. Indeed, function composition is a fundamental operation for creating functions from other functions. The composition of two functions is a new function that we evaluate by first evaluating one of the functions with the given input and then passing its return value as input to the other to get the final return value. See Figure 13.2 for a simple example.

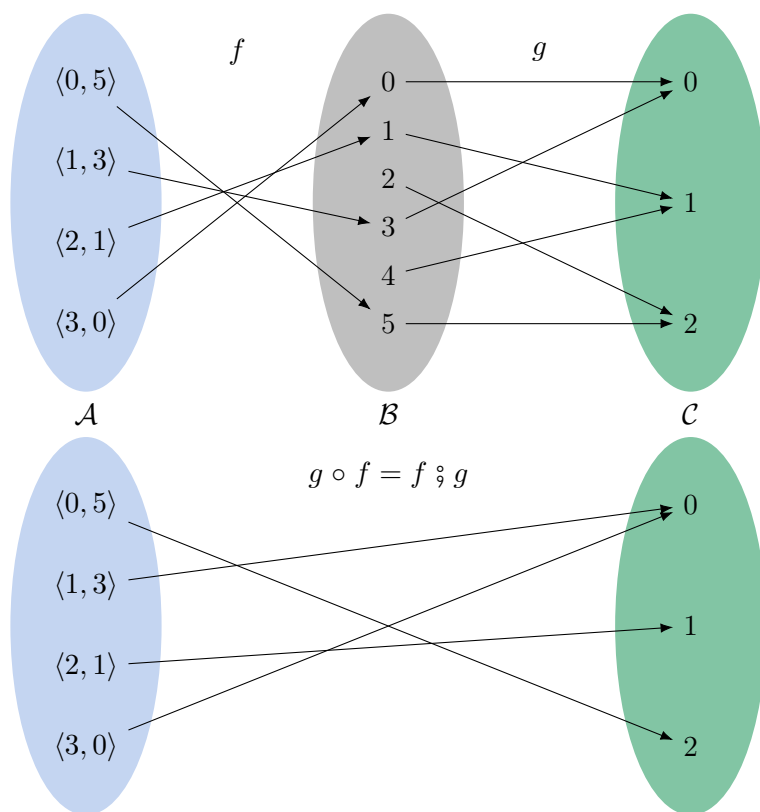


FIGURE 13.2. The composition of two simple functions where the output of f is input to g .

As another example, consider shuffling a deck of cards, which we do by repeatedly

rearranging/interleaving the cards. Each rearrangement – or *permutation* – of the cards is the evaluation of a function that takes a deck as input and returns a (rearranged) deck as output. If we do two permutations in sequence, the second function receives the rearranged deck output by the first. The combined function is itself a permutation of the deck. And so we can continue until the cards are suitably “randomized.”

13.1 The Operation of Composition

Composition is an operation we perform on two functions to create a new function, feeding the outputs of one as the inputs of the other. Only functions that are *compatible* can be composed, where the outputs of the one function must be valid inputs to the other.

There are two common notations for the composition, differing only in the order in which the two composed functions are listed. In the **syntactic order**, we write $g \circ f$ for the function that takes f ’s inputs and passes f ’s outputs to g ’s inputs. We read this as “ g after f ”. In the **pipeline (or diagrammatic) order**, we write $f \circledast g$ instead. Both notations can be useful and they can be used freely as preferred in any situation.

If $f: \mathcal{A} \rightarrow \mathcal{B}$ and $g: \mathcal{B} \rightarrow \mathcal{C}$, the **composition** of g with f is a function $\mathcal{A} \rightarrow \mathcal{C}$. Composition is well-defined when f and g are *compatible*, meaning that the codomain of f matches the domain of g .

The syntactic order $g \circ f$ is read “ g after f ” and is defined by

$$g \circ f: \mathcal{A} \rightarrow \mathcal{C} \quad (13.1)$$

$$(g \circ f)(x) = g(f(x)). \quad (13.2)$$

We evaluate $g \circ f$ at x by first evaluating f at x and then evaluating g at the value $f(x)$ returned by f . (Thus, g after f .)

The pipeline or diagrammatic order $f \circledast g$ is read “ f then g ” and is defined by

$$f \circledast g: \mathcal{A} \rightarrow \mathcal{C} \quad (13.3)$$

$$(f \circledast g)(x) = g(f(x)). \quad (13.4)$$

We first evaluate f at x then g at the returned value $f(x)$. (So, f then g .)

The syntactic order represents the way we *write* the evaluation of the functions, $g(f(x))$, with the second function listed first. The diagrammatic order represents the order of evaluation as we trace a path in a diagram like $\mathcal{A} \xrightarrow{f} \mathcal{B} \xrightarrow{g} \mathcal{C}$ or in a pipeline as we pass data through f then g in a computer program. (The semicolon evokes the separator between statements in a program.) The syntactic $g \circ f$ form (“ g after f ”) is more frequently seen and used. In some cases, such as with matrices/linear transformations and some operators, the syntactic order is used without the $\circ$ just juxtaposition, e.g., for matrices A and B of appropriate dimensions, AB is the composition “ A after B ” of the corresponding linear transformations.

Puzzle 107. If $f(x) = x + 1$, $g(x) = x^2$, and $h(x) = x - 1$:

- What is $g \circ f$?
- What is $f \circ g$?
- What is $f \circ h$? Does it equal $h \circ f$?
- What are $(f \circ g) \circ h$ and $f \circ (g \circ h)$?

In terms described in the last chapter (Section OPERATORS), composition is an operator that takes two functions and returns a new function, i.e., with type

$$\circ: (\mathcal{B} \rightarrow \mathcal{C}) \times (\mathcal{A} \rightarrow \mathcal{B}) \rightarrow (\mathcal{A} \rightarrow \mathcal{C}) \quad (13.5)$$

$$\circ: (\mathcal{A} \rightarrow \mathcal{B}) \times (\mathcal{B} \rightarrow \mathcal{C}) \rightarrow (\mathcal{A} \rightarrow \mathcal{C}). \quad (13.6)$$

Composition has two essential properties:

1. **Composition with id gives the original function:** $f \circ \text{id} = f = \text{id} \circ f$.

For any function f , $f(\text{id}(x)) = f(x)$ and $\text{id}(f(x)) = f(x)$, by definition of the identity function. (If f ’s domain and codomain are different, these are two different versions of the identity function.)

Equivalently:

$$\text{id} \circ f = f = f \circ \text{id}.$$

2. **Function composition is associative:** $(h \circ g) \circ f = h \circ (g \circ f)$.

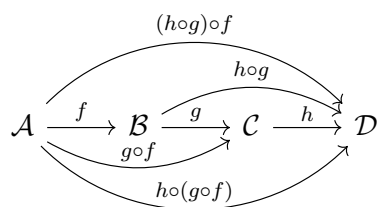
Whichever pair we compose first, we get the same function in the end, so we can unambiguously call this $h \circ g \circ f$. Here, we assume that the domains and codomains of f , g , and h are compatible.

Equivalently:

$$f \circ (g \circ h) = (f \circ g) \circ h.$$

Let’s look more closely at associativity. If we lay out the domains and codomains of

f , g , and h as follows and include all the functions formed from these by composition,



we see that there are multiple paths from $\mathcal{A}$ to $\mathcal{D}$ following the arrows. The associativity property tells us that all these paths represent the same function. Thus the order in which we apply the composition operator among various pairs¹ does not matter. This holds for any number of terms $f_1 \circ f_2 \circ \cdots \circ f_n$ as well.

Puzzle 108. In general, function composition is **not** commutative, that is $f \circ g$ and $g \circ f$ need not be the same (or even both defined). Find functions $f, g: \mathcal{A} \rightarrow \mathcal{A}$ for some $\mathcal{A}$ such that $f \circ g \neq g \circ f$ yet both are defined.

¹The order in which we apply the composition operator does not matter, but the order of the arguments to the composition operator *does* matter. See the next puzzle.

We often use function composition to build or describe complicated functions from simpler pieces.

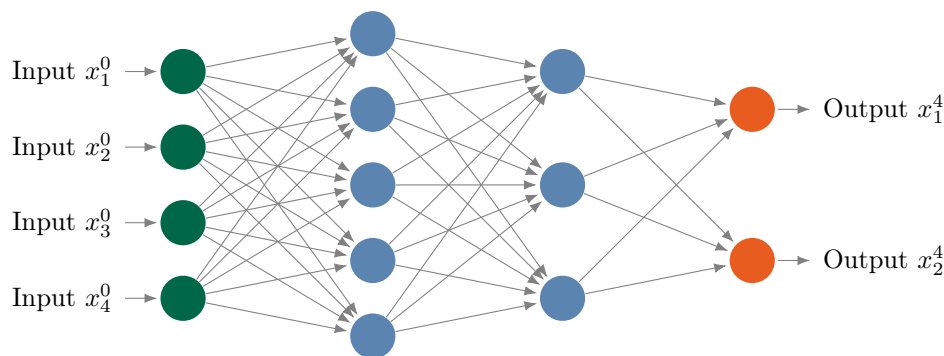


FIGURE 13.3. A four layer, feed-forward neural network with input layer on the left and output layer on the right. Weights on the edges are not illustrated.

Example 13.1 Neural Networks

The basic neural networks that started the modern revolution in data-driven Artificial Intelligence are built on function composition. An example of a *multi-layer, feed-forward neural network* is shown in Figure 13.3.

The computation composes artificial “neurons” (nodes) arranged in “layers”

(columns of nodes). Informations flows from the input layer at the left to the output layer at the right. We count layers from left to right $1 \dots L$ (where $L = 4$ in the Figure) and nodes from top to bottom within layer, $1 \dots n^\ell$ in layer ℓ which has n^ℓ nodes. We pass our original data tuple x^0 to the input layer and receive the processed data, often a prediction or model fit, as a tuple (x^4 in this case) from the output layer.

In this example, there are lots of component indices, so we put indices in both subscripts and superscripts.

Each node represents a function. The function takes as input the signals carried on the edges entering the node from the left and returns a signal, a copy of which is sent along every edge leaving the node on the right. All nodes in a layer receive the same inputs. Specifically when $\ell > 1$, nodes in the ℓ th layer receives inputs $x_k^{\ell-1}$ for $k \in [1 \dots n^{\ell-1}]$ from the nodes in the previous layer and the i th node outputs signal x_i^ℓ . For the input layer $\ell = 1$, the i th node receives only the i th component of the original data x_i^0 .

Write f_i^ℓ for the function represented by the i th node in layer ℓ . When $\ell = 1$ (the input layer), $f_i^1 = \text{id}$, but for $\ell > 1$, f_i^ℓ combines all the input signals entering the node using a nonlinear function ϕ , called the activation function, and exogenous parameters $w_{\ell i}^k$ for $k \in [0 \dots n^{\ell-1}]$. Specifically,

$$f_i^\ell(x \mid w_{ji}) = \phi(w_{ji}^0 + \sum_{k=1}^{n^{\ell-1}} w_{\ell i}^k x_k). \quad (13.7)$$

The exogenous parameters – that is, components of the tuple $w_{\ell i}$ – are called the *weights*. There is a weight $w_{\ell i}^k$ for the edge entering from each node k in the previous layer, which multiplies the output of that node, and a weight $w_{\ell i}^0$, which can be thought of as multiplying the output of an invisible node that always produces a signal 1. The weights are often thought of as attached to the edges in the network. The function ϕ is typically a nonlinear, non-decreasing function, like the “sigmoidal” functions in Example 14.6 or the **relu** function shown in Section 14.2.

In practice, a neural network is “fit” to a data set by finding settings of the weights (exogenous parameters for *all* the nodes) that optimize some criterion on the output. Once the data are fit, we can think of the exogenous parameters as fixed, and because all nodes in a layer get the same inputs, each layer comprises a function f^ℓ that takes those inputs and returns a tuple that concatenates the outputs of all the nodes in layer ℓ . The computation that the network does is

just the composition of these functions

$$f^1 \circ f^2 \circ \dots \circ f^L. \quad (13.8)$$

The first layer receives the data; the second layer receives the output of the first layer; and so on, until the output layer emits the final result.

Puzzle 109. If $f: \mathcal{A} \rightarrow \mathcal{A}$, then we can compose f with itself. For any x , this gives us a sequence $x, f(x), f(f(x)), f(f(f(x))), \dots, f^{\circ k}(x), \dots$ where

$$f^{\circ 0} = \text{id} \qquad f^{\circ k} = \overbrace{f \circ f \circ \dots \circ f}^{k \text{ times}}, \quad \text{for } k \in [1..).$$

If $f: (0_-) \rightarrow (0_-)$ with $f(x) = \frac{1}{2} \left(x + \frac{c}{x} \right)$ for some constant $c > 0$, what does $f^{\circ k}(1)$ look like as k gets large. (Try it for a few values of c .)

Puzzle 110. For any real number x_0 and any function $h: \mathbb{R} \rightarrow \mathbb{R}$,

$$\text{map}(h, \text{iterate}(h, x_0)) = \text{iterate}(h, h(x_0)),$$

where `map` and `iterate` were defined in Puzzle 95. Recall that $\text{map}(h, a) = h \circ a$ for any $a \in \text{Seq}(\mathbb{R})$ and `iterate` produces a sequence like that shown in the last puzzle.

Look at these sequences for a concrete choice of h and x_0 to make sure the meaning of the equation is clear. Explain informally why the sequences on both sides of this equation are equal; prove it if you can. To do this, try looking at the heads and tails of both sequences.

Part of the “Sequences and Streams” series.

Example 13.2 Push-button Automata

Imagine a machine with a button, an LED, and a display screen. We do not observe the machine’s inner workings, but we do know that it records some internal *state* that reflects the machine’s current configuration. We observe only a *signal* that it emits each time the button is pushed. When the LED is off, the machine is in its initial state, and at the first button press, the LED turns on for good. Each time the button is pushed two things happen: 1. the machine computes the emitted signal from its internal state and displays it; and 2. the machine computes the next state from the current state and moves to that

state. The machine thus has two associated functions: an *output function* that computes the signal to emit and a *transition function* that computes the next state from the current state.

A machine is specified by its set of possible states $\mathcal{S}$, set of possible signals $\mathcal{Y}$, output function $s: \mathcal{S} \rightarrow \mathcal{Y}$, transition function $t: \mathcal{S} \rightarrow \mathcal{S}$, and initial state $x_0 \in \mathcal{S}$. When the machine is in state x and the button is pressed, the signal $s(x)$ is emitted and the machine moves to state $t(x)$.

Across pushes of the button, the machine displays the emitted signals $s(x_0), s(t(x_0)), \dots, s(t^{(n-1)}(x_0)), \dots$ and generates (internally) the states $t(x_0), t(t(x_0)), \dots, t^{(n)}(x_0), \dots$ (Here, we use the notation from the Puzzle 109 for iterated composition.)

A few concrete examples may make this more meaningful. First, these machines can simulate discrete dynamical systems. For instance, to simulate a traffic light, let $\mathcal{Y} = \{\text{red}, \text{yellow}, \text{green}\}$ and $\mathcal{S} = \{0, 1, 2\} \times [1 \dots \max(r, y, g)]$, where r, y, g are positive integers giving the number of “cycles” the light shows red, yellow, and green. The first component of the state indicates the current color, and the second component records how many cycles (including the current one) have shown that color. We think of the button as being pushed automatically on the ticks of some clock. The output function computes the color signal from the state’s first component

$$s(j, c) = \begin{cases} \text{red} & \text{if } j = 0 \\ \text{yellow} & \text{if } j = 1 \\ \text{green} & \text{if } j = 2. \end{cases}$$

The transition function ensures that the machine emits **red** r times, then **yellow** y times, then **green** g times, repeating this ad infinitum. When the cycle count hits its maximum for the current color, it switches colors and restarts the cycle count:

$$t(j, c) = \begin{cases} \langle j, c + 1 \rangle & \text{if } (j = 0 \wedge c < r) \vee (j = 1 \wedge c < y) \vee (j = 2 \wedge c < g) \\ \langle j + 1 \bmod 3, 1 \rangle & \text{if } (j = 0 \wedge c = r) \vee (j = 1 \wedge c = y) \vee (j = 2 \wedge c = g). \end{cases}$$

We can start with $x_0 = \langle 0, 1 \rangle$.

Second, these machines can recognize a wide variety of “languages.” For instance, we can use integers to represent finite strings of bits, with the least

significant bit first. Large enough integers can encode any practical string of information in this way. Let's create a machine that determines whether the binary representation of an integer has three consecutive 1 bits. Let $\mathcal{Y} = \mathbb{B}$, the Booleans, and $\mathcal{S} = [0..] \times [0..] \times \mathbb{B}$, with $x_0 = \langle n, 0, \perp \rangle$ for some n . The state encodes three pieces of data: (i) the unprocessed bit string as an integer, (ii) the length of the current run of consecutive 1 bits, and (iii) whether three consecutive 1 bits have been seen so far. The initial state has the input bit string (as an integer n) as its first component. The emitted signal starts as false and will eventually be true if the binary representation of n has three consecutive 1 bits; otherwise, it will always be false. So, $s(a, b, d) = d$ simply extracts whether three 1 bits have been seen from the state. The transition function strips off the least significant bit of n , counting the number of consecutive bits seen so far (and resetting the count to 0 if it ever sees a 0). Once the remainder (the first component of the tuple) becomes 0, the machine stays in that state forever. Hence, the transition function is

$$t(a, b, d) = \begin{cases} \langle 0, b, d \rangle & \text{if } a = 0 \\ \langle 0, 3, \top \rangle & \text{if } b + (a \bmod 2) = 3 \\ \langle a \operatorname{div} 2, b + (a \bmod 2), \perp \rangle & \text{if } a \bmod 2 = 1 \\ \langle a \operatorname{div} 2, 0, \perp \rangle & \text{otherwise,} \end{cases}$$

where div is integer division.

Third, these machines can do a wide variety of computations. For instance, let $\mathcal{Y} = \mathbb{R}$ and $\mathcal{S} = \mathbb{R} \times \mathcal{A} \times [0..]$ where $\mathcal{A}$ is the set of functions $\mathbb{R} \rightarrow \mathbb{R}$ that are “analytic at 0,” meaning that the function has a derivative that is also “analytic at 0.” Define $s(z, f, n) = z + f(0) \frac{x^n}{n!}$ and $t(z, f, n) = \langle s(z, f, n), f', n + 1 \rangle$. Then, for any initial state $x_0 = \langle 0, f, 0 \rangle$, the emitted signals after successive button pushes are the partial sums $\sum_{k=0}^{n-1} f^{(k)}(0) \frac{x^k}{k!}$ which gives the *Taylor series for f* . For instance, when $f(x) = e^x$, we get $\sum_{k=0}^{n-1} \frac{x^k}{k!}$.

13.2 Commutative Diagrams

When working with functions built with composition, we will often encounter equations like $r \circ f = h \circ g$ that describe the relationship among various functions. These equations are often more clearly expressed, and easier to use, with a *diagram*. For example, the equation $r \circ f = h \circ g$ can be represented by the directed graph

$$\begin{array}{ccc}
 \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\
 \downarrow g & & \downarrow r \\
 \mathcal{C} & \xrightarrow{h} & \mathcal{D}
 \end{array}$$

Each node of this directed graph is a set and each edge is a function with the source node as its domain and the target node as its codomain. (Hence, $f: \mathcal{A} \rightarrow \mathcal{B}$, $g: \mathcal{A} \rightarrow \mathcal{C}$, and so forth.) Following a path in the graph gives a composition of the functions along that path, in diagrammatic order. Thus, the path from $\mathcal{A}$ to $\mathcal{D}$ along the edge f then the edge r corresponds to the function $f \circ r = r \circ f$. The fundamental constraint on these graphs is that *any distinct paths between two nodes represent equal functions*. In diagram above, we have two paths from $\mathcal{A}$ to $\mathcal{D}$, f then r and g then h , so the diagram expresses the equality $f \circ r = g \circ h$, or equivalently $r \circ f = h \circ g$. As another example, the identity property of composition ($\text{id} \circ f = f = f \circ \text{id}$) can be expressed as the following diagram:

$$\begin{array}{ccc}
 \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\
 \downarrow \text{id} & \searrow f & \downarrow \text{id} \\
 \mathcal{A} & \xrightarrow{f} & \mathcal{B}
 \end{array}$$

for domain $\mathcal{A}$ and codomain $\mathcal{B}$. Any path from upper left to lower right gives one of the functions in the equation. We call these graphs *commutative diagrams*.

A **commutative diagram** is a directed graph whose nodes are sets and whose edges are functions with those sets as domains or codomains, as appropriate. This graph satisfies:

1. A path between two nodes represents the composition of the functions along the edges of the path, in diagrammatic order.
2. Any distinct paths between two nodes represent *equal functions*.

We say colloquially that a diagram “commutes” to indicate that such a graph is a commutative diagram.

As practice for reasoning about compositions, especially using commutative diagrams, consider the path of a dragonfly through space as it flies over some time interval.² We measure at each instant the dragonfly’s location in three-dimensional space giving a function $\text{Time} \rightarrow \text{Space}$. We could instead measure just the dragonfly’s height above the ground at any instant, giving a function $\text{Time} \rightarrow \text{Height}$, or alternatively, the position on the ground directly beneath it at any instant giving a function $\text{Time} \rightarrow \text{Position}$, Figure 13.4 shows these functions (called s , h , and p

²This example is motivated by [Lawvere*2005].

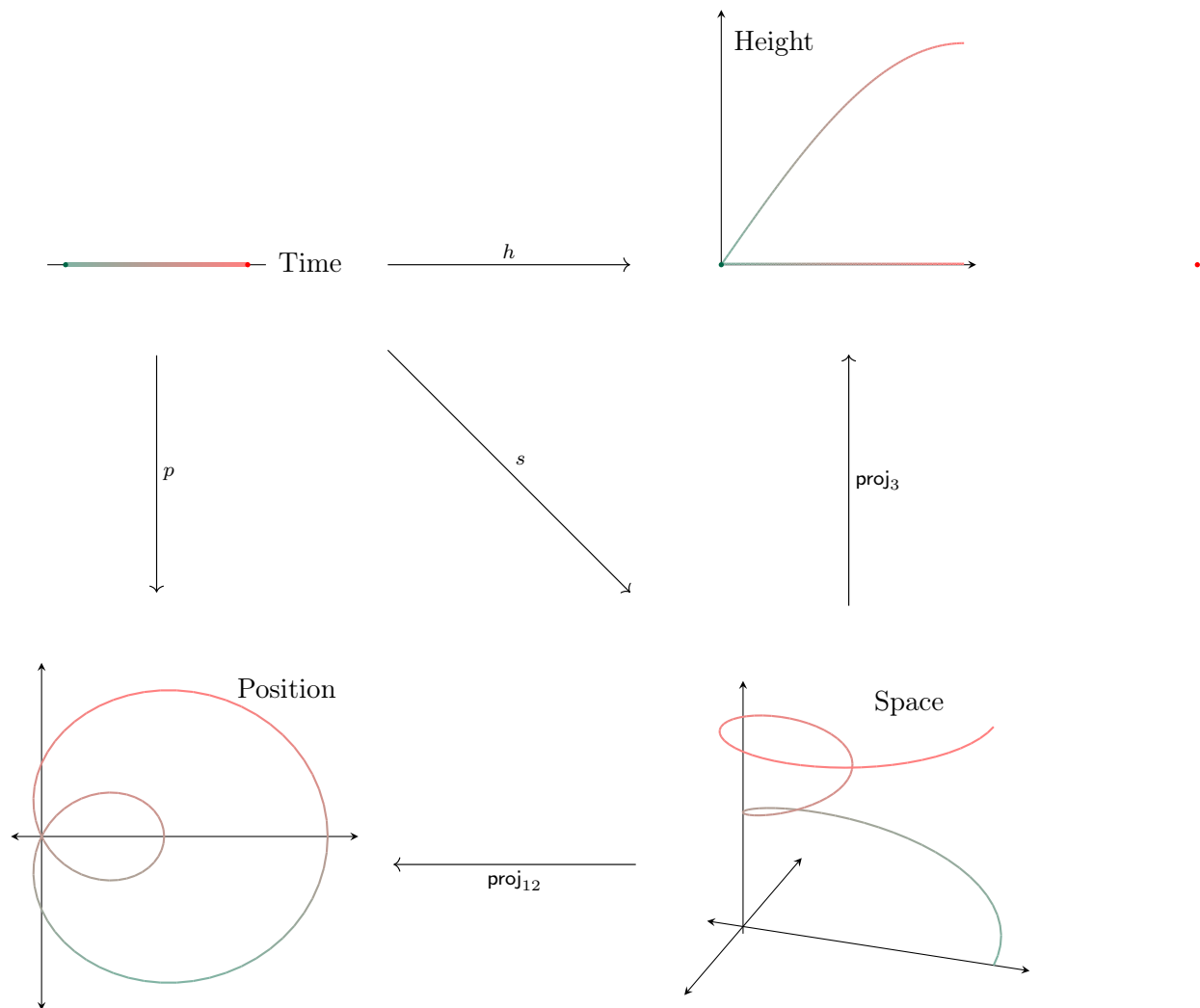


FIGURE 13.4. Functions representing a dragonfly's path arranged in a commutative diagram that reflects the relationships among them. The height in the top right graph includes the time dimension for clarity.

respectively) corresponding to a specific path for the dragonfly. The functions are related by a commutative diagram, as depicted, because the dragonfly's location in space at any time determines both its height and ground position. Specifically, we can find $h(t)$ by taking the third coordinate of $s(t)$, which we write as $\text{proj}_3(s(t))$, and we can find $p(t)$ by taking the first two coordinates of $s(t)$, which we write as $\text{proj}_{12}(s(t))$.

$$\begin{array}{ccc}
 \text{Time} & \xrightarrow{h} & \text{Height} \\
 \downarrow p & \searrow s & \uparrow \text{proj}_3 \\
 \text{Position} & \xleftarrow{\text{proj}_{12}} & \text{Space}
 \end{array} \tag{13.9}$$

The diagram (here and in the Figure) tells us that $h = \text{proj}_3 \circ s$ and $p = \text{proj}_{12} \circ s$, which embodies the relationships among these functions.

Now turn this around, and suppose I told you only the dragonfly's height h as a function of time and its ground position p as a function of time. Can we reconstruct its location in space as a function of time? Yes – if the h and p measurements are *consistent* with the relationships in the diagram. From any reconstructed path, extracting height and position functions from that path should just give us back h and p . When h and p are consistent, there is a *unique* function s that makes the diagram commute.

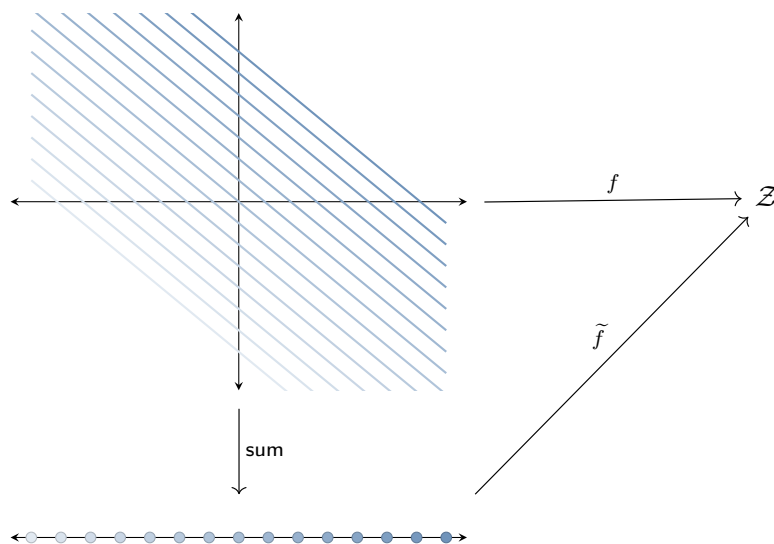


FIGURE 13.5. Commutative diagram factoring a function $\mathbb{R}^2 \rightarrow \mathcal{Z}$ from Example 13.3 that is constant on diagonal lines as shown, with the sum of components on each line represented by its color.

Example 13.3.

Consider functions that take as inputs points in the plane $\mathbb{R}^2$ but whose return values depend only on the *sum of the two components*. Examples include $p(x, y) = (x + y)^2$, $w(x, y) = e^{-\frac{x+y}{2}}$, and $h(x, y) = \langle \cos(x + y), \sin(x + y) \rangle$. All of these examples look like some other function applied to $x + y$, and indeed we can write $p = \tilde{p} \circ \text{sum}$, $w = \tilde{w} \circ \text{sum}$, and $h = \tilde{h} \circ \text{sum}$, where $\text{sum}(x, y) = x + y$ and $\tilde{p}(r) = r^2$, $\tilde{w}(r) = e^{-r/2}$, and $\tilde{h}(r) = \langle \cos(r), \sin(r) \rangle$. Thus, each of these functions “factors” into a composition of a function taking real numbers with the function sum .

A general function $f: \mathbb{R}^2 \rightarrow \mathcal{Z}$ for some $\mathcal{Z}$ whose return value depends only on the sum of input components factors similarly. Figure 13.5 shows the situation with a commutative diagram. The function f is constant on each line parallel to the depicted diagonal lines in $\mathbb{R}^2$; the sampled lines are colored to indicate the value of sum for points on that line. The correspondence is shown in the points in the bottom plot. That f is constant on such lines implies the existence of a function $\tilde{f}: \mathbb{R} \rightarrow \mathcal{Z}$ that makes the diagram commute, i.e., f factors as $f = \tilde{f} \circ \text{sum}$.

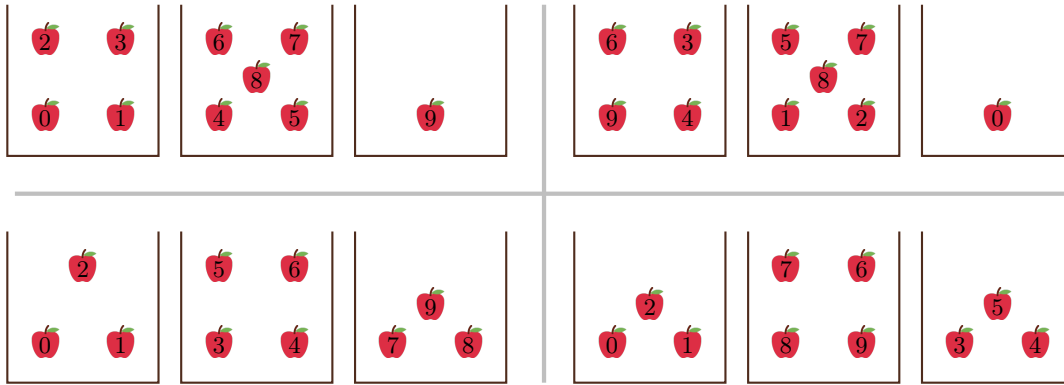


FIGURE 13.6. Four assignments of 10 apples to 3 buckets. Each row gives a pair of assignments that are indistinguishable if the apples are unlabeled. Assignments in different rows are distinguishable.

Example 13.4 Unlabeled Apples in Buckets

In Example 10.8, we consider functions that describe ways to allocate apples from a finite set $\mathcal{A}$ to buckets from a finite set $\mathcal{B}$. There, we think of each apple (element of $\mathcal{A}$) as a distinct and identifiable object; so a function that put apple

1 in bucket 1 and the rest in bucket 2 would be *different* than a function that put apple 2 in bucket 1 and the rest in bucket 2.

Suppose, however, that we don't want to distinguish two ways of distributing apples if they only differ in the labels of the apples. (We still distinguish by the labels on the buckets, for the moment.) When we distinguished labels, we could tell if two functions $\mathcal{A} \rightarrow \mathcal{B}$ represented the same way of distributing apples by checking if they were equal. But now, how can we tell?

Figure 13.6 shows four ways to assign 10 apples to 3 buckets. If we pay attention to the labels on the apples in the Figure, all four assignments are distinct. But if we disregard the labels, both assignments in the first row are indistinguishable because they differ only by a rearrangement of the labels – a relabeling of the apples. Both the assignments in the second row are similarly indistinguishable from each other if we disregard the labels, though the assignments in the two rows are distinguishable either way.

Let $m = \#\mathcal{A}$ and list out the elements of $\mathcal{A}$ (the labeled apples) in some arbitrary but fixed order $\langle a_1, a_2, \dots, a_m \rangle$. A permutation $p: \mathcal{A} \rightarrow \mathcal{A}$ is a function that maps every element of $\mathcal{A}$ to a unique element of $\mathcal{A}$, which has the effect of rearranging the list of apples. So, if the elements of $\mathcal{A}$ are $\langle 1, 2, 3, 4 \rangle$, a permutation p might map these values to $\langle 4, 3, 1, 2 \rangle$ in corresponding order. (Note $a \neq a' \in \mathcal{A}$ implies that $p(a) \neq p(a')$ and every $a' \in \mathcal{A}$ equals $p(a)$ for some a . In the next section, we will call this a bijection.) Think of applying such a permutation to $\mathcal{A}$ as taking the labels off the apples, rearranging them, and putting the labels back on. A permutation is a **relabeling of the apples**.

Two ways of distributing labeled apples into labeled buckets are the same – disregarding the apples labels – if they could be made equal by relabeling the apples. That is, $f, g: \mathcal{A} \rightarrow \mathcal{B}$ are equivalent up to the labeling of the apples if *there is a permutation p of the apples such that the following diagram commutes*:

$$\begin{array}{ccc} \mathcal{A} & & \\ \downarrow p & \searrow f & \\ \mathcal{A} & \xrightarrow{g} & \mathcal{B} \end{array}$$

In equation terms, this diagram means that $f = g \circ p$. The $\searrow$ arrow for p indicates that p is a *bijection*, a property discussed in Chapter 14 that ensures it is a permutation of $\mathcal{A}$. (Chapter 18 describes how to count the number of unique assignments, with and without labels.)

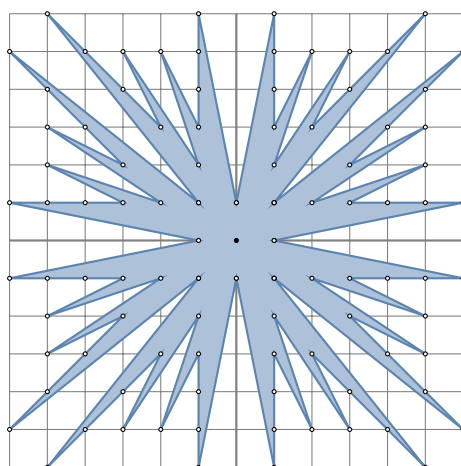


FIGURE 13.7. A polygon with vertices at integer coordinates. Integer boundary points are marked with $\circ$ and interior points with $\bullet$. This is the Farey sunburst, see [WikiFarey].

Example 13.5 Pick's Formula

Figure 13.7 displays a polygon all of whose vertices have coordinates of the form $\langle j, k \rangle$ where j and k are *integers*. With the wonderful Pick's formula, we know that the area enclosed by this polygon is 48.

Let $\mathcal{P}$ be the set of polygons whose vertices have integer coordinates. Given such a polygon, Pick's formula gives its area in terms of the number m integer points in the interior of the polygon and n integer points on the boundary of the polygon. Try to derive Pick's formula for yourself without looking at the answer in the margin; start with simple triangles and squares and go from there.

Define $\text{pick}(m, n)$ to be Pick's formula and $\text{grid}(p) = \langle m, n \rangle$, where m and n are defined as above for the polygon p . Then, Pick's formula represents a compositional relationship as depicted in the following commutative diagram:

$$\begin{array}{ccc}
 \mathcal{P} & \xrightarrow{\text{area}} & [0\_) \\
 \downarrow \text{grid} & \nearrow \text{pick} & \\
 [0 \dots) \times [3 \dots) & &
 \end{array}$$

This captures a common situation where we have a set of complicate objects and a function of interest that only depends on *some features of the object*. The diagram shows how we factor our those features for computing the function of interest.

$$I - \frac{z}{u} + uu$$

Note that we can apply Pick's formula more generally. For real numbers $\alpha, \beta > 0$, let $\mathcal{P}_{\alpha, \beta}$ be the set of polygons all of whose vertices have coordinates of the form $\langle \alpha j, \beta k \rangle$ for integers j, k . For any $p \in \mathcal{P}_{\alpha, \beta}$, scaling the x - and y -coordinates by $1/\alpha$ and $1/\beta$ respectively, gives a polygon in $\mathcal{P} = \mathcal{P}_{1,1}$. Applying Pick's formula and scaling the result by $\alpha\beta$ yields the area of the polygon.

$$(\mathbb{I} - \frac{z}{u} + w)g^v$$

Example 13.6 A Second Look at Cartesian Products and Disjoint Unions

Given two sets $\mathcal{A}$ and $\mathcal{B}$, we defined both their *Cartesian product* $\mathcal{A} \times \mathcal{B}$ in equation (9.8) and their *disjoint union* $\mathcal{A} \sqcup \mathcal{B}$ in equation (9.10) in terms of tuples. Specifically, $\mathcal{A} \times \mathcal{B}$ is the set of tuples $\langle a, b \rangle$ with $a \in \mathcal{A}$ and $b \in \mathcal{B}$; and $\mathcal{A} \sqcup \mathcal{B}$ is the set of tuples of the form $\langle a, 1 \rangle$ for $a \in \mathcal{A}$ or $\langle b, 2 \rangle$ for $b \in \mathcal{B}$, where 1 and 2 are arbitrary but distinct “tags.” (See also Example 10.10.)

Both $\mathcal{A} \times \mathcal{B}$ and $\mathcal{A} \sqcup \mathcal{B}$ have special functions associated with them that connect each of the sets to the components $\mathcal{A}$ and $\mathcal{B}$. In the former case, we have *projections* onto $\mathcal{A}$ and $\mathcal{B}$, and in the latter case, we have *injections* from $\mathcal{A}$ and $\mathcal{B}$.

$$\begin{array}{ccc} \mathcal{A} \times \mathcal{B} & \xrightarrow{\text{proj}_2} & \mathcal{B} \\ \text{proj}_1 \downarrow & & \downarrow \text{inj}_2 \\ \mathcal{A} & & \mathcal{A} \sqcup \mathcal{B} \end{array} \quad \begin{array}{c} \mathcal{A} \xrightarrow{\text{inj}_1} \mathcal{A} \sqcup \mathcal{B} \end{array} \quad (13.10)$$

Here, $\text{proj}_1(\langle a, b \rangle) = a$ and $\text{proj}_2(\langle a, b \rangle) = b$ extracts from the tuple in the Cartesian product the first and second components, respectively. And $\text{inj}_1(a) = \langle a, 1 \rangle$ and $\text{inj}_2(b) = \langle b, 2 \rangle$ wraps a value in $\mathcal{A}$ or $\mathcal{B}$, respectively, into the disjoint union by tagging those values appropriately.

The Cartesian product satisfies the property that for *any* set $\mathcal{C}$ with functions p_1, p_2

$$\begin{array}{ccccc} \mathcal{A} \times \mathcal{B} & & \xrightarrow{\text{proj}_2} & & \mathcal{B} \\ \text{proj}_1 \downarrow & \nwarrow q & & \nearrow p_2 & \\ \mathcal{A} & & \xleftarrow{p_1} & & \mathcal{C} \end{array} \quad (13.11)$$

there is a *unique* function q that makes this diagram commute. We will call this function the *fork* of p_1 and p_2 and denote it by $p_1 \vee p_2$, where $(p_1 \vee p_2)(c) = \langle p_1(c), p_2(c) \rangle$. (This is discussed further in “Joining Tuples” in Chapter 15.)

Similarly, the disjoint union satisfies the property that for *any* set $\mathcal{C}$ with

functions ℓ_1, ℓ_2

$$\begin{array}{ccc}
 \mathcal{C} & \xleftarrow{\ell_2} & \mathcal{B} \\
 \ell_1 \uparrow & \swarrow u & \downarrow \text{inj}_2 \\
 \mathcal{A} & \xrightarrow{\text{inj}_1} & \mathcal{A} \sqcup \mathcal{B}
 \end{array} \tag{13.12}$$

there is a unique function u that makes this diagram commute. We call this function *destructuring* with ℓ_1 and ℓ_2 and denote it by $\ell_1 \sqcup \ell_2$. It is defined by $(\ell_1 \sqcup \ell_2)(\langle x, j \rangle) = \ell_j(x)$. (This is also discussed in Example 10.10.)

Puzzle 111. What is q in diagram (13.11) when $\mathcal{C} = \mathcal{B} \times \mathcal{A}$ and $p_i = \text{proj}_{3-i}$ for $i \in \{1, 2\}$? What is u in diagram (13.12) when $\mathcal{C} = \mathcal{B} \sqcup \mathcal{A}$ and $\ell_i = \text{inj}_{3-i}$?

Example 13.7 Extensions

Consider the function $f: [1..] \rightarrow [1..]$ defined by $f(n) = (n-1)!$, where $!$ denotes the factorial operator. Can we find a function $(0..) \rightarrow (0..)$ that smoothly interpolates f , i.e., equals f for positive integers? The answer is yes; in fact, there are infinitely many such functions. Such a function is an *extension* of f from the positive integers to the positive real numbers. An example of such a function is the *Gamma Function* Γ (see Appendix B), which has $\Gamma(n) = (n-1)!$, $\Gamma(1/2) = \sqrt{\pi}$, $\Gamma(3/2) = \frac{1}{2}\sqrt{\pi}$, and so forth. In fact, Γ extends the factorial function to all the complex numbers except for the non-positive integers.

In general, an extension of a function creates a new function that takes additional inputs while giving the same value as the original function when given the original inputs. Formally, if $f: \mathcal{A}_0 \rightarrow \mathcal{B}$ and $\mathcal{A}_0 \subseteq \mathcal{A}$, then an *extension* of f from $\mathcal{A}_0$ to $\mathcal{A}$ is a function $\tilde{f}: \mathcal{A} \rightarrow \mathcal{B}$ for which the following diagram commutes:

$$\begin{array}{ccc}
 \mathcal{A} & & \\
 \text{incl} \uparrow & \searrow \tilde{f} & \\
 \mathcal{A}_0 & \xrightarrow{f} & \mathcal{B}
 \end{array}$$

Recall that the inclusion incl maps $\mathcal{A}_0$ to $\mathcal{A}$ via $\text{incl}(x) = x$. The hook on the arrow, which looks like a subset sign, indicates an inclusion. Thus, the diagram tells us that $\tilde{f}$ must equal f for inputs in $\mathcal{A}_0$.

When we extend a function in practice, we usually do so to ensure that $\tilde{f}$ has some desirable property, such as *smoothness* of interpolation described above.

Example 13.8 FRPs, Kinds, and Statistics

Let ψ be a Statistic, let $\mathcal{F}$ be the set of FRPs compatible with ψ , and let $\mathcal{K}$ be the corresponding set of their kinds. Then, as we saw in Part I, the following diagram commutes:

$$\begin{array}{ccc} \mathcal{F} & \xrightarrow{\psi} & \mathcal{F} \\ \text{kind} \downarrow & & \downarrow \text{kind} \\ \mathcal{K} & \xrightarrow{\psi} & \mathcal{K} \end{array}$$

There is some abuse of notation here; three distinct but related functions claiming the mantle of “ ψ .” The statistic ψ is a function from values to values. For an FRP X with kind k , we write $\psi(X)$ and $\psi(k)$ for the FRP and kind *transformed by* ψ . But ψ is not really a function on $\mathcal{F}$ or $\mathcal{K}$, instead there are related functions, call them ψ_* on $\mathcal{F}$ and ψ_{**} on $\mathcal{K}$, that represent the corresponding transformation of $\mathcal{F}$ and $\mathcal{K}$. In practice, though we elide this distinction and just call them all ψ because which one we mean is always clear from context.

Alternatively, we could think of ψ as an *extension* of the original statistic from $\text{dom}(\psi)$ to $\text{dom}(\psi) \sqcup \mathcal{F} \sqcup \mathcal{K}$, as described in the previous example.

13.3 Inverse Functions

Consider the function illustrated in the top left of Figure 8.5. For any value returned by the function, we can determine which input produced it, and thus we can define a *function* that maps the outputs to their corresponding inputs. Such functions are said to be *invertible* and to be *inverses* of each other. In contrast, the function in the bottom right of Figure 8.5 does not have this property; if we see a 7 on the output, we cannot tell which input die roll produced it.

A familiar pair of inverse functions is the exponential function and the natural logarithm. Define $f: \mathbb{R} \rightarrow (0_+)$ by $f(x) = e^{cx}$ for a constant c and $g: (0_+) \rightarrow \mathbb{R}$ by $g(y) = \frac{1}{c} \ln(y)$. We have $\text{dom}(f) = \text{codom}(g)$ and $\text{codom}(g) = \text{dom}(f)$, so we can compute the logarithm of an exponential and the exponential of a logarithm. That is, we can compose f and g in both directions: for any real x and positive y :

$$\begin{aligned} g(f(x)) &= \frac{1}{c} \ln(e^{cx}) = \frac{1}{c} \cdot cx = x \\ f(g(y)) &= e^{c \cdot \frac{1}{c} \ln(y)} = e^{\ln(y)} = y. \end{aligned}$$

This is more simply expressed as $f \circ g = \text{id} = g \circ f$. We can determine the input of

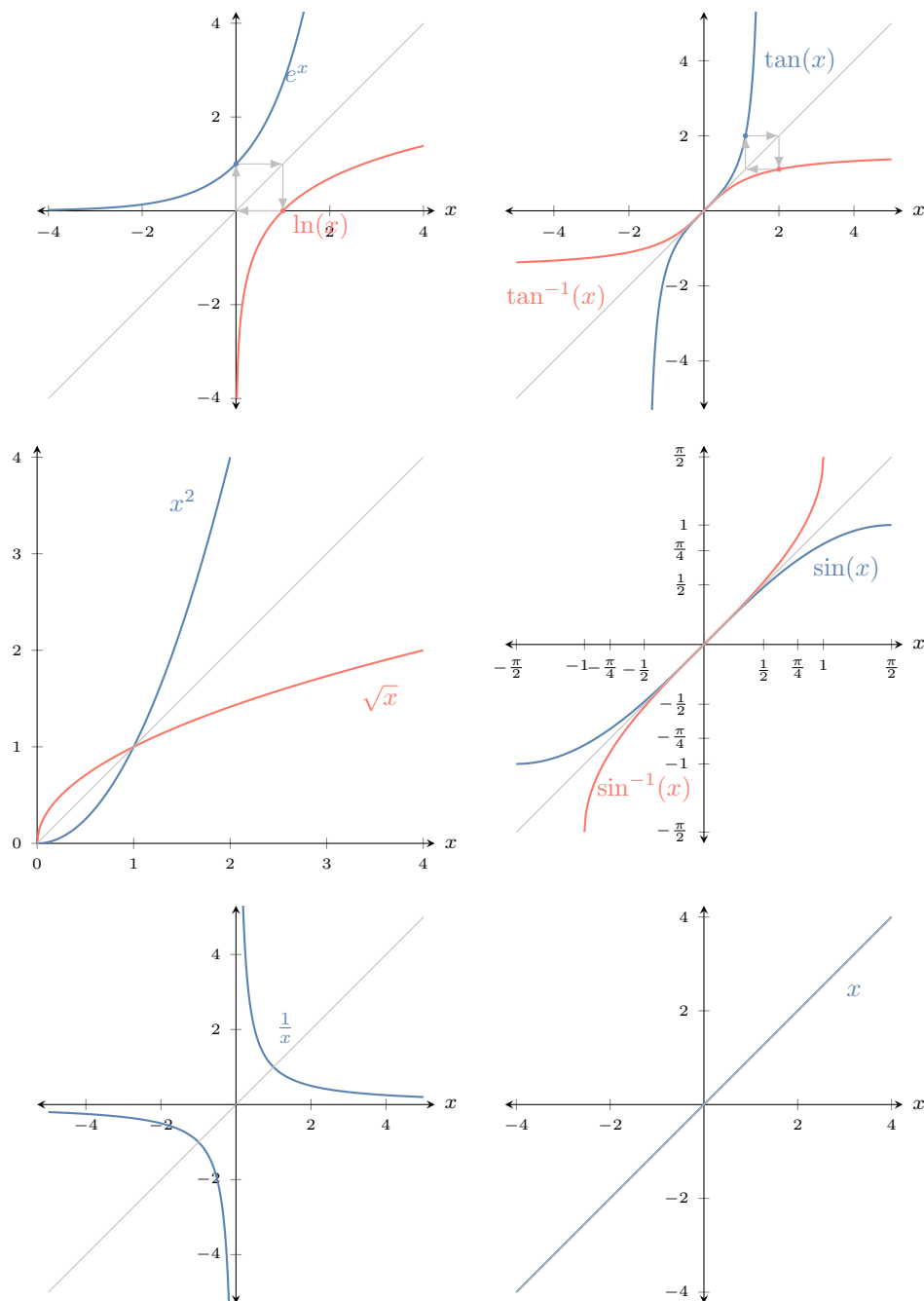


FIGURE 13.8. Several numeric functions and their inverses on the same axes. Each function's graph is the mirror image across the $y = x$ line of its inverse. In the top two panels, this reflection is illustrated for a pair of points with gray arrows. The bottom two functions are their own inverse, as can be seen by the reflection.

either function from its output, using the other function. Graphs of these and other invertible numeric functions are shown in Figure 13.8.

Definition. If $f: \mathcal{A} \rightarrow \mathcal{B}$ and $g: \mathcal{B} \rightarrow \mathcal{A}$ satisfy

$$f \circ g = \text{id} = g \circ f \quad (13.13)$$

we say that f and g are *invertible*. We call g the **inverse function** of f and f the inverse function of g .

The relationships between inverse functions can also be expressed in two commutative diagrams:

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\ & \searrow \text{id} & \downarrow g \\ & & \mathcal{A} \end{array} \quad \begin{array}{ccc} \mathcal{B} & \xrightarrow{g} & \mathcal{A} \\ & \searrow \text{id} & \downarrow f \\ & & \mathcal{B} \end{array} \quad (13.14)$$

Both are required for f and g to be inverses; in general, neither implies the other.

If $h: \mathcal{X} \rightarrow \mathcal{Y}$ is invertible, we write h^{-1} to denote its inverse $\mathcal{Y} \rightarrow \mathcal{X}$.

The superscript notation for the inverse, where f^{-1} denotes the inverse function of f , is slightly ambiguous with notation because it looks like a power. For instance, we often write $\sin^2$ or $\sin^2(x)$ for the square of the sine. We use inverses much more frequently than negative powers of a function, so we will use $^{-1}$ to denote inverses only and avoid negative powers of a function completely. As the inverse notation is well entrenched and lacking a meaningful alternative, this is a worthwhile tradeoff.

Puzzle 112. Is id invertible? What is id^{-1} ?

Note that the definition of inverse functions does not require that the pair of functions be distinct. Suppose, however, I claim that both $f \circ h = \text{id}$ and $g \circ f = \text{id}$. What can we conclude about g and h ?³ These equations correspond to the commutative diagrams

$$\begin{array}{ccc} \mathcal{B} & \xrightarrow{h} & \mathcal{A} \\ \downarrow h & \searrow \text{id} & \downarrow f \\ \mathcal{A} & \xrightarrow{f} & \mathcal{B} \end{array} \quad \begin{array}{ccc} \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\ \downarrow f & \searrow \text{id} & \downarrow g \\ \mathcal{B} & \xrightarrow{g} & \mathcal{A} \end{array}$$

When we join two commutative diagrams on a common edge, the resulting diagram still commutes:

³Notice that we have only assumed that one of the two equations in (13.13) each of g and h , and importantly a different equation for each.

$$\begin{array}{ccccc}
 \mathcal{B} & \xrightarrow{h} & \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\
 \downarrow h & \searrow \text{id} & \downarrow f & \searrow \text{id} & \downarrow g \\
 \mathcal{A} & \xrightarrow{f} & \mathcal{B} & \xrightarrow{g} & \mathcal{A}
 \end{array}$$

This shows that the two paths from top left to bottom right East-Southeast ($\mathcal{B} \xrightarrow{h} \mathcal{A} \xrightarrow{\text{id}} \mathcal{B}$) and Southeast-East ($\mathcal{B} \xrightarrow{\text{id}} \mathcal{A} \xrightarrow{g} \mathcal{B}$) are equal functions, meaning that $h = g$. This reflects the identity and associativity properties of composition:

$$g = \text{id} \circ g = (h \circ f) \circ g = h \circ (f \circ g) = h \circ \text{id} = h.$$

A corollary of this result is that the inverse of a function, if it exists, is unique!

As an example, let the function R_θ rotate a point in the plane counter-clockwise about the origin by an angle θ . If $v = \langle v_1, v_2 \rangle \in \mathbb{R}^2$ is a point and $v' = R_\theta(v)$, then

$$v'_1 = v_1 \cos(\theta) - v_2 \sin(\theta)$$

$$v'_2 = v_1 \sin(\theta) + v_2 \cos(\theta).$$

The inverse of this function is a rotation in the other direction of the same magnitude, i.e., $R_\theta^{-1} = R_{-\theta}$. With a bit of arithmetic and the identity $\cos^2(a) + \sin^2(a) = 1$, you can confirm that $R_{-\theta} \circ R_\theta = \text{id} = R_\theta \circ R_{-\theta}$.

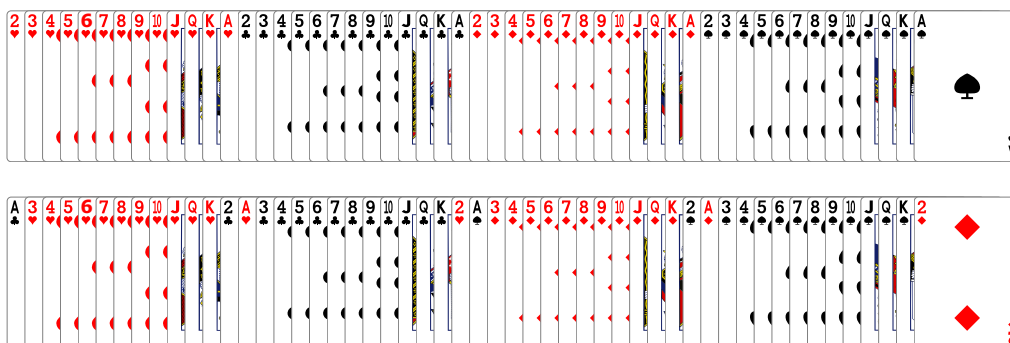


FIGURE 13.9. A permutation of a standard card deck that swaps $2\heartsuit$, $A\clubsuit$; $A\heartsuit$, $2\clubsuit$; $2\diamondsuit$, $A\spadesuit$; $A\diamondsuit$, $2\spadesuit$. The correspondence from top arrangement to bottom arrangement represents the function; going from bottom to top represents the inverse.

If we think of a deck of cards as a list of specific cards in order from top to bottom, then any rearrangement of a deck (e.g., moving the top card to the bottom or interleaving the top and bottom halves of the deck) is a function from the set of orderings $\mathcal{D}$ to itself. And any such rearrangement is invertible, we simply reverse the steps. See Figure 13.9. Indeed, if $\mathcal{A}$ is a finite set, we define a **permutation** of $\mathcal{A}$ as

any invertible function $p: \mathcal{A} \rightarrow \mathcal{A}$. Think of p as a relabeling of the elements of $\mathcal{A}$, where $a \in \mathcal{A}$ gets relabeled as $p(a)$. The inverse permutation p^{-1} just relabels $p(a)$ with a .

Puzzle 113. If $f: \mathcal{A} \rightarrow \mathcal{B}$ and $g: \mathcal{B} \rightarrow \mathcal{C}$ are both invertible, use diagrams like those in (13.14) to argue that $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$. Explain the order reversal here in a sentence or two.

Puzzle 114. An invertible function $i: \mathcal{A} \rightarrow \mathcal{A}$ with $i = i^{-1}$ is called an *involution*. Find three examples of involutions, excluding id .

Puzzle 115. Consider the function $s: [1..5] \rightarrow \mathbb{Z}$ defined by $s(k) = k(k+1)$. For any possible output of this function, we can determine what the original input was unambiguously, yet there is no function $t: \mathbb{Z} \rightarrow [1..5]$ that is an inverse function to s . Why? What can we do about this?

The previous puzzle reminds us that the domain and codomain are part of the function, but remember that we can use restriction (defined on page 583) and inclusion (defined on page 563) to transparently cast a function's type to appropriate domain and/or codomain. In this way, for example, we can write $s: [1..5] \rightarrow \{2, 6, 12, 20, 30\}$ and then an inverse function $t: \{2, 6, 12, 20, 30\} \rightarrow [1..5]$.

Both equalities in equation (13.13), and both diagrams in (13.14), must hold for a function to have an inverse, but each can be of interest separately. We say that $f: \mathcal{A} \rightarrow \mathcal{B}$ has a **pre-inverse** $g: \mathcal{A} \rightarrow \mathcal{B}$ if $f \circ g = \text{id}$. That is, *pre*-composing with g (“ g then f ”) gives the identity.

$$\begin{array}{ccc} \mathcal{B} & \xrightarrow{g} & \mathcal{A} \\ & \searrow \text{id} & \downarrow f \\ & & \mathcal{B} \end{array}$$

Similarly, we say that f has a **post-inverse** $h: \mathcal{A} \rightarrow \mathcal{B}$ if $h \circ f = \text{id}$.⁴ That is, *post*-composing with h (“ h after f ”) gives the identity.

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\ & \searrow \text{id} & \downarrow h \\ & & \mathcal{A} \end{array}$$

⁴The pre- and post-inverse are also known as right- and left-inverses because of its position in the syntactic order of composition.

If h is a post-inverse of f , then f is a pre-inverse of h , and vice versa. As we saw above, if f has *both* a pre-inverse and a post-inverse, then the two functions must be equal, giving f^{-1} .

For example, consider the functions $f: \mathbb{R} \rightarrow [0_\infty)$ and $g: [0_\infty) \rightarrow [0_\infty)$ with $f(x) = x^2$ and $g(y) = \sqrt{y}$. We have $f \circ g = \text{id}$ but $g \circ f \neq \text{id}$. If we take the square root of a non-negative number and then square the result, we get the number back. But if we take the square root of a square x^2 , we cannot determine if the original input was x or $-x$. Thus, g (sqrt) is a pre-inverse of f (square), and f is a post-inverse of g .

The existence of a pre- or post-inverse for a function allows us to effectively cancel a function by pre- or post-composition. For instance, for g (sqrt) and f (square) from the previous paragraph, if we have a function of the form $g \circ h$, then post-composing with f gives $f \circ g \circ h = h$. Squaring a square root “cancels” it out.

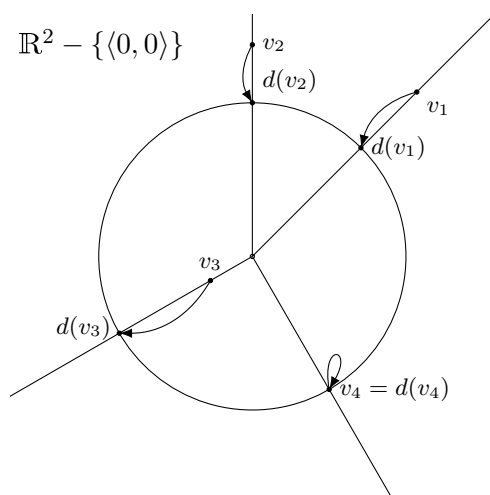


FIGURE 13.10. The mapping from (non-zero) points in the plane to their directions from the origin as a point on the unit circle, in Example 13.9.

Example 13.9 Directions The plane $\mathbb{R}^2$ consists of all pairs $\langle x, y \rangle$ of real numbers. We can measure the distance between two such points $\langle x, y \rangle$ and $\langle u, w \rangle$ by $\sqrt{(x-u)^2 + (y-w)^2}$. The unit circle in the plane, denoted $\mathbb{S}^1$, is the set of points distance 1 from the origin $\langle 0, 0 \rangle$, $\mathbb{S}^1 = \{ \langle x, y \rangle \mid x^2 + y^2 = 1 \}$. Excluding the origin, we get the set $\mathbb{R}^2 - \{ \langle 0, 0 \rangle \}$. Every point $v = \langle v_1, v_2 \rangle$ in that set defines a *direction* that represents a ray from the origin through v . Each such ray corresponds to the point on $\mathbb{S}^1$ where the ray intersects $\mathbb{S}^1$, so we can

define the function $d: \mathbb{R}^2 - \{(0, 0)\} \rightarrow \mathbb{S}^1$ to return the direction of every point, where

$$d(x, y) = \left\langle \frac{x}{\sqrt{x^2 + y^2}}, \frac{y}{\sqrt{x^2 + y^2}} \right\rangle$$

This function is illustrated in Figure 13.10.

The inclusion function $\text{incl}: \mathbb{S}^1 \rightarrow \mathbb{R}^2 - \langle 0, 0 \rangle$, defined by $\text{incl}(x, y) = \langle x, y \rangle$, gives

$$d \circ \text{incl} = \text{id}$$

$$\text{incl} \circ d \neq \text{id}$$

If we start on the circle, d keeps us there, but every point on a ray from the origin maps to the same point on the circle. So, incl is a pre-inverse of d and d is a post-inverse of incl .

One way to think of this pair is to see the rays emanating from the origin as “fibers” that make up the set $\mathbb{R}^2 - \langle 0, 0 \rangle$. Every point lies on exactly one fiber, and any two points on the same fiber are related by having the same direction. So $\text{incl} \circ d$ returns a representative point from the fiber containing its input. This is a common occurring pattern, as we will see next.

Puzzle 116. Suppose that $r, s: \mathcal{A} \rightarrow \mathcal{A}$ with

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{s} & \mathcal{A} \\ & \searrow \text{id} & \downarrow r \\ & & \mathcal{A} \end{array}$$

so that r is a post-inverse of s and s is a pre-inverse of r . Define $h = s \circ r$. Then $h \circ h = h$, i.e., $h(h(a)) = h(a)$ for any $a \in \mathcal{A}$. (For a concrete case, consider $\text{incl} \circ d$ from the Example 13.9.)

Explain why this is true. Use the associativity of composition, or even easier, use the commutative diagram

$$\begin{array}{ccccc} & & \text{hoh} & & \\ & \curvearrowright & & \curvearrowleft & \\ \mathcal{A} & & \mathcal{A} & \xrightarrow{s} & \mathcal{A} \\ \downarrow r & \nearrow \text{id} & \uparrow r & & \\ \mathcal{A} & \xrightarrow{s} & \mathcal{A} & & \end{array}$$

Functions with the property of h that applying it multiple times does not change the result are called **idempotent**.

The general pattern

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{s} & \mathcal{C} \\ & \searrow \text{id} & \downarrow r \\ & & \mathcal{A} \end{array} \quad (13.15)$$

is rich with interpretation. Here, r is a post-inverse of s and s is a pre-inverse of r . If $a \neq a' \in \mathcal{A}$, then $s(a) \neq s(a')$ because $r(s(a)) = a \neq a' = r(s(a'))$. So, s never maps distinct points to the same value. And for any $a \in \mathcal{A}$, $r(s(a)) = a$, so there is a $c = s(a) \in \mathcal{C}$ with $r(c) = a$.

The function r specifies a way of stacking/grouping elements of $\mathcal{C}$, assigning a “kind” $r(c)$ to each $c \in \mathcal{C}$. The left panel of Figure 13.11 shows $\mathcal{C}$ arranged in stacks⁵ over the different “kinds” in $\mathcal{A}$, with all elements in a stack having the same “kind” as produced by r . The right panel of Figure 13.11 depicts a function s that is a pre-inverse of r . We think of s as a “cross-section” of $\mathcal{C}$, assigning a single “representative” for each “kind,” one element in each stack, as shown. That diagram (13.15) commutes tells us that for any kind $a \in \mathcal{A}$, its representative $s(a)$ has kind a , i.e., $r(s(a)) = a$.

⁵Often called *fibers*, which may seem more apt in Example 13.9.

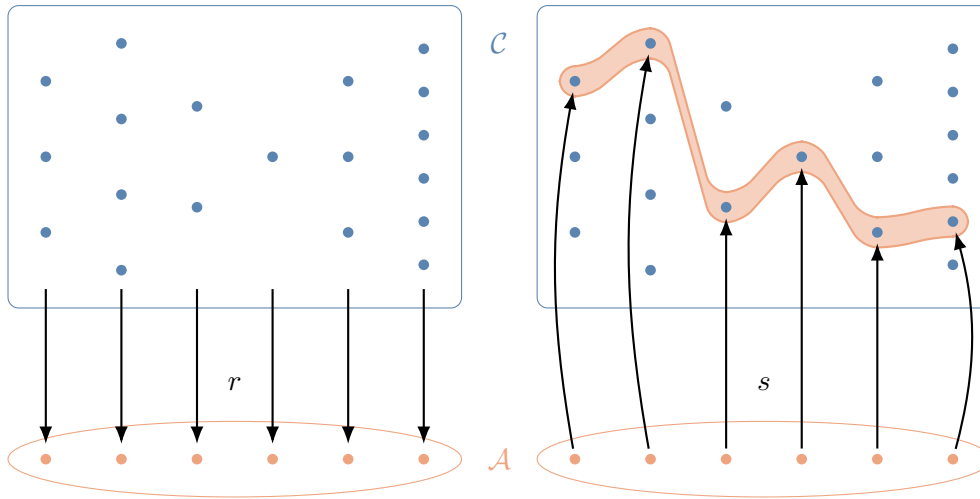


FIGURE 13.11. (Left) The function r that classifies each element of $\mathcal{C}$ with an element of $\mathcal{A}$; r maps elements of $\mathcal{C}$ in a vertical stack to the same “kind.” (Right) The function s associates a “representative” in $\mathcal{C}$ to each “kind” in $\mathcal{A}$.

Example 13.9 gives a concrete illustration of these ideas. Here, $\mathcal{C} = \mathbb{R}^2 - \{(0, 0)\}$ is organized by stacking points together that are on the same *ray* emanating from the origin, and the function d (playing the role of r in this discussion) maps each point to the direction of the ray to which it belongs, with directions represented by

points on the unit circle $\mathcal{A} = \mathbb{S}^1$. A cross-section s – a pre-inverse of d – is a function $\mathbb{S}^1 \rightarrow \mathbb{R}^2 - \{(0,0)\}$ that selects one point on each ray. The condition that diagram (13.15) commutes tells us that s always chooses a point on the same ray as its input.

For another concrete example, let $\mathcal{C}$ be the collection of books held in a library, and $\mathcal{A}$ be the set of authors of books in the collection.⁶ Imagine organizing the library's collection into stacks of books, where the books in each stack have the same author. This is the function r . If we choose one book from each stack, we get the function s ; for each different selection of representatives, we get a different function.

⁶More realistically, a book may have multiple writers, so each element of $\mathcal{A}$ – each “author” – is itself a unique set of writers' names.

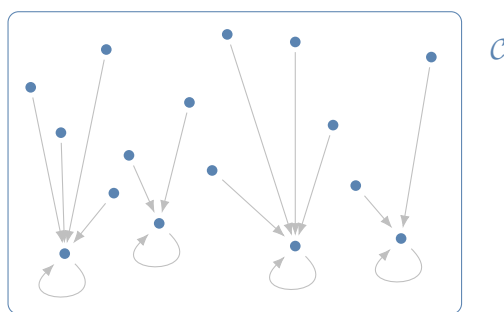
Now, let $\mathcal{P}$ be the set of library patrons, and consider the function mapping each book to the patron that most recently checked out the book. Here, we've stacked all the books in the collection so that all books in the same stack were most recently checked out by the same patron. This gives us a function (analogous to r) *only if* every book in the library has been checked out by *someone*. Assuming that is so, we can select a representative book from each stack – but *only if* every stack is non-empty. That is, we can only find a cross-section (analogous to s) if every patron has checked out at least one book; if a patron never has, we cannot find a representative book for that patron. Thus, diagram (13.15) entails strong constraints on r and s that allow us to interpret them as we have seen.

Puzzle 117. Draw three visually distinctive cross-sections associated with d in Example 13.9. Remember that these functions do not have to be “smooth;” they merely need to choose one point on each ray.

Suppose that $f: \mathcal{C} \rightarrow \mathcal{C}$ is *idempotent* in the sense described in Puzzle 116, i.e., $f \circ f = f$. Then, we can “split” f to find functions r and s as in diagram (13.15) where $f = s \circ r$. Because f is idempotent, if $x = f(c)$ for some c , then $f(x) = f(f(c)) = f(c) = x$, meaning that x is a **fixed point** of f , i.e., it is unchanged by f , $f(x) = x$. So every point x in the image of f is a fixed point. Conversely, if x is a fixed point of f , $x = f(x)$, and x is consequently in the image of f . Figure 13.12 shows a schematic of the action of f on elements of $\mathcal{C}$.

We can construct the set $\mathcal{A}$ in several ways. Most evocatively, we can let elements of $\mathcal{A}$ be the “clusters,” meaning the sets of elements that f maps to the same fixed point: $\mathcal{A} = \{f^{-1}(x) \mid x \in \mathcal{C} \text{ with } f(x) = x\}$. Then r maps each point in $\mathcal{C}$ to its cluster, and s maps each cluster to its corresponding fixed point, yielding diagram (13.15) and $f = s \circ r$. More prosaically, we can let $\mathcal{A} \subseteq \mathcal{C}$ be the set of fixed points of f ; then r equals f on $\mathcal{C}$ and s is the inclusion $\mathcal{A} \hookrightarrow \mathcal{C}$.

As a concrete example, let $\mathcal{C}$ be a set of dinosaurs. Dinosaurs lived in three distinct

FIGURE 13.12. The action of an idempotent function on a set $\mathcal{C}$.

time periods (the Triassic, Jurassic, and Cretaceous) and (loosely speaking) fell into one of two anatomical groups (Saurischian, or “lizard hipped,” and Ornithischian, or “bird hipped”). Suppose the idempotent $f: \mathcal{C} \rightarrow \mathcal{C}$ always returns a dinosaur that lived in the same period and with the same group as its input dinosaur and that it has one fixed point for each period-group pair.⁷ Then, take $\mathcal{A}$ to be the set containing the six period-group pairs. We can split f into an r that maps a dinosaur to its period and group and an s that returns a representative for each period-group pair.

⁷For instance, *Tyrannosaurus rex* for Cretaceous-Saurischian, *Triceratops horridus* for Cretaceous-Ornithischian, *Stegosaurus stenops* for Jurassic-Ornithischian, *Brachiosaurus altithorax* for Jurassic-Saurischian, and so forth.

Example 13.10 Under-determined and Over-determined Systems of Equations

We want to solve the following system of linear equations:

$$a + b + c + d + e = 180$$

$$a + 2b + 3c + 4d + 5e = 60$$

$$a - b + c - d + e = 150$$

but the equations are under-determined: there are more unknowns than equations.

Similarly, we want to solve the following system of linear equations

$$a + 2b + 4c = 51$$

$$-a + 2c = 51$$

$$a + b + c = 102$$

$$a - b - 2c = 0$$

$$b = -51$$

but these equations are *over*-determined, with more equations than unknowns.

How can we find “reasonable” assignments of values to these variables that

solve or nearly solve these equations? Neither case has a unique solution.

The first system has many exact solutions. The left-hand sides of the first system comprise a function $u: \mathbb{R}^5 \rightarrow \mathbb{R}^3$, taking five values as input and returning three as output, where the equations are written as $u(a, b, c, d, e) = \langle 180, 60, 150 \rangle$. The function u does not have an inverse, but it does have a *pre-inverse* $u^\dagger: \mathbb{R}^3 \rightarrow \mathbb{R}^5$ where $w = u^\dagger(r, s, t)$ has components

$$\begin{aligned} w_1 &= \frac{23}{30}r - \frac{1}{5}s + \frac{1}{6}t \\ w_2 &= \frac{11}{20}r - \frac{1}{10}s - \frac{1}{4}t \\ w_3 &= \frac{1}{6}r + \frac{1}{6}t \\ w_4 &= -\frac{1}{20}r + \frac{1}{10}s - \frac{1}{4}t \\ w_5 &= -\frac{13}{30}r + \frac{1}{5}s + \frac{1}{6}t. \end{aligned}$$

This satisfies $u \circ u^\dagger = \text{id}$, and so we can set $\langle a, b, c, d, e \rangle = u^\dagger(180, 60, 150)$ to solve the equations. This solution is the “smallest” among all possible solutions in that it minimizes $a^2 + b^2 + c^2 + d^2 + e^2$.

The second system need not have an exact solution, but we can approximate a solutions to some degree. The left-hand sides of the second system comprise a function $v: \mathbb{R}^5 \rightarrow \mathbb{R}^3$, and the equation can be written as $v(a, b, c) = \langle 51, 51, 102, 0, -51 \rangle$. The function v does not have an inverse, but it does have a *post-inverse* $v^\dagger: \mathbb{R}^3 \rightarrow \mathbb{R}^5$ where $z = v^\dagger(p, q, r, s, t)$ has components

$$\begin{aligned} z_1 &= \frac{4}{17}p - \frac{8}{51}q + \frac{10}{51}r + \frac{7}{17}s - \frac{13}{51}t \\ z_2 &= -\frac{1}{17}p - \frac{5}{17}q + \frac{2}{17}r - \frac{6}{17}s + \frac{11}{17}t \\ z_3 &= \frac{3}{17}p + \frac{11}{51}q - \frac{1}{51}r + \frac{1}{17}s - \frac{14}{51}t. \end{aligned}$$

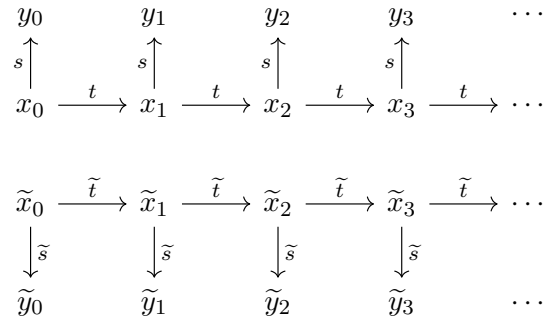
This satisfies $v^\dagger \circ v = \text{id}$, so we set $\langle a, b, c \rangle = v^\dagger(51, 51, 102, 0, -51)$. If the solution exists, we get it from the post-inverse equation, but in general we get an approximate solution that makes $v(a, b, c)$ as close as possible to $\langle 51, 51, 102, 0, -51 \rangle$.

These pre- and post-inverses can be found for any system of linear equations; we use this for “least squares regression” in Chapter 6.

13.4 Describing Structure

One of the most important uses for functions is to express relationships between different information structures that we build. For instance, in Example 13.2, we built machines that track their internal state and emit signals. Such a machine is specified with a choice of state space $\mathcal{S}$, signal space $\mathcal{Y}$, output function s , transition function t , and initial state x_0 . If we had two such machines $A = \langle \mathcal{S}, \mathcal{Y}, s, t, x_0 \rangle$ and $\tilde{A} = \langle \tilde{\mathcal{S}}, \tilde{\mathcal{Y}}, \tilde{s}, \tilde{t}, \tilde{x}_0 \rangle$, what would we need to be able to *simulate* the machine $\tilde{A}$ using A ?

Consider the sequences of states and emitted signals from A and $\tilde{A}$:



where we use values in lieu of singleton sets in the diagrams. We could simulate $\tilde{A}$ using A if we had a *translator* that could map states and signals from A to states and signals from $\tilde{A}$ in a way that is consistent with the workings of each machine. Such a translator is just a *pair of functions* $\langle \psi, \sigma \rangle$ where $\psi: \mathcal{S} \rightarrow \tilde{\mathcal{S}}$ translates states and $\sigma: \mathcal{Y} \rightarrow \tilde{\mathcal{Y}}$ translates signals and such that for every $n \in [1..)$, the following diagram commutes at the n th button push:

$$\begin{array}{ccc}
 y_n & & y_{n+1} \\
 \uparrow s & & \uparrow s \\
 x_n & \xrightarrow{t} & x_{n+1} \\
 \downarrow \psi & & \downarrow \psi \\
 \tilde{x}_n & \xrightarrow{\tilde{t}} & \tilde{x}_{n+1} \\
 \downarrow \tilde{s} & & \downarrow \tilde{s} \\
 \tilde{y}_n & & \tilde{y}_{n+1}
 \end{array}
 \quad \sigma \quad (13.16)$$

In words, the middle square (of x 's and $\tilde{x}$'s) says: we get the *same state* if we first translate the state of A with ψ and then transition in $\tilde{A}$, *or* if we first transition in

A and then translate with ψ . Similarly, the left and right “squares” (with y ’s and $\tilde{y}$ ’s) says: we get the *same signal* if we first emit the signal from a state of A then translate it with σ or if we first translate that state to $\tilde{A}$ with ψ and then emit the signal. In equational terms, (13.16) is written

$$\sigma \circ s = \tilde{s} \circ \psi \quad (13.17)$$

$$\psi \circ t = \tilde{t} \circ \psi. \quad (13.18)$$

We call the transformation $\langle \psi, \sigma \rangle$ a **homomorphism** between these two machines. The word comes from “homo” and “morphism” for “same form”, or “same structure,” and in general, a homomorphism is a structure preserving function. The homomorphism $\langle \psi, \sigma \rangle$ maps from one machine to another while preserving the machine’s dynamics.

Puzzle 118. Suppose you have a machine A that runs *two* identical traffic lights. Find a homomorphism $\langle \psi, \sigma \rangle$ from A to an automaton $\tilde{A}$ that runs one traffic light of the same type. Clearly describe the state spaces and signal spaces of both machines.

The next example illustrates this idea with simple graphs – a structure we use to capture relationships – and shows some applications. See Chapter 17 for a deeper discussion of this theme.

Example 13.11 Transforming Simple Graphs

This example builds on Example 10.19; see also Example 10.18.

If we have two graphs $G = \langle \mathcal{N}_G, \mathcal{E}_G, \psi_G \rangle$ and $H = \langle \mathcal{N}_H, \mathcal{E}_H, \psi_H \rangle$, then a **graph homomorphism** is a mapping from G to H that respects the relationships in the graph. Specifically, a graph homomorphism is a pair of functions $\langle h_0, h_1 \rangle$ with $h_0: \mathcal{N}_G \rightarrow \mathcal{N}_H$ and $h_1: \mathcal{E}_G \rightarrow \mathcal{E}_H$ where *an edge in G and its incident nodes must map to an edge in H and its incident nodes*.

Let’s spell this definition out more carefully. In this example, we will consider only simple, undirected graphs without loops. So, we can describe an edge with a set of two nodes, meaning $\psi_G: \mathcal{E}_G \rightarrow \binom{\mathcal{N}_G}{2}$ and $\psi_H: \mathcal{E}_H \rightarrow \binom{\mathcal{N}_H}{2}$. To ensure at most one edge between any pair of nodes, we require that both ψ_G and ψ_H map distinct edges to distinct sets.

Then, $\langle h_0, h_1 \rangle$ is a graph homomorphism if mapping an edge with h_1 then describing it is the same as describing an edge then mapping its incident nodes

with $h_0^\rightarrow$, where $h_0^\rightarrow(\{n, m\}) = \{h_0(n), h_0(m)\}$ is the image of h_0 (see page 581). This is easiest to see with a commutative diagram:

$$\begin{array}{ccc}
 \mathcal{E}_G & \xrightarrow{h_1} & \mathcal{E}_H \\
 \downarrow \psi_G & & \downarrow \psi_H \\
 \binom{\mathcal{N}_G}{2} & \xrightarrow{h_0^\rightarrow} & \binom{\mathcal{N}_H}{2}
 \end{array} \tag{13.19}$$

There are two distinct paths from $\mathcal{E}_G$ to $\binom{\mathcal{N}_H}{2}$. We can describe an edge e of G (with ψ_G) to get the set of two nodes $\{n_1, n_2\}$ of G incident to e and then map that to a set of nodes in H (with $h_0^\rightarrow$) to get $\{h_0(n_1), h_0(n_2)\}$. Or: we can map e to an edge $h_1(e)$ of H and describe that edge (with ψ_H) to get a set of nodes in H $\{m_1, m_2\}$. The diagram tells us that these two sets must be the same: the nodes incident to e must map to the nodes incident to $h_1(e)$. That is, the mapping of edges and nodes from G to H must be consistent. In equational terms, Diagram 13.19 means $\psi_H \circ h_1 = h_0^\rightarrow \circ \psi_G$.

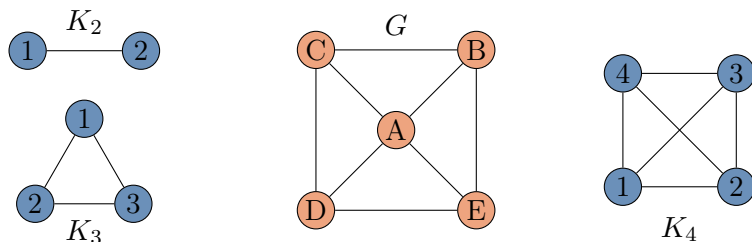
(Note on the diagram: as we will discuss in Chapter 14, the tails on the arrows for ψ_G and ψ_H in the diagram indicate our constraints on these functions: that for each ψ , $e_1 \neq e_2 \in \mathcal{E}$ implies $\psi(e_1) \neq \psi(e_2)$.)

The graph homomorphism condition implies that if we know where the nodes map, we can determine where the edges map, and vice versa. Multiple nodes can map to the same node, and multiple edges to the same edge, as long as the nodes and edges are assigned consistently. Edges cannot be destroyed, however, and because we have excluded loops here, the nodes incident to an edge must map to different nodes. (With loops, we could join nodes and turn their edge, if any, into a loop)

Next, let's delve into the implications of this definition by looking at some concrete graph homomorphisms. Consider the graphs G and K_2 , K_3 , K_4 below. The graph K_c has c -nodes, and *every pair* of nodes is connected by an edge; these are called the **complete graph** on c nodes. A graph homomorphism gives a picture of one graph inside the other, allowing for folding of the input graph in a way that matches nodes and edges consistently.

For instance, every homomorphism from K_2 to G must map nodes 1 and 2 to a connected pair of nodes in G (and map the 1-2 edge to the corresponding edge in G). And for every connected pair of nodes in G there is such a homomorphism. Notice, however, that there are **no** graph homomorphisms from G to K_2 . If node

A maps to node 1, say, then nodes B–E must map to 2 because their edges with A must map to the unique edge in K_2 . But then the edge from B to C cannot map to an edge in K_2 . The same happens if node A maps to 2.



Every homomorphism from K_3 to G must map K_3 to a connected triangle in G . There are four such triangles (the small triangles meeting at node 1) and three ways to map K_3 onto each of them. No other node assignment allows consistent edge assignment, so no other homomorphisms exist. In this case, there are exactly six homomorphisms from G to K_3 , each of which folds the four small triangles in G onto the triangle K_3 . Once node A is assigned, then nodes B and D must map to the same node, and nodes C and E must map to the same node, to allow the edges to be mapped consistently. When $A \mapsto 1$, we have either $B, D \mapsto 2; C, E \mapsto 3$ or $B, D \mapsto 3; C, E \mapsto 2$, and similarly for the other assignments of A.

There are no homomorphisms from K_4 to G because any assignment of nodes 1–4 leaves unassigned or inconsistently assigned edges. Put simply: we see no subgraph like K_4 in G . However, homomorphism from G to K_4 *do* exist; for instance, $D \mapsto 1; A \mapsto 2; B \mapsto 3; C, E \mapsto 4$. Try checking this.

As a first application, we consider the problem of graph coloring. For $c \in [1..]$, a **c -coloring** of a graph is an assignment to each node of G one of c colors such that *no two nodes joined by an edge have the same color*. The famous Four Color Theorem says that any simple, loopless, graph that can be drawn on a piece of paper without edges crossing (except at the nodes) can be 4-colored.

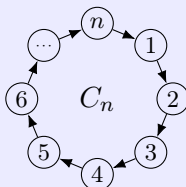
Notice that a c -coloring of a graph G is just a *graph homomorphism from G to K_c* ! Each node in K_c corresponds to a color, so the node mapping (h_0) returns the “color” of each node in G . The homomorphism condition ensures that the nodes connected by an edge map to different nodes, and thus different colors. Conversely, given any c -coloring of G , we map nodes of G to nodes of K_c with nodes connected by an edge mapped to different colors; since K_c is

A valid schedule is a graph homomorphism from C to X , and vice versa. The node mapping (h_0) indicates which exam period a course is assigned to, and any edges from that node must map to edges in X , meaning that students in both courses can take exams in both periods. The following table shows one such homomorphism.

$C \mapsto X$		$C \mapsto X$		$C \mapsto X$		$C \mapsto X$	
A	3	E	3	I	5	M	4
B	6	F	1	J	3	N	4
C	1	G	2	K	5		
D	5	H	6	L	6		

Puzzle 119. Show how to define a homomorphism of simple *directed* graphs. Use Diagram 13.19 as a reference/starting point.

How might you use the existence of such a homomorphism to determine if a simple directed graph contains a *cycle* (a path from a node back to itself following the directed edges)? (The directed graphs C_n below might prove useful here.)



Function Properties

14

Chapter

Contents

14.1 Input-Output Correspondence	654
14.2 Analytical Constraints	665

Up to this point, we have focused on the various ways we define and use functions. In this section, we look more closely at several important properties functions can have and what those properties give us.

14.1 Input-Output Correspondence

Looking again at Figure 8.5 on page 537, we see some immediate differences among the functions. For both functions on the left of the Figure, every input maps to a *different output*. For both functions on the top of the Figure, every possible output is returned for *some input*. The function on the top left has both features, and the function on the bottom right has neither. These simple functions exemplify fundamental properties about how a functions' inputs and outputs are connected. The bottom left function is an *injection*; the top right function is a *surjection*; and the top left function is a *bijection*.

If $f: \mathcal{A} \rightarrow \mathcal{B}$, we say that

- f is an **injection** if $a, a' \in \mathcal{A}$ with $a \neq a'$ implies that $f(a) \neq f(a')$ and we write its type as $f: \mathcal{A} \rightarrowtail \mathcal{B}$;
- f is an **surjection** if for every $b \in \mathcal{B}$, there is an $a \in \mathcal{A}$ with $f(a) = b$ and we write its type as $f: \mathcal{A} \twoheadrightarrow \mathcal{B}$;
- f is a **bijection** if it is *both* an injection and a surjection and we write its type as $f: \mathcal{A} \xrightarrow{\sim} \mathcal{B}$.

To help keep these terms and arrows straight, so to speak, pay attention to the prefix. An **in**jection injects a copy of $\mathcal{A}$ *into* $\mathcal{B}$, and the condition starts with two unequal *inputs*, which is depicted on the tail (input end) of the arrow $\rightarrowtail$. A **sur**jection – “sur” is French for “on” – pastes $\mathcal{A}$ *onto* $\mathcal{B}$, and the condition requires that every output is covered, which is depicted on the head (output end) of the arrow $\twoheadrightarrow$. A **bi**jection – “bi” for “two” – has *both* properties and satisfies both conditions, which is depicted on both ends of the arrow $\xrightarrow{\sim}$. These distinctions will come further into focus as we see examples below.

INJECTIONS. A function is an injection if it *maps different inputs to different outputs*. It is equivalent to say that f is an injection when $f(x) = f(y)$ requires that $x = y$. If we see the returned output of an injection, only one possible input can

have produced it. An injection $f: \mathcal{A} \rightarrow \mathcal{B}$ therefore has a post-inverse r

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\ & \searrow \text{id} & \downarrow r \\ & & \mathcal{A} \end{array}$$

that gives the input of f associated with any returned output. Formally, a post-inverse r is defined so that when $b \in f^{-1}(\mathcal{A}) \subseteq \mathcal{B}$ satisfies $b = f(a)$, $r(b) = a$ and otherwise, $r(b) = a_0$ for an arbitrarily chosen $a_0 \in \mathcal{A}$.¹ Notice that r must be a surjection here.

For example, the function $\ell: \mathbb{R} \rightarrow \mathbb{R}$ defined by $\ell(x) = \frac{e^x}{1+e^x}$ is an injection. An informal way to see this is to graph the function and observe that every horizontal line intersects the graph *at most once*. More directly, $\text{range}(\ell)$ is the interval $(0, 1)$ because $e^x > 0$ and $e^x < 1 + e^x$. If $\ell(x) = \ell(y)$ then

$$\begin{aligned} \frac{e^x}{1+e^x} &= \frac{e^y}{1+e^y} \\ e^x(1+e^y) &= e^y(1+e^x) \\ e^x + e^x e^y &= e^y + e^x e^y \\ e^x &= e^y \\ x &= y, \end{aligned}$$

using the invertibility of the exponential function described earlier. A post-inverse for ℓ is a function r with $r(p) = \ln(p/(1-p))$ when $p \in (0, 1)$ and $r(p) = 0$ otherwise.²

$$\begin{aligned} (r \circ \ell)(x) &= \ln\left(\frac{\frac{e^x}{1+e^x}}{1 - \frac{e^x}{1+e^x}}\right) \\ &= \ln\left(\frac{\frac{e^x}{1+e^x}}{\frac{1}{1+e^x}}\right) \\ &= \ln(e^x) \\ &= x. \end{aligned}$$

If we *restrict*³ r and ℓ to functions $\tilde{\ell}: \mathbb{R} \rightarrow (0, 1)$ and $\tilde{r}: (0, 1) \rightarrow \mathbb{R}$, then $\tilde{\ell}$ and $\tilde{r}$ are inverses of each other.

As another example, recall that in Examples 10.18, 10.19, and 13.11, we saw that the data needed to specify a Graph, directed or undirected, include a function that maps an edge identifier to a description of the edges incident nodes. A simple graph is one for which this function is an injection; that condition enforces the idea that

¹If there is no b outside $f^{-1}(\mathcal{A})$, then f is also a surjection and therefore is a bijection. See below.

²The returned value 0 is arbitrary here. We get a different post-inverse for each choice of this value.

³See "Restriction" on page 583.

there is at most one edge between any pair of nodes.

In contrast, the functions $g(u) = u^2$ and $h(t) = \sin(t)$ are not injections. In the first case, $g(1) = g(-1)$ and yet $1 \neq -1$, and in the second case, $h(t + 2\pi k) = h(t)$ for all integers k , giving many inputs that map to the same output.

We can think of an injection from $\mathcal{A}$ into $\mathcal{B}$ as a way of *labeling* some elements of $\mathcal{B}$ with a distinct value of $\mathcal{A}$. For instance, if $n \in [1..n]$, an injection $[1..n] \rightarrow \mathcal{B}$ labels n *distinct* elements of $\mathcal{B}$. Such an injection specifies one way of selecting a subset of $\mathcal{B}$ of cardinality n ; that is for $f: [1..n] \rightarrow \mathcal{B}$, $f^* \in \binom{\mathcal{B}}{n}$. So injections $\{1\} \rightarrow \mathcal{B}$ select single elements, injections $\{1, 2\} \rightarrow \mathcal{B}$ select two distinct elements, injections $\{1, 2, 3\} \rightarrow \mathcal{B}$ select three distinct elements, and so on.⁴

⁴Chapter 18 shows how to count the injections from $[1..n]$ to a set, and thus find $\#\binom{\mathcal{B}}{n}$.

Puzzle 120. Is id an injection? When is const_c an injection?

If $f: \mathcal{A} \rightarrow \mathcal{B}$ is an injection, for any pair of functions $h, g: \mathcal{X} \rightarrow \mathcal{A}$ on some set $\mathcal{X}$, $f \circ h = f \circ g$ implies that $h = g$. This follows from the existence of a post-inverse r of f because $h = r \circ f \circ h = r \circ f \circ g = g$. To see this more directly, suppose $f \circ h = f \circ g$ but $h \neq g$, then there is a point $x \in \mathcal{X}$ for which $h(x) \neq g(x)$ but $f(h(x)) = f(g(x))$, which contradicting the condition of an injection.

Conversely, if for any pair of functions $h, g: \mathcal{X} \rightarrow \mathcal{A}$ on some set $\mathcal{X}$, $f \circ h = f \circ g$ implies that $h = g$, then f must be an injection. Let $\mathcal{X} = \mathbb{U}$, a set with one element; then h and g just pick out elements a and a' of $\mathcal{A}$, and the condition becomes: $f(a) = f(a')$ implies $a = a'$, which makes f an injection. So if we can always “cancel” a function by post-composition, it is an injection. Thus any function with a post-inverse is an injection.

More generally, if $h \circ f$ is an injection, then f must be as well. Otherwise, we have $x \neq y$ with $f(x) = f(y)$ and hence $h(f(x)) = h(f(y))$.

Summary of Injections.

- The following statements are all equivalent:
 1. $f: \mathcal{A} \rightarrow \mathcal{B}$ is a **injection**.
 2. $a \neq a'$ implies $f(a) \neq f(a')$.
 3. $f(a) = f(a')$ implies $a = a'$.
 4. For $h, g: \mathcal{X} \rightarrow \mathcal{A}$ on a set $\mathcal{X}$, $f \circ h = f \circ g$ implies $h = g$.
- Every injection has a post-inverse, and every function with a post-inverse is an injection.

- If $h \circ f$ is an injection, so is f .
- If f and g are injections, so is $f \circ g$.

Puzzle 121. Give an argument to convince you that the last statement is true.

SURJECTIONS. A surjection is a function for which *every possible output is an actual output for some input*; its range equals its codomain. Because every output has at least one associated input, a surjection $f: \mathcal{A} \twoheadrightarrow \mathcal{B}$ has a *pre-inverse* s

$$\begin{array}{ccc} \mathcal{B} & \xrightarrow{s} & \mathcal{A} \\ & \searrow \text{id} & \downarrow f \\ & & \mathcal{B} \end{array}$$

that selects one associated input for each output. Formally, a post-inverse s is defined by setting $s(b)$ to any element of $f^{-}(\{b\})$. Notice that s must be an injection here.

For example, the function $q: \mathbb{R} \rightarrow \mathbb{R}$ given by $q(t) = t \cos(2\pi t)$ is a surjection. An informal way to see this is to graph the function and observe that every horizontal line intersects the graph *at least once*. More directly, $q(k) = k$ for every integer k , and because q is continuous,⁵ q takes on every intermediate value between k and $k + 1$ for all integers k . We can define a pre-inverse s for q by taking $s(k) = k$ for every integer and then for each y pick one of the input values for which the graph intersects the horizontal line at y .

⁵See “Smooth Functions” on page 667.

As another example, the functions **sum** in Example 13.3 and d in Example 13.9 are both surjections. In contrast, the functions $h, g: \mathbb{R} \rightarrow \mathbb{R}$ given by $h(x) = x^2$ and $g(t) = \sin(t)$ are not surjections. If we restricted these functions to smaller codomains, they would be.

Whereas injections *into* $\mathcal{A}$ label subsets of the codomain, we can think of surjections *out of* $\mathcal{A}$ as sorting each of $\mathcal{A}$ ’s elements into one or more mutually exclusive categories. If $f: \mathcal{A} \twoheadrightarrow \mathcal{C}$, then for each $c \in \mathcal{C}$, then $f^{-}(\{c\})$ is the set of $\mathcal{A}$ ’s element in category c . Because f is a surjection, $f^{-}(\{c\}) \neq \{\}$, so every category has a representative. The discussion of the diagram (13.15) describes this in more detail.

For a function $f: \mathcal{A} \rightarrow \mathcal{B}$, we can ask what equations $f(a) = b$ can be “solved” for a . If f is an injection, then not every b admits a solution, but if it does, that solution is unique. If f is a surjection, then every b admits a solution, but it need not be unique. If f is a surjection *and* an injection – meaning that f is a bijection – then every b admits a unique solution.

If $f: \mathcal{A} \twoheadrightarrow \mathcal{B}$, then for any $g, h: \mathcal{B} \rightarrow \mathcal{Y}$ with some set $\mathcal{Y}$, $h \circ f = g \circ f$ implies that $h = g$. To see this, compose both sides of the first equation with f 's pre-inverse (call it s): $h = h \circ f \circ s = g \circ f \circ s = g$.

Conversely, if for any $g, h: \mathcal{B} \rightarrow \mathcal{Y}$ with some set $\mathcal{Y}$, $h \circ f = g \circ f$ implies that $h = g$, then f must be a surjection. To see this, observe that for any $b \in \mathcal{B}$, there is an $a \in \mathcal{A}$ with $f(a) = b$. Thus, $h(b) = h(f(a)) = g(f(a)) = g(b)$, so $h = g$. So if we can always “cancel” a function with pre-composition, it is a surjection. Thus, any function with a pre-inverse is a surjection.

Summary of Surjections.

- The following statements are all equivalent:
 1. $f: \mathcal{A} \rightarrow \mathcal{B}$ is an **surjection**.
 2. For each $b \in \mathcal{B}$, there is an $a \in \mathcal{A}$ with $f(a) = b$.
 3. For $h, g: \mathcal{B} \rightarrow \mathcal{Y}$ for a set $\mathcal{Y}$, $h \circ f = g \circ h$ implies $h = g$.
- Every surjection has a pre-inverse, and every function with a pre-inverse is a surjection.
- If $f \circ g$ is a surjection, so is f .
- If f and g are surjections, so is $f \circ g$.

Puzzle 122. Give an argument or draw a picture to convince yourself of either (or both!) of the last two statements.

Example 14.1 Set Partitions

If $\mathcal{S}$ is a non-empty set, then a **partition** of $\mathcal{S}$ is a way of dividing $\mathcal{S}$ into mutually exclusive and collectively exhaustive parts, a family of subsets over some index set $\mathcal{T}$, $\mathcal{S}_t \subseteq \mathcal{S}$ for $t \in \mathcal{T}$, where $\bigcup_{t \in \mathcal{T}} \mathcal{S}_t = \mathcal{S}$ and $\mathcal{S}_t \cap \mathcal{S}_{t'} = \{\}$ when $t \neq t'$. Each $\mathcal{S}_t$ is called a *part* of the partition, with $\mathcal{S}_t$ the t^{th} part. For instance, $\{1\}, \{2, 3\}, \{4\}$ is a partition of $[1..4]$.

We can represent partitions by surjections. Every surjection $p: \mathcal{S} \twoheadrightarrow \mathcal{T}$ yields a partition of $\mathcal{S}$ with $\mathcal{S}_t = p^{-1}(\{t\})$. And every partition of $\mathcal{S}$ can be associated with a surjection on $\mathcal{S}$ that maps $s \in \mathcal{S}$ to its part.

We can put an order on partitions of $\mathcal{S}$. Given two partitions of $\mathcal{S}$, call them

P and Q , we say that P is *finer* than Q (and conversely that Q is *coarser* than P) if every part of P belongs to some part of Q . We write $P \prec Q$ to denote this.

If we represent partitions by surjections $p: \mathcal{S} \twoheadrightarrow \mathcal{T}$ and $q: \mathcal{S} \twoheadrightarrow \mathcal{U}$, we can see this ordering as well. We have that p is finer than q , $p \prec q$, if there is a $f: \mathcal{T} \rightarrow \mathcal{U}$ such that

$$\begin{array}{ccc} \mathcal{S} & \xrightarrow{p} & \mathcal{T} \\ & \searrow q & \downarrow f \\ & & \mathcal{U} \end{array}$$

The function f indicates to which part of q each part of p belongs. We say that q “factors” through p and f .

Puzzle 123. For any non-empty set $\mathcal{S}$, there is a coarsest possible partition and a finest possible partition. What are these partitions? And what are the surjections that give rise to them? What is the mapping between the codomains of these surjections that makes the previous diagram hold?

Construct an example with a partition intermediate between the finest and coarsest and answer the same questions.

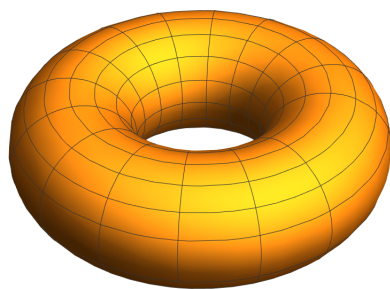
Puzzle 124. Suppose that we have a partition of the set $\mathcal{X}$ represented by a surjection $p: \mathcal{X} \twoheadrightarrow \mathcal{I}$ for some set $\mathcal{I}$. We have a surjection $f: \mathcal{X} \twoheadrightarrow \mathcal{Y}$ for a set $\mathcal{Y}$. Explain how we can use p and f to create a partition of $\mathcal{Y}$.

Give concrete instance of this using the set $\mathcal{X} = [1..7]$.

Example 14.2 Gluing Spaces

The unit circle $\mathbb{S}^1 \subseteq \mathbb{R}^2$ is the set $\{\langle x, y \rangle \mid x^2 + y^2 = 1\}$. Consider the surjection $g: [0..2\pi] \twoheadrightarrow \mathbb{S}^1$ given by $g(t) = \langle \cos(t), \sin(t) \rangle$. This function *identifies* the two endpoints of the interval; we can think of this as gluing together the endpoints to make the circle.

We can do the same in constructing spaces. For instance, the torus



is a surface of revolution of a circle around a bigger circle and can be expressed as a surface in $\mathcal{T} \subseteq \mathbb{R}^3$ or as $\mathbb{S}^1 \times \mathbb{S}^1$. (Here, the outer radius is twice the inner radius.) We can create a torus from a square (with a bit of stretching) by gluing together opposite edges:



This gluing is described by a surjection $h: [0\_1] \times [0\_1] \twoheadrightarrow \mathcal{T}$ defined by $h(u, w) = \langle \cos(2\pi u)(2 + \cos(2\pi w)), \sin(2\pi u)(2 + \cos(2\pi w)), \sin(2\pi w) \rangle$.

BIJECTIONS. A bijection *matches every possible output with exactly one input*; it specifies a one-to-one correspondence between the elements of two sets. Because a bijection is both an injection and a surjection, it has both post-inverse and a pre-inverse by the arguments above. Hence, every bijection is invertible, and the inverse function of a simply reverses the one-to-one correspondence. In fact, we can say more. An invertible function has a pre-inverse – and is thus a surjection – and a post-inverse – and is thus an injection. So a function is a invertible if and only if it is a bijection.

Summary of Bijections.

- The following statements are all equivalent:
 1. $f: \mathcal{A} \rightarrow \mathcal{B}$ is an **bijection**.
 2. f is an injection and a surjection.
 3. f is invertible.
- If f is a bijection, then so is f^{-1} .

- If $f: \mathcal{A} \rightarrow \mathcal{B}$ and $g: \mathcal{B} \rightarrow \mathcal{C}$ are bijections, then so is $g \circ f$.

Puzzle 125. If $g \circ f$ is a bijection, we can conclude from the properties of injections and surjections established earlier that g is a surjection and f is an injection.

Give an example to show that f and g need *not* be bijections.

Example 14.3. Recall that the “unit set” $\mathbb{U} = \{\langle \rangle\}$ has a single element, the empty tuple. Each function from $\mathbb{U}$ to a set $\mathcal{A}$ simply picks an element of $\mathcal{A}$, the element it returns.

Thus, for any set $\mathcal{A}$, there is a one-to-one correspondence between elements of $\mathcal{A}$ and functions from $\mathbb{U}$ into $\mathcal{A}$. Specifically, const is a bijection $\mathcal{A} \rightarrow (\mathbb{U} \rightarrow \mathcal{A})$ where for each $a \in \mathcal{A}$, $\text{const}_a: \mathbb{U} \rightarrow \mathcal{A}$ is the function that returns a .

This identification means that we can think of any constant $a \in \mathcal{A}$ as a “nullary” function – a function that take no arguments. (See Chapter 15.)

One use of bijections is to give a correspondence between two different representations of an object. In the previous example, we see that we can think about elements of a set as a functions into that set. In the next example, we see an analogue for subsets of a set. The two representations are not identical objects, but for some purposes, we can use whichever of the representations is more convenient.

Example 14.4. In Example 10.12, we defined the *power set* $2^{\mathcal{A}}$ of a set $\mathcal{A}$ as the set of all subsets of $\mathcal{A}$,⁶ and we saw there that there is a bijection between $2^{\mathcal{A}}$ and the set of functions from $\mathcal{A} \rightarrow 2$.

⁶Recall that $2 = \{0, 1\}$.

If $\mathcal{B} \in 2^{\mathcal{A}}$, i.e., a subset of $\mathcal{A}$, define the function

$$i_{\mathcal{B}}(a) = \begin{cases} 1 & \text{if } a \in \mathcal{B} \\ 0 & \text{otherwise.} \end{cases}$$

This is the *indicator of $\mathcal{B}$* ; see Chapter 12. Conversely, if $j: \mathcal{A} \rightarrow 2$, define $\mathcal{B} \in 2^{\mathcal{A}}$ by $\mathcal{B} = j^{-1}(\{1\})$. If for some $\mathcal{B} \in 2^{\mathcal{A}}$, we map $i_{\mathcal{B}}$, we get exactly the set $\mathcal{B}$. And similarly $i_{j^{-1}(\{1\})} = j$. Hence, the functions we get in both directions are inverses of each other and thus bijections.

As we have seen, we sometimes have structure on a set and consider functions (called homomorphisms) that preserves that structure. When we have a bijection that

also preserves structure – and thus so does its inverse – we call it an **isomorphism**.⁷ Consider undirected, simple graphs as in Example 13.11. Graphs $G = \langle \mathcal{N}_G, \mathcal{E}_G, \psi_G \rangle$ and $H = \langle \mathcal{N}_H, \mathcal{E}_H, \psi_H \rangle$ are isomorphic if there is a bijection between the graphs' nodes and a bijection between the graphs' edges that makes diagram (13.19) commute. The idea is that we can convert from one graph to another just by relabeling the nodes and edges; the relationships themselves do not change.

⁷See Chapter 17.

Example 14.5 Currying. If we have a function $f: \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{C}$ that takes two arguments, then for each fixed value $a \in \mathcal{A}$ of the first argument, we can think of $f(a, \blacksquare)$ as a function on $\mathcal{B}$ (where a value from $\mathcal{B}$ fills the hole). Conversely, if we have a family of functions indexed by $\mathcal{A}$, $g_a: \mathcal{B} \rightarrow \mathcal{C}$, we can make a function of two arguments with $f(a, b) = g_a(b)$.

This gives us a bijection **curry** (with inverse function **uncurry**) between the set of functions of two arguments, $\mathcal{A} \times \mathcal{B} \rightarrow \mathcal{C}$, and the set of function families indexed by $\mathcal{A}$, which is $\mathcal{A} \rightarrow (\mathcal{B} \rightarrow \mathcal{C})$. It is defined as follows: if $g = \text{curry}(f)$, then $g_a(b) = f(a, b)$. **uncurry** reverses this.

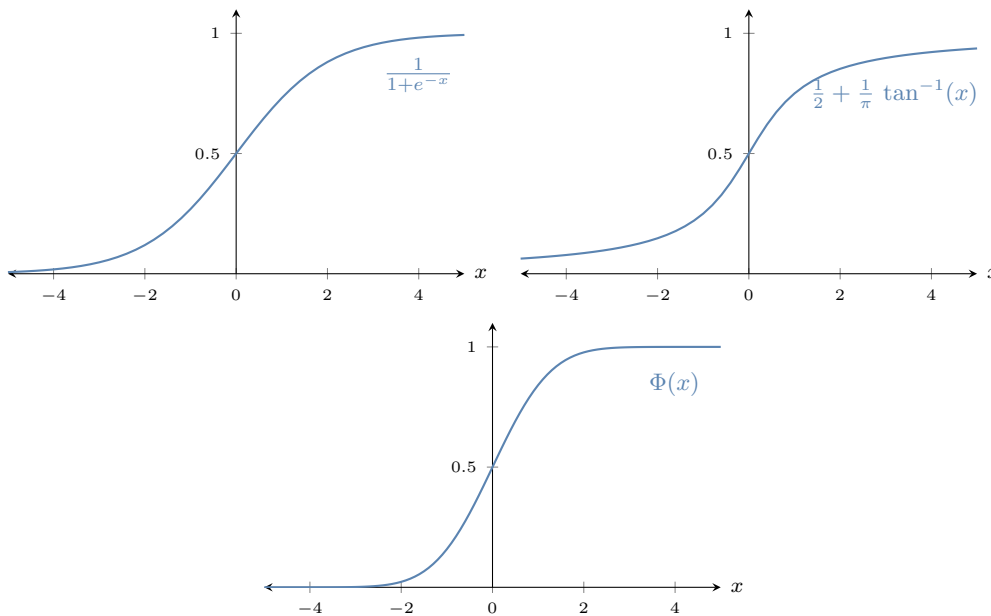


FIGURE 14.1. Three commonly used sigmoidal functions. (Top left) The logistic function; (Top right) arctangent mapped to $(0, 1)$; (Bottom) The Normal CDF Φ

Example 14.6 Sigmoidal Functions

A “sigmoidal” function is an increasing function defined on the real numbers that has a vaguely “S” shape. Figure 14.1 shows three commonly used sigmoidal functions: the logistic function, a scaled arc tangent, and what we will call the Normal CDF Φ in Chapter 1.

All three functions are bijections from $\mathbb{R}$ to the interval $(0,1)$. You can see in the Figures that each horizontal line that hits the vertical axis in $(0,1)$ intersects each graph in only one place. To see the inverse function, reflect the graph about the 45-degree line through the original. Or equivalently, from any point y in the codomain, move horizontally from the vertical axis to the graph, then move vertically to the horizontal axis. The value you end up at is the inverse function at y .

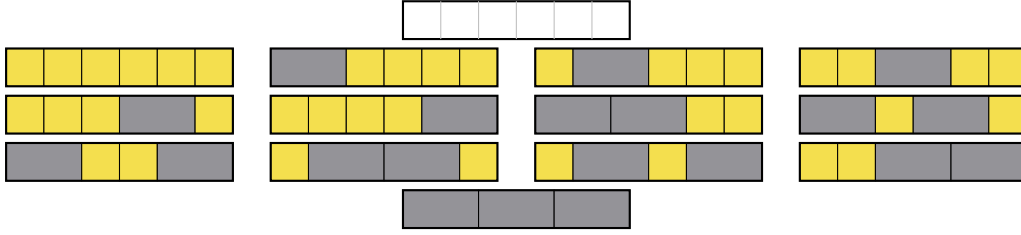
Bijections underlie the process of counting. When we pick up items one by one from a collection and count “one,” “two,” “three,” ..., we are defining a bijection between the set of items in the collection and a set $[1..n]$. Indeed, this characterizes the notion of cardinality.

Counting Principle

Two sets have the same cardinality if and only if there is a bijection between them.

Because a bijection between two sets $\mathcal{A}$ and $\mathcal{B}$ matches each element of $\mathcal{A}$ to exactly one element of $\mathcal{B}$, and vice versa, intuitively, two finite sets $\mathcal{A}$ and $\mathcal{B}$ with a bijection between them have the same number of elements. However, our intuition about “same number” gets a bit stretched when the sets are infinite. For example, let $2\mathbb{Z}$ denote the set of even integers and consider $f: \mathbb{Z} \rightarrow 2\mathbb{Z}$ and $g: 2\mathbb{Z} \rightarrow \mathbb{Z}$ with $f(k) = 2k$ and $g(m) = m/2$. These are inverses of each other and thus give a bijection between the set of *all* integers and *even* integers. The Counting Principle says that $\mathbb{Z}$ and $2\mathbb{Z}$ have the same cardinality – and indeed that any set that can be put in bijection with $\mathbb{Z}$ has the same cardinality. But how can that be when $2\mathbb{Z}$ is a proper subset of $\mathbb{Z}$? The failure here is not with the Counting Principle but with our intuition of the infinite, and the Counting Principle lets us extend the idea of Cardinality beyond the finite.

The next example shows one way we use this principle to ... count. We will discuss this in more detail in Chapter 18.

FIGURE 14.2. An untiled 6-board and all 13 6-board tilings in $\mathcal{T}_6$, as described in Example 14.7.**Example 14.7 Fibonacci Tilings**

An n -board is a row of $n \in \mathbb{N}$ square cells, as in the top panel of Figure 14.2. We can lay tiles on the board from an unlimited collection of two sorts: squares (which cover one cell) and dominoes (which covers two cells). How many distinct tilings of an n -board are there? Figure 14.2 shows all tilings of a 6-board.

Let $\mathcal{T}_n$ be the set of such tilings and let $t_n = \#\mathcal{T}_n$ the number of tilings. If $n = 0$, the board is empty, and the only tiling is the empty tiling, so $t_0 = 1$. If $n = 1$, there is one cell and a single tiling of it with a square, so $t_1 = 1$. To avoid boundary cases, it will be convenient to define $\mathcal{T}_{-1} = \{\}$ and thus $t_{-1} = 0$.

Define the sets $\mathcal{S}_n$ and $\mathcal{D}_n$ to be the sets of tilings of an n -board that begin, respectively, with a square in the first cell and a domino in the first two cells. Notice that $\mathcal{S}_n \cap \mathcal{D}_n = \{\}$. There are notable three bijections among these sets; refer to Figure 14.2 to help visualize them.

1. There is a bijection $\mathcal{S}_n \rightsquigarrow \mathcal{T}_{n-1}$.

For any tiling in $\mathcal{S}_n$, we can cut off the first cell, and what remains is a tiling in $\mathcal{T}_{n-1}$. For any tiling in $\mathcal{T}_{n-1}$, we can attach a cell on the left with a square tile, yielding a tiling in $\mathcal{S}_n$. These two operations are inverses of each other.

2. There is a bijection $\mathcal{D}_n \rightsquigarrow \mathcal{T}_{n-2}$.

The same argument works here except we cut off or glue back two cells with a domino in them.

3. There is a bijection $\mathcal{T}_n \rightsquigarrow \mathcal{S}_n \sqcup \mathcal{D}_n$.

Every tiling of an n -board must start with *either* a square *or* domino; map that tiling to one component or the other of the disjoint union accordingly.

Conversely, map a tiling $\langle t, i \rangle \in \mathcal{S}_n \sqcup \mathcal{D}_n$ to $t \in \mathcal{T}_n$. Again, these two mappings are inverse functions.

These bijections and the Counting Principle tell us that

1. $\#\mathcal{S}_n = \#\mathcal{T}_{n-1} = t_{n-1}$.
2. $\#\mathcal{D}_n = \#\mathcal{T}_{n-2} = t_{n-2}$.
3. $t_n = \#\mathcal{T}_n = \#(\mathcal{S}_n \sqcup \mathcal{D}_n) = \#\mathcal{S}_n + \#\mathcal{D}_n = t_{n-1} + t_{n-2}$.

Hence, $t_0 = 1, t_1 = 1$ and $t_n = t_{n-1} + t_{n-2}$, which implies that t_n is the $(n+1)^{\text{st}}$ [Fibonacci number](#).

We can use the same approach to learn more about the t_n 's. Consider the identity (*) below. Is it true for all $n \in \mathbb{N}$?

$$t_0 + t_1 + t_2 + \cdots + t_n = t_{n+2} - 1, \quad (*)$$

The left and right sides of (*) count the elements in two sets:

1. Let $\mathcal{L}_k$ be the set of tilings of an $(n+2)$ -board in which the *last* domino covers cells $k+1$ and $k+2$. Any tiling in $\mathcal{L}_k$ contains, from cell $k+1$ to cell $n+2$, one domino and the rest squares, which can only happen in one way. So, $\#\mathcal{L}_k = t_k$, and the left-hand side counts the number of tilings in $\bigsqcup_{k=0}^n \mathcal{L}_k$.
2. The right-hand side counts all the tilings of an $(n+2)$ -board *except* for the tiling s that consists of all squares.

Then, $f: \bigsqcup_{k=0}^n \mathcal{L}_k \xrightarrow{\sim} \mathcal{T}_{n+1} - \{s\}$ that maps $\langle t, k \rangle$ to t is a bijection. The inverse function maps a tiling t to $\langle t, k \rangle$ where $k+1$ is first cell of the last domino (which exists since we've excluded tiling s). The Counting Principle then implies (*).

14.2 Analytical Constraints

In practice, we come to see some functions as “better behaved” than others. These may be easier to graph or to compute or to reason about; they avoid some of the pathological complexity that exists in the vast universe of possible functions; and so we find them convenient and, in a way, comforting. Some of the properties that make a function “well behaved” arise quite naturally as conditions in our analysis or emerge from the mechanisms we use to build functions. We mostly apply these to functions that take or return numbers or tuples of numbers, but all of these properties can be generalized. We look at a different type of good behavior in Chapter 17. This

Section gives a high-level overview of several common and important properties; for a first reading, it is reasonable to get a quick sense of what these properties are and to delve into them in detail when we use them later.

MONOTONE FUNCTIONS. The real numbers are *ordered*: if $x, y \in \mathbb{R}$, then exactly one of $x < y$, $x = y$, or $x > y$ must be true. We must also have $x \leq y$ or $x \geq y$, where $x \leq y$ and $x \geq y$ implies that $x = y$. We can extend this order to tuples, points $v = \langle v_1, v_2, \dots, v_n \rangle \in \mathbb{R}^n$, via **lexicographic ordering**: compare the first unequal components, or if all components are equal the points are equal.⁸ That is, for $v, w \in \mathbb{R}^n$, we have $v < w$ if $v_1 < w_1$ or if $v_1 = w_1 \wedge v_2 < w_2$ or if $v_1 = w_1 \wedge v_2 = w_2 \wedge v_3 < w_3$ or ... if $v_1 = w_1 \wedge v_2 = w_2 \wedge \dots \wedge v_n < w_n$. Similarly, we say that $v \leq w$ if $v_i \leq w_i$ for $i \in [1..n]$. (The relations $>$ and $\geq$ just the opposites.)

⁸This is how we sort words in “alphabetic order.”

A function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ is **monotone** if either (i) $x < y$ implies $f(x) \leq f(y)$ or (ii) $x < y$ implies $f(x) \geq f(y)$. The first case is called order-preserving or **non-decreasing**; the second case is called order-reversing or **non-increasing**.

This definition allows f to return the same value for two different inputs. If we require a monotone function to be an injection, we get a *strictly* monotone function. A function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ is **monotone** if either (i) $x < y$ implies $f(x) < f(y)$ or (ii) $x < y$ implies $f(x) > f(y)$. The first case is called **increasing**; the second case is called **decreasing**.

For example, both the so-called “rectified linear unit” $\text{relu}(x) = \max(x, 0)$ and the function j shown below in Figure 14.3 are monotone; relu is non-decreasing and j is non-increasing. The “sigmoidal” functions illustrated in Example 14.6 are monotone injections and thus are strictly monotone, all increasing. The function $r: (0_-) \rightarrow (0_-)$ with $r(x) = 1/x$ is strictly monotone, and specifically decreasing. The functions $f, g: \mathbb{R} \rightarrow \mathbb{R}^3$ with $f(t) = \langle t, t^3, e^t \rangle$ and $g(t) = \langle t, \lfloor t \rfloor, \lceil t \rceil \rangle$ are both monotone, with f increasing and g non-decreasing. The functions $\sin$ and $s(x) = x^2$ are *not* monotone.

If a *strictly* increasing f is a bijection, then its inverse f^{-1} is strictly increasing as well. (If $y_1 = f(x_1) < y_2 = f(x_2)$, then we must have $x_1 = f^{-1}(y_1) < x_2 = f^{-1}(y_2)$.)

The idea of monotone functions carries over to sets with other order relations defined. See Chapter 16 for more.

BOUNDED FUNCTIONS. Both numbers and tuples of numbers have a *magnitude*. For $x \in \mathbb{R}$, its absolute value $|x|$ is x if $x \geq 0$ and $-x$ if $x < 0$. We can extend this to tuples: if $v = \langle v_1, \dots, v_n \rangle \in \mathbb{R}^n$, define $|v| = \sqrt{v_1^2 + \dots + v_n^2}$. This reduces to the

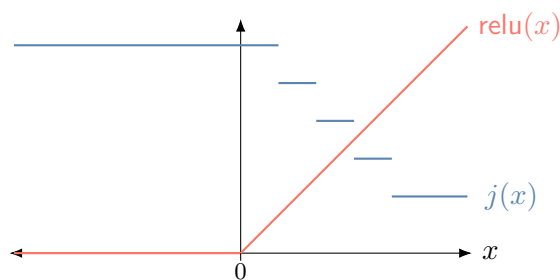


FIGURE 14.3. The graphs of two monotone functions; relu is non-decreasing and j is non-increasing.

absolute value when $n = 1$; in general $|v|$ is called the (Euclidean) **norm** of v .

A function with codomain $\mathbb{R}^n$ for some n is **if** there is a constant $B < \infty$ such that $|f(x)| \leq B$ for all $x \in \text{dom}(f)$. For any input, we are guaranteed that the returned value's magnitude cannot get arbitrarily large.

For example, the logistic function $\ell(x) = \frac{1}{e^{-x} + 1}$ is bounded (with bound 1), but the exponential function e^x is not bounded. The function d in Example 13.9 is bounded, but the function $n(x) = |x|$ on $\mathbb{R}^k$ is not.

SMOOTH FUNCTIONS. A function is smooth when small changes to the input value produce small changes in the output value. There are several, increasingly restrictive notions of smoothness that we touch on here. We rarely make use of the technical details, so we will cover the ideas at a high level so they are familiar. More detail is given in Appendix B, if needed.

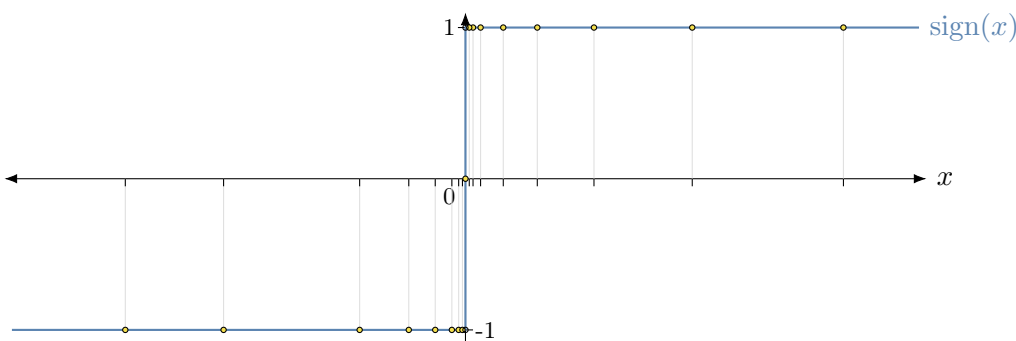


FIGURE 14.4. The graph of a function that is not continuous, indicating its value at points in a sequence approaching 0.

To motivate these ideas, we start with a function that is *not* smooth, as shown in Figure 14.4. This function returns 0 at 0, -1 for $x < 0$, and 1 for $x > 0$, so it *jumps*

suddenly from 1 to 0 to -1. The figure shows two sequences of points approaching 0, one from the right and one from the left. If we evaluate the function at the points in the sequence, the values are all 1 or -1, giving limiting endpoints displayed with the open circles at $\langle 0, 1 \rangle$ and $\langle 0, -1 \rangle$. But the function is not equal to 1 or -1 at these points because of the jump. This is a failure of *continuity*.

A function f is **continuous** if for every a sequence of values $x_0, x_1, x_2, \dots \in \text{dom}(f)$ that converges to $x \in \text{dom}(f)$, the sequence $f(x_0), f(x_1), f(x_2), \dots$ converges to $f(x)$. Loosely speaking, a function $\mathbb{R} \rightarrow \mathbb{R}$ is continuous if we can draw the graph without lifting the pencil from the paper.

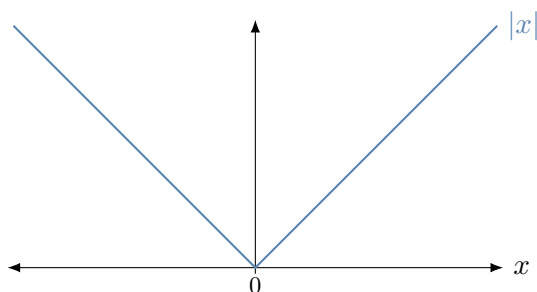


FIGURE 14.5. The graph of a function that is continuous but not differentiable. The sharp corner at 0 implies that the derivative is not defined at 0; the slope *jumps* there.

A stronger notion of smoothness is *differentiability*. Recall that the derivative f' of a function f is a local linear approximation to the function.⁹ Specifically, for $x_0 \in \text{dom}(f)$, $f'(x_0)$ is defined so that

$$\frac{|f(x) - (f(x_0) + f'(x_0)(x - x_0))|}{|x - x_0|} \rightarrow 0 \quad (14.1)$$

as x approaches x_0 .¹⁰ The derivative need not exist; a function f is **differentiable** if the derivative exists for all inputs in $\text{dom}(f)$. For numeric functions, $f'(x_0)$ is the slope of the tangent to the graph at the point $\langle x_0, f(x_0) \rangle$.¹¹ A differentiable function is necessarily continuous, though the derivative need not be. A differentiable function whose derivative is also continuous is said to be **continuously differentiable**.

An even stronger form of smoothness is when the derivative itself is differentiable. A function is twice differentiable if its derivative is differentiable; it is three times differentiable if its derivative is twice differentiable; and for $k \in [1..)$, a function is **k -times differentiable** if its derivative is $(k - 1)$ -times differentiable. Taken to extremes, we get an *infinitely-differentiable* functions, where every derivative is differentiable. Finally, we say that f is **(real) analytic** if f is infinitely-differentiable

⁹The derivative is denoted in various ways depending on the context. The k -th derivative is often denoted $f^{(k)}$, where $f^{(0)} = f$.

¹⁰Formally, this is expressed through the idea of a *limit*.

¹¹In general, the derivative at x_0 , $f'(x_0)$, acts as a *linear function* in equation (14.1); see Example 17.40. When f is real valued, this is just multiplying by the number $f'(x_0)$.

and its *Taylor series*

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!} (x - x_0)^k \quad (14.2)$$

converges for every x in some interval $(x_0 - \delta, x_0 + \delta)$.

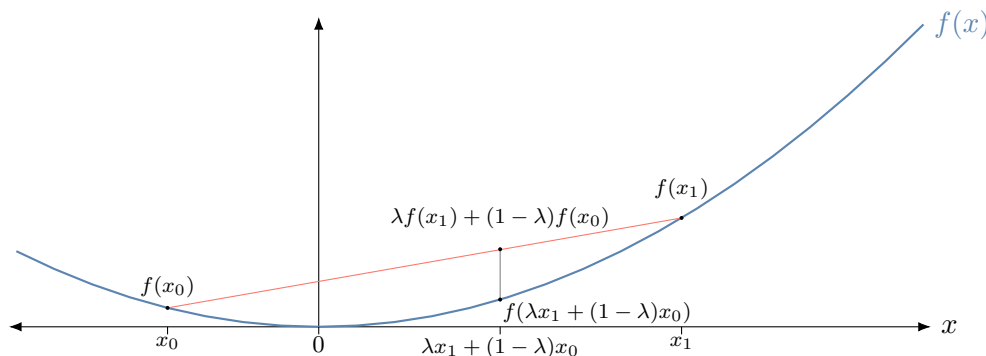


FIGURE 14.6. The graph of a (strictly) convex function illustrating equation (14.3).

CONVEX FUNCTIONS. A convex function is one whose graph is continuous and vaguely “U”-shaped. Formally, we say that a function f is **convex** if for any $x_0, x_1 \in \text{dom}(f)$ and any $\lambda \in [0, 1]$,

$$f(\lambda x_1 + (1 - \lambda)x_0) \leq \lambda f(x_1) + (1 - \lambda)f(x_0), \quad (14.3)$$

as illustrated in Figure 14.6. As λ varies from 0 to 1, the value $\lambda x_1 + (1 - \lambda)x_0$ moves from x_0 to x_1 and the value $\lambda f(x_1) + (1 - \lambda)f(x_0)$ moves from $f(x_0)$ to $f(x_1)$. The definition says that the points on the latter line segment must be no lower than the corresponding points on the graph of the function. The pictured function actually satisfies a slightly stronger condition; the inequality in equation (14.3) is strict here. We say that a function is **strictly convex** when the $\leq$ in equation (14.3) can be replaced with $<$. For instance, the function $\text{id}: \mathbb{R} \rightarrow \mathbb{R}$ is convex but not strictly convex, whereas $m: \mathbb{R}^n \rightarrow [0, \infty)$ with $m(x) = |x|^2$ is strictly convex. A function g is said to be **concave** (**strictly concave**) if $-g$ is convex (strictly convex).

Puzzle 126. Is the function $\sin$ convex or concave or neither? Give an argument or a picture to support your answer.

Puzzle 127. Sketch a strictly convex function $\mathbb{R} \rightarrow \mathbb{R}$ that has a local minimum. Explain with words or a picture why this function can have only one local minimum.

Tuple/Vector Functions

15

Chapter

Contents

15.1 Functions on Tuples	672
15.2 Projections	673
15.3 Permutations	674
15.4 Joining Tuples	675
15.5 Combining Tuple Functions	676

As we have seen throughout the previous chapter, **tuples** are lists of items of a fixed length. The entries in the list are called the tuple's **components**, and the number of components is the tuple's **dimension**. In mathematical use, we typically 1-index the components of the tuple. That is, if x is a tuple of dimension d , then x_1 is the first component, x_2 the second, and so forth. In this case, we can write $x = \langle x_1, x_2, \dots, x_d \rangle$. A tuple of dimension 0 has no components, and there is only one, the *empty tuple* $\langle \rangle$.

Tuples on which we define two extra operations – *scaling* them by a number and *adding* them component by component – are called **vectors**. For example, $2\langle x, y \rangle \equiv \langle 2x, 2y \rangle$ scales the components by 2, and $\langle x, y, z \rangle + \langle a, b, c \rangle \equiv \langle x + a, y + b, z + c \rangle$ adds corresponding components together. Any tuple of numbers, for instance, can be considered a vector with scaling and addition defined by

$$c\langle x_1, x_2, \dots, x_d \rangle = \langle cx_1, cx_2, \dots, cx_d \rangle \quad (\text{Scaling})$$

$$\langle x_1, x_2, \dots, x_d \rangle + \langle y_1, y_2, \dots, y_d \rangle = \langle x_1 + y_1, x_2 + y_2, \dots, x_d + y_d \rangle. \quad (\text{Additivity})$$

We discuss vectors in general in Section 17.3.

15.1 Functions on Tuples

Consider a function $\psi: \mathbb{R}^2 \rightarrow \mathbb{R}$. We think of this function in two ways. First and most directly, the function takes as input pairs of real $v \in \mathbb{R}^2$ and returns a number $\psi(v)$. If we write the tuple explicitly $v = \langle v_1, v_2 \rangle$, this looks like $\psi(\langle v_1, v_2 \rangle)$. The second way think about ψ is as a function that takes *two* real arguments v_1, v_2 and returns a number $\psi(v_1, v_2)$. One key point of this section is that we treat these three expressions – $\psi(v)$, $\psi(\langle v_1, v_2 \rangle)$, $\psi(v_1, v_2)$ – as equivalent and interchangeable.

A function $\psi: \mathcal{A}_1 \times \mathcal{A}_2 \times \dots \times \mathcal{A}_n \rightarrow \mathcal{B}$ can be viewed interchangeably as either taking a single argument that is an n -tuple or taking n arguments (from $\mathcal{A}_1, \dots, \mathcal{A}_n$ respectively). As such, when evaluating ψ , given an argument $v = \langle v_1, \dots, v_n \rangle$ we treat the following as **equivalent and interchangeable** ways to write ψ 's return value:

$$\psi(v) \qquad \psi(\langle v_1, \dots, v_n \rangle) \qquad \psi(v_1, \dots, v_n),$$

using whichever form is clearest and most convenient at any moment.

If a function ψ *returns* a tuple of dimension n , then for every input $\psi(x)$ is an n -tuple. We can consider the i th component of $\psi(x)$ as the output of a function

ψ_i . As such, we write $\psi = \langle \psi_1, \psi_2, \dots, \psi_n \rangle$, and the ψ_i 's are called the **component functions** of ψ . Evaluating ψ at x gives us $\psi(x) = \langle \psi_1(x), \psi_2(x), \dots, \psi_n(x) \rangle$.

15.2 Projections

If we have a tuple, we often want to extract particular components. A function that extracts one or more components from a tuple is called a **component projection**. We denote component projections with a function named **proj** using subscripts to indicate which components are extracted.¹ If $u = \langle u_1, \dots, u_n \rangle$, then

$$\begin{aligned} \text{proj}_i(u) &= u_i && \text{for any } 1 \leq i \leq n \\ \text{proj}_{ij}(u) &= \langle u_i, u_j \rangle && \text{for any } 1 \leq i \leq j \leq n \\ \text{proj}_{ijk}(u) &= \langle u_i, u_j, u_k \rangle && \text{for any } 1 \leq i \leq j \leq k \leq n \\ &\dots && \\ \text{proj}_{i_1 i_2 \dots i_m}(u) &= \langle u_{i_1}, u_{i_2}, \dots, u_{i_m} \rangle && \text{for any } 1 \leq m \leq n, 1 \leq i_1 \leq \dots \leq i_m \leq n. \end{aligned} \quad (15.1)$$

¹Notice that proj_i and proj_{ij} and so forth correspond to $\text{Proj}[i]$ and $\text{Proj}[i, j]$ in **frplib** and Part I.

In the subscript, we can freely use commas between the components if needed for clarity, such as when using numbers with more than one digit, e.g., $\text{proj}_{12,37}$, $\text{proj}_{1,17,94}$. So $\text{proj}_{12,37}(u) = \langle u_{12}, u_{37} \rangle$ whereas $\text{proj}_{1237}(u) = \langle u_1, u_2, u_3, u_7 \rangle$. For a single multi-digit number, use a trailing comma or parentheses as you prefer: proj_{17} , or $\text{proj}_{(17)}$ extracts the 17th component, not the first and seventh. If we want to extract a *range* of components, say all the components from index i up to and including index j , we write the range as $i..j$ in the subscript. If either endpoint is missing like $..j$ or $i..$, the range extends all the way to the first or last component, respectively. So, $\text{proj}_{i..j}(u) = \langle u_i, u_{i+1}, \dots, u_j \rangle$ and $\text{proj}_{..3}(u) = \langle u_1, u_2, u_3 \rangle$.

If a function $\psi: \mathcal{A} \rightarrow \mathcal{B}_1 \times \dots \times \mathcal{B}_n$ returns a tuple of dimension n , then the component functions $\psi_i: \mathcal{A} \rightarrow \mathcal{B}_i$ for $i \in [1..n]$ satisfy $\psi_i = \text{proj}_i \circ \psi$, making the following diagram commute:

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{\psi} & \mathcal{B}_1 \times \dots \times \mathcal{B}_n \\ & \searrow \psi_i & \downarrow \text{proj}_i \\ & & \mathcal{B}_i \end{array}$$

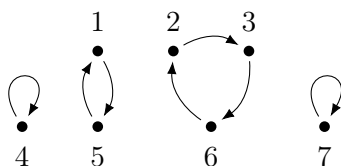
Sometimes it is easier to specify projections by which coordinates are *excluded*, so for convenience, we define complementary projections $\overline{\text{proj}}$, where $\overline{\text{proj}}_i$ drops the i th component, $\overline{\text{proj}}_{ij}$ drops the i th and j th, $\overline{\text{proj}}_{ijk}$ the i th, j th, and k th, and so on. These

accept tuples with dimension at least as large as the maximum index specified. So, for instance, $\overline{\text{proj}}_4(\langle 1, 2, 3, 4 \rangle) = \langle 1, 2, 3 \rangle$ and $\overline{\text{proj}}_{13}(\langle 1, 2, \dots, 100 \rangle) = \langle 2, 4, 5, \dots, 100 \rangle$.

15.3 Permutations

Another common operation on tuples is to rearrange their components; functions that do this are called **permutations**. We use the base name **permute** to denote the family of functions that take a tuple as input and returns a permuted tuple as output. A subscript on the name, e.g., **permute**₂₁ or **permute**₄₃₁, indicates which permutation it is. Each such function is polymorphic in the sense described earlier: it takes *any* tuple of length greater than or equal to the largest index in the subscript and returns a permuted tuple of the same length.

We interpret the subscript as describing the *cycles* that comprise the permutation. A **cycle** of a permutation is a sequence of values $v_1, v_2, \dots, v_m$ in the domain of the permutation that map successively each other, wrapping around from last to first; that is, v_1 maps to v_2 , v_2 to v_3 , and so on until v_{m-1} maps to v_m and v_m maps to v_1 . For example, consider a permutation $p: [1..7] \rightarrow [1..7]$ with $p(1) = 5, p(2) = 3, p(3) = 6, p(4) = 4, p(5) = 1, p(6) = 2, p(7) = 7$. For each $i \in [1..7]$, we can look at the sequence $i, p(i), p(p(i)), p(p(p(i))), \dots$; this sequence eventually repeats and cycles *ad infinitum*. For instance, the sequence for 1 is 1, 5, 1, 5, 1, 5, ... giving 1, 5 as a cycle, for 2 is 2, 3, 6, 2, 3, 6, 2, ... giving the cycle 2, 3, 6, and for 4 is 4, 4, 4, 4, ... giving the cycle 4. Every element of $[1..7]$ appears in exactly one such cycle, so the permutation p is *composed of the cycles* as (4)(51)(623)(7), which corresponds to the following simple directed graph with loops:



Our convention for describing a permutation is to list the cycles of a permutation such that: (i) within a cycle, we *start with the largest index*, and (ii) we list cycles in *increasing order of largest index*, yielding 4516237. With this convention, we need no separators like $()$'s between cycles because *a cycle boundary occurs at any number that is bigger than all the earlier numbers in the list*. We do need to distinguish multi-digit and single-digit numbers, any indices > 9 are delimited with ','s.

The subscripts of **permute** refer to the components being permuted and follow

this convention with the exception that we *exclude cycles with just one element from the subscript*. For instance, 51623 leaves 4 and 7 untouched with cycles (51) and (623), as above. So, $\text{permute}_{51623}(u_1, u_2, \dots, u_7) = \langle u_5, u_3, u_6, u_4, u_1, u_7 \rangle$, and $\text{permute}_{31}(u_1, u_2, u_3, \dots, u_n) = \langle u_3, u_2, u_1, u_4, \dots, u_n \rangle$, for any $n \in [3..]$. This is still unambiguous and makes it easier to specify simple permutations and keeps the subscripts more manageable in practice.

Puzzle 128. Describe an algorithm that takes a list of distinct indices in $[1..n]$ ordered by our convention for listing the cycles of a permutation and returns a list of cycles in increasing order of largest index. So 2365714 with $n = 7$ would produce (2)(3)(65)(714) and with $n = 4$, 2143 would produce (21)(43).

Can you simply describe an algorithm for going the other way? Are these two algorithms inverses of each other?

15.4 Joining Tuples

We often build tuples from smaller tuples and deconstruct larger tuples into smaller pieces. For this, we use the $::$ operator, read as “join.” This concatenates tuples of possibly different dimensions, into a combined and “flattened” tuple that contains all of the components in the order given. This is an associative, binary operator defined as follows. For $n, m, r \in [0..]$ and given tuples $u = \langle u_1, \dots, u_m \rangle$, $v = \langle v_1, \dots, v_n \rangle$, $w = \langle w_1, \dots, w_r \rangle$, we define

$$\begin{aligned} u :: v &= \langle u_1, \dots, u_m, v_1, \dots, v_n \rangle \\ u :: v :: w &= \langle u_1, \dots, u_m, v_1, \dots, v_n, w_1, \dots, w_r \rangle, \end{aligned} \tag{15.2}$$

and similarly for any number of terms. So, for instance, $\langle 1, 2 \rangle :: \langle 3, 4, 5 \rangle = \langle 1, 2, 3, 4, 5 \rangle$. This is an associative operator (meaning $(u :: v) :: w = u :: (v :: w)$) but is not commutative (meaning that $u :: v$ need not equal $v :: u$). Associativity means that we can write $u :: v :: \dots :: z$ unambiguously for any number of terms.

A one-dimensional tuple $\langle x \rangle$ and the value x it contains are for practical purposes *interchangeable*, so we elide the superficial distinction between them. Thus, we allow $::$ to accept a value in lieu of the corresponding 1-tuple; so, $1 :: \langle 2, 3 \rangle = \langle 1, 2, 3 \rangle = \langle 1 \rangle :: \langle 2, 3 \rangle$ and $1 :: 2 = \langle 1, 2 \rangle = \langle 1 \rangle :: \langle 2 \rangle$.

15.5 Combining Tuple Functions

It will prove useful to have short-hands for combining functions that take or return tuples. Here, we define operators for several different ways of combining that we will use frequently.

Skip on first reading

Here, let ψ and ϕ be functions and let $m, n \in [1..)$. We will define several operations on functions (with corresponding operators) that produce new functions: fork ($\curlyvee$), join ($::$), tensor product ($\otimes$), and Cartesian product ($\times$).

- **join** $\psi :: \phi$, defined by $(\psi :: \phi)(u :: v) = \psi(u) :: \phi(v)$.

Here ψ and ϕ take as input m -dimensional and n -dimensional, respectively, and $\psi :: \phi$ takes an $(m + n)$ -tuple. Specifically, for some arbitrary sets, we assume

$$\begin{aligned}\psi &: \mathcal{A}_1 \times \cdots \times \mathcal{A}_m \rightarrow \mathcal{C} \\ \phi &: \mathcal{B}_1 \times \cdots \times \mathcal{B}_n \rightarrow \mathcal{D}.\end{aligned}$$

$\psi :: \phi$ takes an $(m + n)$ -tuple $\langle a_1, \dots, a_m, b_1, \dots, b_n \rangle$, passes $\langle a_1, \dots, a_m \rangle$ to ψ and $\langle b_1, \dots, b_n \rangle$ to ϕ , and does a flattening join on their return values, yielding $\psi(a_1, \dots, a_m) :: \phi(b_1, \dots, b_n)$. Both m and n are determined by the domains of ψ and ϕ ; they can be 1. The re-use of the join operator $::$ here is representative of what the operator does – join the returned values of the functions.

- **fork** $\psi \curlyvee \phi$, defined by $(\psi \curlyvee \phi)(x) = \psi(x) :: \phi(x)$

Here, ψ and ϕ have an arbitrary but *common domain*. $\psi \curlyvee \phi$ takes an input from that domain, and passes the same input to both ψ and ϕ , joining their result into a (flattened) tuple. The $\curlyvee$ notation for the operator is designed to suggest one input being split to two outputs.

- **tensor product** $\psi \otimes \phi$, defined by $(\psi \otimes \phi)(u :: v) = \psi(u) \phi(v)$

Here, ψ and ϕ take m -dimensional and n -dimensional tuples as input, and *return* real numbers, and $\psi \otimes \phi$ returns a real number as well.

The Tensor product splits the input like join but then takes the *product* of the functions' return values. Put in terms of components, this becomes:

$$(\psi \otimes \phi)(w_1, \dots, w_m, w_{m+1}, \dots, w_{m+n}) = \psi(w_1, \dots, w_m) \phi(w_{m+1}, \dots, w_{m+n}). \quad (15.3)$$

The $\otimes$ notation is intended to evoke multiplication and Cartesian product while being visually distinct from either.

- **Cartesian product** $\psi \times \phi$, defined by $(\psi \times \phi)(u, v) = \langle \psi(u), \phi(v) \rangle$.

Very similar to the join, but does not explicitly flatten the resulting tuples. In our applications with functions returning tuples of numbers, we tend to use $\psi :: \phi$ instead to avoid implicit nesting of tuples.

All of these operators are associative but need not be commutative.

Here are a few concrete examples of how these operators are used:

1. The function $c = \cos \curlyvee \sin$ is given by $c(t) = \langle \cos(t), \sin(t) \rangle$. It has type $\mathbb{R} \rightarrow \mathbb{S}^1$, where $\mathbb{S}^1 \subseteq \mathbb{R}^2$ is the unit circle. We think of $c(t)$ as the position at time t of a particle moving in a circle.

We can speed up the motion of the particle by modifying the function; for instance, $\tilde{c}_\nu = (\cos(2\pi\nu\blacksquare)) \curlyvee (\sin(2\pi\nu\blacksquare)) = c \circ (2\pi\nu\blacksquare)$ has frequency ν .

2. If $f: \mathcal{A} \rightarrow \mathcal{B}$, then $\text{id} \curlyvee f$ returns for each $a \in \mathcal{A}$ a point $\langle a, f(a) \rangle$ on $\text{graph}(f)$.
3. If $\mathcal{U}$ and $\mathcal{V}$ are indicators of sets $\mathcal{U}$ and $\mathcal{V}$, $\mathcal{U} \otimes \mathcal{V}$ is the indicator of $\mathcal{U} \times \mathcal{V}$.
4. Suppose that $\mathcal{I}$ is the set of interviewers at a particular company. Applicants are given the same series of questions, with each question posed by a particular interviewer. However, interviewers have different approaches and rate applicants on their own scales. Let $\phi_q: \mathcal{I} \rightarrow [0\_1]$ return the proportion of applicants that will get a “good” score when question q is administered by a given interviewer. Then $\phi_1 \otimes \phi_2$ with $(\phi_1 \otimes \phi_2)(i_1, i_2)$ is the proportion of applicants who get a “good” score on both questions when assigned interviewers i_1 and i_2 . Similarly, $\phi_1 \otimes \phi_2 \otimes \cdots \otimes \phi_k$ returns for any assignment of k interviewers to the k questions, what portion of applicants will get a “good” score on all the questions.
5. If f returns the list of capabilities of products of one type and g returns the list of capabilities of products of another type, then $f :: g$ returns the list of capabilities for a pair of products of each type. The function $f \times g$ is analogous but returns a pair whose components are themselves lists.

Relations

16

Chapter

A function $f: \mathcal{A} \rightarrow \mathcal{B}$ depicts a kind of relationship between its domain $\mathcal{A}$ and codomain $\mathcal{B}$, associating some points in $\mathcal{B}$ with each point in $\mathcal{A}$. Let's turn this around to emphasize the relationship. Define a Boolean¹ function $\tilde{f}: \mathcal{A} \times \mathcal{B} \rightarrow \mathbb{B}$ to capture this relationship:

$$\tilde{f}(a, b) = \begin{cases} \top & \text{if } b = f(a) \\ \perp & \text{if } b \neq f(a). \end{cases}$$

Then $\tilde{f}$ has exactly the same information that f does, and $\tilde{f}^{-1}(\top) \subseteq \mathcal{A} \times \mathcal{B}$ is the set of pairs $\langle a, f(a) \rangle$, which is exactly $\text{graph}(f)$!²

Functions are required to return only one value for each input, so the “slice” of $\text{graph}(f)$ with first coordinate a is a set with only one element $\langle a, f(a) \rangle$. However, we can generalize this way of relating points in two sets. Let $r: \mathcal{A} \times \mathcal{B} \rightarrow \mathbb{B}$ be an arbitrary function. If the function r returns $\top$ (true) for a pair $\langle a, b \rangle$, it tells us that a and b are related in some way. Recall from Chapter 14 that there is a bijection between functions from a set into $\mathbb{B}$ and subsets of that set. Corresponding to the function r is the set $r^{-1}(\top) \subseteq \mathcal{A} \times \mathcal{B}$, the set of pairs for which the “relation” holds. Accordingly, we call any subset of $\mathcal{A} \times \mathcal{B}$ a *binary relation* on $\mathcal{A}$ and $\mathcal{B}$; if $\mathcal{A} = \mathcal{B}$ we just call it a binary relation on $\mathcal{A}$.

We have already seen that every function from $\mathcal{A}$ to $\mathcal{B}$ gives rise to a binary relation between the sets (i.e., the function's graph), and other examples abound. If $\mathcal{P}$ is a set of people, then two relations on $\mathcal{P}$ are “likes” ($\mathcal{L}$) and “is friends with” ($\mathcal{F}$), where if p likes p' then $\langle p, p' \rangle \in \mathcal{L} \subseteq \mathcal{P} \times \mathcal{P}$ and if p is friends with p' then $\langle p, p' \rangle \in \mathcal{F} \subseteq \mathcal{P} \times \mathcal{P}$. “Likes” can be an asymmetric relation, so $\langle p, p' \rangle \in \mathcal{L}$ does not imply $\langle p', p \rangle \in \mathcal{L}$. It is possible for $\langle p, p' \rangle$ and $\langle p', p \rangle$ to both belong in $\mathcal{L}$, but it is not required. In contrast, “friends” is a symmetric relation, so $\langle p, p' \rangle \in \mathcal{F}$ implies $\langle p', p \rangle \in \mathcal{F}$ by definition.

If $\mathcal{C}$ is a set of cities, we can ask whether there is a direct flight available between

¹Recall that $\mathbb{B} = \{\perp, \top\}$ is the set of Boolean values, false ($\perp$) and true ($\top$).

²See page 581 for definition of the inverse image and equation (10.2) for the definition of graph.

cities $c, c' \in \mathcal{C}$. This gives rise to a binary relation on $\mathcal{C}$ containing the pairs of cities for which a direct flight is available. Asking instead whether there is a flight with zero or one stops, gives a different relation, containing all the pairs from the first. You might also wonder what happens if we add a set $\mathcal{A}$ of airlines into the mix and ask whether there is a direct flight from c to c' *on airline* $a \in \mathcal{A}$. This will indeed give us a kind of relation as we will see.

If these examples remind you of graphs (the node and edge kind) that we introduced in Examples 10.18 and 10.19, it's for good reason. Graphs model binary relations between the entities represented by their nodes. Any simple graph (direct or undirected, with or without loops) immediately gives rise to a binary relation as we have defined it here. For instance, if $G = \langle \mathcal{N}, \mathcal{E}, \varphi \rangle$ is a directed simple graph, then $\varphi^\rightarrow(\mathcal{E}) \subseteq \mathcal{N} \times \mathcal{N}$ gives the binary relation on $\mathcal{N}$ that two nodes are incident on the same edge (aka. adjacent or neighbors). If $G' = \langle \mathcal{N}', \mathcal{E}', \psi \rangle$ is an undirected simple graph, then $\psi^\rightarrow(\mathcal{E}')$ is a set of sets $\{n_0\}$ for loops and $\{n_1, n_2\}$ for undirected edges. For the former, we have $\langle n_0, n_0 \rangle \in \mathcal{N} \times \mathcal{N}$ and for the latter, both pairs $\langle n_1, n_2 \rangle$ and $\langle n_2, n_1 \rangle$. Combining all these pairs across all elements of $\psi^\rightarrow(\mathcal{E}')$ again gives us a binary relation on $\mathcal{N}'$ that nodes are adjacent.

Binary relations are the most commonly used, but relations can be connect multiple entities.

An ***n*-ary relation** on sets $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ is a set $\mathcal{R} \subseteq \mathcal{A}_1 \times \mathcal{A}_2 \times \dots \times \mathcal{A}_n$. The sets $\mathcal{A}_i$ are called the *domains* of the relation, and n is its *arity*.²

If $n = 1$, we call it a *unary* relation or – more evocatively – a **property**, if $n = 2$, we call it a *binary* relation, and if $n = 3$, we call it a *ternary* relation. When we omit the arity referring to a particular relation, we take it to be a binary relation by default, unless otherwise clear from context.

If all the domains of a relation are the same set, we call the relation *homogeneous*; if any are not equal, the relation is *heterogeneous*.

Every n -ary relation corresponds to a unique Boolean function of type

$$\mathcal{A}_1 \times \mathcal{A}_2 \times \dots \times \mathcal{A}_n \rightarrow \mathbb{B},$$

called its **predicate**, that returns $\top$ for every tuple in $\mathcal{R}$ and $\perp$ otherwise. We can use either the relation or its predicate as needed.

Binary relations are often depicted in *infix form*, with $a \mathcal{R} b$ written for $\langle a, b \rangle \in \mathcal{R}$ and $a \not\mathcal{R} b$ for $\langle a, b \rangle \notin \mathcal{R}$. Binary relations are often represented by

²The arity is also sometimes called the *degree* of the relation.

operators as well as set names (e.g., $\leq$, $\sim$, $\neg$, $<$).

Relations are useful for the same reason that graphs and functions are. They model a wide variety of relationships that arise in practice, including one-to-many and many-to-one relationships that are not functions. We also use relations for describing orderings of objects as described in Chapter 17. And a concept that arises frequently is that of an *equivalence relation*, discussed below. We start by considering various examples and types of relations.

Example 16.1 Where No Relation Has Gone Before

Consider a set of *Star Trek* characters:

$\{\text{Kirk, Spock, McCoy, Uhura, Sulu, Pike, Picard, Sarek, Sisko, Janeway, Riker, Seven of Nine}\}$

The binary relation “shared a mind meld” on this set is

$\{\langle \text{Kirk, Spock} \rangle, \langle \text{Spock, Kirk} \rangle, \langle \text{McCoy, Spock} \rangle, \langle \text{Spock, McCoy} \rangle, \langle \text{Picard, Spock} \rangle,$
 $\langle \text{Spock, Picard} \rangle, \langle \text{Picard, Sarek} \rangle, \langle \text{Sarek, Picard} \rangle, \langle \text{Kirk, Spock} \rangle, \langle \text{Spock, Kirk} \rangle, \}$

the relation “stranded together in a bizarre region outside time and space” is

$\{\langle \text{Kirk, Spock} \rangle, \langle \text{Spock, Kirk} \rangle, \langle \text{Picard, Kirk} \rangle, \langle \text{Kirk, Picard} \rangle,$
 $\langle \text{Janeway, Seven of Nine} \rangle, \langle \text{Seven of Nine, Janeway} \rangle, \}$

Notice that every pair of the last two relation appears twice in both orders because the relationships are mutual. In contrast, the binary relation “recommended for command” has direction:

$\{\langle \text{Kirk, Spock} \rangle, \langle \text{Kirk, Sulu} \rangle, \langle \text{Pike, Kirk} \rangle, \langle \text{Picard, Riker} \rangle, \langle \text{Janeway, Seven of Nine} \rangle\}.$

Two unary relations are: “captained a starship” $\{\text{Kirk, Spock, Sulu, Pike, Picard, Sisko, Janeway}\}$ and “sings or plays a musical instrument” $\{\text{Uhura, Spock, Riker, Picard}\}$. And finally, a ternary relation “joined together in a corny laugh-track ending”:

$\{\langle \text{Kirk, Spock, McCoy} \rangle, \langle \text{Kirk, McCoy, Spock} \rangle, \langle \text{Spock, Kirk, McCoy} \rangle,$
 $\langle \text{Spock, McCoy, Kirk} \rangle, \langle \text{McCoy, Spock, Kirk} \rangle, \langle \text{McCoy, Kirk, Spock} \rangle\},$

All six tuples are included because there is no order to the relationship.

Similarly, all the familiar comparison operators on real numbers (e.g., $\leq$, $\geq$, $<$, $>$, $=$, $\neq$)³ are relations on $\mathbb{R}$. Notice that $\leq$, $\geq$, and $=$ contain all the pairs $\langle x, x \rangle$, that is $x \leq x$, $x \geq x$, and $x = x$. We say that $\leq$, $\geq$, and $=$ are *reflexive* (applying to oneself). In contrast, $<$, $>$, and $\neq$ *never* satisfy $x < x$ or $x > x$ or $x \neq x$, so they

³Equality $=$ is a relation here or whenever we restrict it to a particular set. The general $=$ between objects is more complicated because the collection of all objects we might compare is too big to be a set.

are *irreflexive* (not applying to oneself). Relations need not be either reflexive or irreflexive; they might contain $\langle x, x \rangle$ for some x but not others.

If for two numbers x and y , $x = y$, then we know $y = x$ as well, and similarly for $\neq$. We say that these relations are *symmetric*, the order does not matter. In contrast, if $x < y$, we know that y not less than x , and similarly for $>$. These are *asymmetric*. There is a third distinction: if $x \leq y$ and $y \leq x$, we know that $x = y$, and likewise if $x \geq y$ and $y \geq x$. We call these *antisymmetric* relations; symmetry implies identity.

For real numbers x, y, z , if we know $x \leq y$ and $y \leq z$, it follows that $x \leq z$. The relation $\leq$ is said to be *transitive*. The same holds for $\geq, <, >$, and $=$, but it does not hold for $\neq$: $x \neq y$ and $y \neq z$ does not preclude equality of x and z .

All of these properties apply to an arbitrary homogeneous binary relation.

Properties of Homogeneous Relations

Three key questions to ask about a binary relation $\mathcal{R}$ on a set $\mathcal{A}$:

- How does an object relate to itself?
 - $\mathcal{R}$ is **reflexive** if for all $a \in \mathcal{A}$, $a \mathcal{R} a$ or equivalently $\langle a, a \rangle \in \mathcal{R}$,
 - $\mathcal{R}$ is **irreflexive** if for all $a \in \mathcal{A}$, $a \not\mathcal{R} a$ or equivalently $\langle a, a \rangle \notin \mathcal{R}$,
- Does the order of objects affect the relation?
 - $\mathcal{R}$ is **symmetric** if $a \mathcal{R} a'$ implies $a' \mathcal{R} a$,
 - $\mathcal{R}$ is **asymmetric** if $a \mathcal{R} a'$ implies $a' \not\mathcal{R} a$, and
 - $\mathcal{R}$ is **antisymmetric** if $a \mathcal{R} a'$ and $a' \mathcal{R} a$ implies $a = a'$.
- Is the relation transitive?
 - $\mathcal{R}$ is **transitive** if $a \mathcal{R} a'$ and $a' \mathcal{R} a''$ implies $a \mathcal{R} a''$,

Puzzle 129. $2^{\mathcal{A}}$ is the *power set* of a set $\mathcal{A}$, the set of all its subsets. Explain why $\subseteq$ is a relation on $2^{\mathcal{A}}$ and indicate which of the above properties it has.

Example 16.2 Relations Induced by Curves

For a function $f: \mathbb{R}^2 \rightarrow \mathbb{R}$, let $\mathcal{C}_f = \{\langle x, y \rangle \in \mathbb{R}^2 \mid f(x, y) = 0\}$ be the set of points in the plane that solve an equation. Such sets include a variety of curves in the plane.

For instance, if $f(x, y) = x^2 + y^2 - 1$, then $\mathcal{C}_f = \{\langle x, y \rangle \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$,

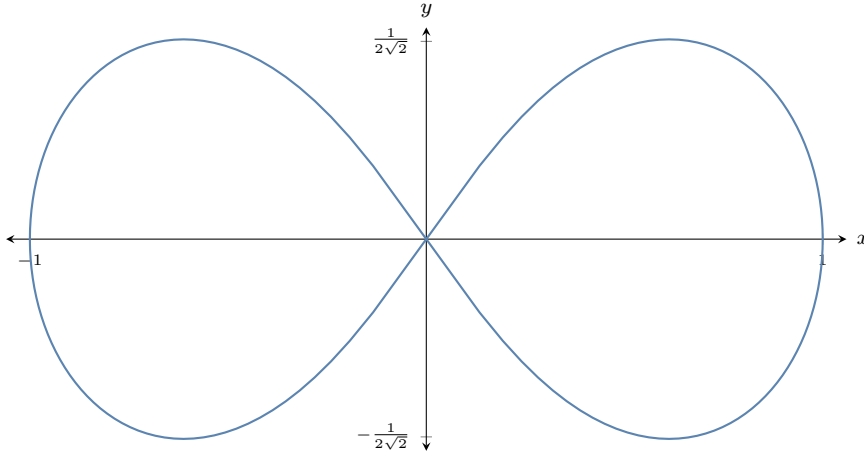


FIGURE 16.1. Bernoulli's Lemniscate.

the unit circle. If $f(x, y) = (x^2 + y^2)^2 - x^2 + y^2$, we get Bernoulli's lemniscate (Figure 16.1) $\mathcal{C}_f = \{ \langle x, y \rangle \in \mathbb{R}^2 \mid (x^2 + y^2)^2 = x^2 - y^2 \}$. And so forth.

We can view $\mathcal{C}_f$ as a relation on $\mathbb{R}$, where x and y are related if $\langle x, y \rangle$ solves the equation in f . This need not be a symmetric or reflexive or have any of the properties defined above, depending on f . Unlike with a function, a value x may be related to more than one y or to none. For instance, with the unit circle, $1/\sqrt{2}$ is related to both $1/\sqrt{2}$ and $-1/\sqrt{2}$.

Example 16.3 Set Partitions and Surjections

In Example 14.1, we saw that a surjection $f: \mathcal{S} \twoheadrightarrow \mathcal{P}$ represents a *partition* of the set $\mathcal{S}$ into mutually exclusive and collectively exhaustive parts $f^{-1}(\{p\})$.

Define a relation $\sim$ on $\mathcal{S}$, where $s \sim s'$ means that s and s' are in the same part, i.e., $f(s) = f(s')$. We then have:

- $f(s) = f(s)$ by definition for every $s \in \mathcal{S}$, so $s \sim s$. Hence, $\sim$ is reflexive.
- $f(s) = f(s')$ ($s \sim s'$) implies $f(s') = f(s)$ ($s' \sim s$). Hence, $\sim$ is symmetric.
- If $f(s) = f(s')$ and $f(s') = f(s'')$, then by the transitivity of equality, $f(s) = f(s'')$, so $s \sim s'$ and $s' \sim s''$ implies $s \sim s''$. Hence, $\sim$ is transitive.

So every partition of a set gives rise to a reflexive, symmetric, transitive relation on that set. Now let's go the other way and see that every reflexive, symmetric, transitive relation on a set gives rise to a partition of that set.

Suppose we have a relation $\sim$ on $\mathcal{S}$ that is reflexive, symmetric, and transitive. For any $s \in \mathcal{S}$, define $\mathcal{A}_s = \{s' \in \mathcal{S} \mid s \sim s'\}$, all the elements in $\mathcal{S}$ that are

related to s . The properties of the relation tell us that:

1. $s \in \mathcal{A}_s$ for all s because $\sim$ is reflexive;
2. $s \in \mathcal{A}_{s'}$ if and only if $s' \in \mathcal{A}_s$ for any s, s' because $\sim$ is symmetric; and
3. $s' \in \mathcal{A}_s$ implies $\mathcal{A}_{s'} = \mathcal{A}_s$ for any s, s' .

To see that this last statement is true, we will show separately $\mathcal{A}_{s'} \subseteq \mathcal{A}_s$ and $\mathcal{A}_s \subseteq \mathcal{A}_{s'}$ when $s' \in \mathcal{A}_s$.

Let $s'' \in \mathcal{A}_{s'}$ be arbitrary; then $s' \sim s''$. Because $\sim$ is transitive and $s \sim s'$ by assumption, we have $s \sim s'$ and $s' \sim s''$ and hence $s \sim s''$. So, $s'' \in \mathcal{A}_s$ and thus $\mathcal{A}_{s'} \subseteq \mathcal{A}_s$.

Now suppose $s'' \in \mathcal{A}_s$ is arbitrary, so $s \sim s''$ and thus $s'' \sim s$ by symmetry. Transitivity of $\sim$ and $s \sim s'$ implies that $s'' \sim s'$ and by symmetry $s' \sim s''$. So, $s'' \in \mathcal{A}_{s'}$ and hence $\mathcal{A}_s \subseteq \mathcal{A}_{s'}$. Taken together we have that $\mathcal{A}_s = \mathcal{A}_{s'}$.

Define $\mathcal{P} = \{\mathcal{A}_s \mid s \in \mathcal{S}\}$ and define $f(s) = \mathcal{A}_s$, which is a surjection by construction. The elements of $\mathcal{P}$ are pairwise disjoint and every element of $\mathcal{S}$ belongs to one element of $\mathcal{P}$, so this gives a partition of $\mathcal{S}$ induced by $\sim$.

The previous example motivates an important type of homogeneous binary relation.

An **equivalence relation** on a set $\mathcal{A}$ is a binary relation on $\mathcal{A}$ that is reflexive, symmetric, and transitive. As we have just seen, an equivalence relation induces a partition on $\mathcal{A}$; the parts of that partition are called **equivalence classes**. If $\sim$ is the equivalence relation, the equivalence class to which an $a \in \mathcal{A}$ belongs is often denoted $[a]_\sim$. The set of equivalence classes in $\mathcal{A}$ induced by $\sim$ is called a *quotient* with respect to $\sim$ and denoted $\mathcal{A}/\sim$ accordingly. The surjection function we constructed above that maps $a \in \mathcal{A}$ to $[a]_\sim$ is the *projection* onto this quotient.

For example, if we look at the set of ASCII strings, we can define a relation $s \sim s'$ to mean that strings s and s' have a common prefix of at least one character. We have $s \sim s$ because s equals itself entirely, so $\sim$ is reflexive. If $s \sim s'$, then s' and s have a common prefix, so $s' \sim s$ and $\sim$ is symmetric. Finally, if $s \sim s'$ and $s' \sim s''$, then the shorter of the common prefixes (which must be at least one character) is common to s and s'' , so $s \sim s''$ and $\sim$ is transitive. Hence, $\sim$ is an equivalence relation. The equivalence class $[\text{abacus}]_\sim$ contains all strings that start with a, with ab, with aba, with abac, with abacu, as well as abacus itself, and the projection maps a string to the collection of strings that share a prefix with it.

As another example, start with a simple, undirected graph without loops $G = \langle \mathcal{N}, \mathcal{E}, \varphi \rangle$,⁴ and define a relation $\circ\!\!\!\circ$ on the set of nodes $\mathcal{N}$ such that $n_1 \circ\!\!\!\circ n_2$ when $n_1 = n_2$ or there is an edge between n_1 and n_2 (that is, $\varphi^-(\{n_1, n_2\})$ is not empty). This relation is, by construction, reflexive and symmetric, but it need not be

⁴See Example 10.19.

transitive. We can construct a relation $\bullet\bullet$ that is the smallest reflexive, symmetric, and transitive relation that contains $\circ\circ$; this is called the **transitive closure**. We start by adding $n_1 \bullet\bullet n_3$ for every pair $n_1 \circ\circ n_2$ and $n_2 \circ\circ n_3$. (Because $\circ\circ$ is symmetric this adds every $n_3 \bullet\bullet n_1$ as well.) Then we repeat this operation again and again until no further pairs are added. The result is that $n \bullet\bullet n'$ whenever either $n = n'$ or there is a sequence of nodes from n to n' connected by an edge, i.e., $n = n_0, n_1, \dots, n_k = n'$ with $n_{i-1} \circ\circ n_i$ for every $i \in [1..k]$. (We will look at transitive closure in a different way when we compose relations below.) The relation $\bullet\bullet$ is then an equivalence relation; two nodes are equivalent in this sense if it is possible to get from one to the other by following a series of edges; so the equivalence classes are the *connected components* of the graph and the projection maps a node to its connected component. The undirected graph in Example 10.19, for instance has two connected components, one containing only node 14 and the other containing the other nodes.

In general, if $\sim$ is an equivalence relation on $\mathcal{A}$ with projection p , a $f: \mathcal{A} \rightarrow \mathcal{B}$ *respects* the equivalence classes if $a \sim a'$ implies that $f(a) = f(a')$. Then we can define a function $\tilde{f}: \mathcal{A}/\sim \rightarrow \mathcal{B}$ so that the following diagram commutes:

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\ p \downarrow & \nearrow \tilde{f} & \\ \mathcal{A}/\sim & & \end{array}$$

That is, we can express $f = \tilde{f} \circ p$ in terms of $\tilde{f}$ and p . Working with the quotient often lets us ignore irrelevant details. We treat the equivalence classes themselves as the objects of interest. For example, in Part I we have an equivalence relation on kinds, where two kinds are equivalent if they have the same canonical form. All our operations on kinds respect this equivalence relation, so we define those operations on the equivalence classes themselves.

We now continue with examples of more general relations.

Example 16.4 Properties of Integers

Properties, as we call unary relations, describe attributes an element of a set may have. The full embodiment of that property is just the set of elements that have it.

Is an integer prime? Is it even? Is it between 1 and some k ? The sets of integers with those properties (e.g., primes, evens, $[1..k]$) serve as witnesses for those properties.

```

def partial1(x: int):
    if x < 0:
        raise Exception("Does not return a value")
    return x - 2

def partial2(x: int) -> int:
    "Might never terminate."
    while x > 3:
        pass
    return x

```

FIGURE 16.2. Two partial functions expressed in Python.

In computer programming, a predicate is a function that returns a Boolean, and we might write

```

def is_even(x):
    return x % 2 == 0

```

as a test for evenness. But this is just the predicate associated with the property $2\mathbb{Z}$, the set of even numbers, as defined above.

We frequently use properties as “filters” to select items in sums, integrals, when building sets, and so forth. For example, when we write $\sum_{k=1}^{100} f(k)$, which is short-hand for $\sum_{k \in [1..100]} f(k)$, we are indicating that we should select only those values with the property $[1..100]$ to sum over. Combining simple properties can often be used to describe highly complex properties.

It is worth noting that, in general, an n -ary relation on $\mathcal{A}_1, \dots, \mathcal{A}_n$ is just a set $\mathcal{P} \subseteq \mathcal{A}_1 \times \dots \times \mathcal{A}_n$, and so we can view it equivalently as a unary relation (property) on $\mathcal{A}_1 \times \dots \times \mathcal{A}_n$.

Example 16.5 Partial Functions

As we have seen, the graph of a function $f: \mathcal{A} \rightarrow \mathcal{B}$ is a relation between $\mathcal{A}$ and $\mathcal{B}$. We say that this relation is *total* because every $a \in \mathcal{A}$ relates to some element of $\mathcal{B}$. We say that this relation is *univalent* because $a \in \mathcal{A}$ relates to *at most one* element of $\mathcal{B}$. These two properties – total and univalent – are the requirements of a function: it must return exactly one output for every possible input.

If we drop the first requirement but keep the second, we get a relation between $\mathcal{A}$ and $\mathcal{B}$ where only some $a \in \mathcal{A}$ relate to an element of $\mathcal{B}$ but those that do relate to exactly one element. We can think of such a relation as a **partial**

function. This behaves like a function *except* that it need not return a value for all possible inputs.

Partial functions are common in computer programming. For example, both of the Python functions in Figure 16.2 are partial. Neither returns a value for some possible inputs; the first by raising an error condition and the second by looping forever.

Given a partial function, we can in principle recover a (total) function by restricting its domain to the set of values for which it returns a value. But for a computational function, finding the values where a function terminates is a famously hard problem (the “Halting problem”) that cannot be solved efficiently in general.

Example 16.6 Collinear Points

Consider a ternary relation $\mathcal{L}$ on the set of points in space (e.g., $\mathbb{R}^3$) that three points are all on a line. Specifically, $\langle u, v, w \rangle \in \mathcal{L} \subseteq \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3$ when $w = v + a(u - v)$ for some real number a or $u = v$.

Notice that this relation has analogues of the properties of symmetry and reflexivity. If $\langle u, v, w \rangle \in \mathcal{L}$, then all three points are on the same line, and it does not matter the order in which we list them. Then $\langle w, v, u \rangle \in \mathcal{L}$ and similarly for any rearrangement of the points. The relation is symmetric under all orderings. Similarly, $\langle u, u, w \rangle \in \mathcal{L}$ for any w (including $w = u$) because two *distinct* points form a line. This is a kind of reflexivity, though a bit different than in the binary case.

However, if $\langle u, v, w \rangle \in \mathcal{L}$ and $\langle u, v, y \rangle \in \mathcal{L}$, we can only conclude that all four points are on the same line if $u \neq v$, so a notion of transitivity does not strictly hold.

Example 16.7 Cyclic Order. The hours on a clock face and the months of a year are not arranged in a line but on a circle. We cannot say that 4 comes after 12 or that 4 comes before 12; both are in some sense true. What we can say unambiguously are that “9, 12, and 4” and “November, January, May” occur with 12 after 9 then 4 after 12 and January after November then May after January. For values arranged in a circle, we can thus define a ternary relation $[a, b, c]$ to mean a, b, c that after a , we come to b before c .

To be consistent, we do require that $[a, b, c]$ implies $[b, c, a]$ and if $[a, b, c]$ then

not $[a, c, b]$. It also makes sense to require at least that $[a, b, c]$ and $[a, c, d]$ implies $[a, c, d]$. The latter two are analogues of the anti-symmetry and transitivity properties.

Example 16.8 Graph Paths. If we have a simple graph (directed or undirected) $G = \langle \mathcal{N}, \mathcal{E}, \varphi \rangle$, then we as have already seen, φ induces a binary relation on $\mathcal{N}$ indicating which nodes are adjacent in the graph. Let's call this relation $\circ\text{--}\circ$.

But there are other useful relations here. Start with a ternary relation on $\mathcal{N}$: $\llbracket n_1, n_2, n_3 \rrbracket$ meaning that $n_1 \circ\text{--}\circ n_2$ and $n_2 \circ\text{--}\circ n_3$; that is, there is a pair of edges from n_1 to n_3 through n_2 ;

We can extend this to an $k + 1$ -ary relation $\llbracket n_1, n_2, n_3, \dots, n_{k+1} \rrbracket$ meaning that $n_j \circ\text{--}\circ n_{j+1}$ for every $j \in [1..k]$; that is, there is *path*, a sequence of k edges connecting all these $k + 1$ nodes.

Example 16.9 Database Tables

A *relational database* is a data store that organizes information in a collection of n -ary relations called *tables*.

Each table describes an *entity*, and the domains of the relation defining the table are sets $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ describing various *attributes* of that entity. The data stored in the table is a collection of tuples $\langle a_1, \dots, a_n \rangle$ with $a_i \in \mathcal{A}_i$, with each tuple (a “row”) describing an instance of that entity.

For example, we might have a table **Movies** whose entity is nationally-released movies in the U.S. market. Attributes for this table might include **title**, **year of release**, **length** (min), **director**, **studio**, **writer**, **budget** (\$ million), **opening** (domestic, \$ million), and such, though in reality the attributes would be more complicated. Each row corresponds to an instance of the **Movies** entity, that is, a movie. Two rows might be like those shown in the top panel of Table 16.1, distinguishing between distinct releases of the same film. Notice that each attribute has a specific *type*, like date or string or minutes or dollars.

In practice, database tables are arranged a bit differently. First, one of the attributes, say $\mathcal{A}_1$, is specified as a *primary key* whose purpose is to uniquely identifies the entity instance associated with the tuple. This may be (and often is) as simple as using a unique integer in the primary key for every row. Second, having repeated strings like “George Lucas” to identify an attribute can be problematic. For instance, the different copies could get out of sync if, say, one

title	released	length	director	studio	writer	budget	opening
Star Wars	25 May 1977	121	George Lucas	Twentieth Century Fox	George Lucas	11	1.55
Star Wars	13 Aug 1982	121	George Lucas	Twentieth Century Fox	George Lucas	11	15.47

id	title	released	length	director	studio	writer	budget	opening
10274	Star Wars	25 May 1977	121	1423	20	1423	11	1.55
21499	Star Wars	13 Aug 1982	121	1423	20	1423	11	15.47

id	last_name	first_name	born
1423	Lucas	George	14 May 1944

TABLE 16.1. Two rows of a `Movies` database table, before and after normalization, and one row from the `People` table.

of them is changed. And if George Lucas were to become known by a different name, we would have to find and change all the copies in the table. So, data in the tables are *normalized* and any attribute that refers to an instance of a different entity is replaced by a pointer to another table, called a *foreign key*. With these changes, the above rows of our `Movies` table might look instead like the middle panel of Table 16.1. Here, the `id` attribute is the primary key, with each row's value unique, and the entries for `director` and `studio` are foreign keys that point into separate `People` and `Studios` tables (see bottom panel of Table 16.1), by giving the primary key of a row in that table

We can now query the data in our table by specifying conditions on the attributes. SQL, Structured Query Language, is often used for that task. For instance, the query

```
SELECT * from Movies where director = 1423;
```

will return all rows of `Movies` corresponding to movies George Lucas directed.

COMPOSITION OF RELATIONS. Like functions, binary relations can be composed, and this generalizes to n -ary relations for $n > 2$. To motivate this, let's consider the functions $f: \mathcal{A} \rightarrow \mathcal{B}$ and $g: \mathcal{B} \rightarrow \mathcal{C}$ and their composition $g \circ f: \mathcal{A} \rightarrow \mathcal{C}$ as *relations* (i.e., in terms of their graphs):

$$\begin{aligned}\text{graph}(f) &= \{ \langle a, f(a) \rangle \mid a \in \mathcal{A} \} \\ \text{graph}(g) &= \{ \langle b, g(b) \rangle \mid b \in \mathcal{B} \} \\ \text{graph}(g \circ f) &= \{ \langle a, g(f(a)) \rangle \mid a \in \mathcal{A} \} .\end{aligned}$$

We get the graph of $g \circ f$ by matching tuples $\langle a, f(a) \rangle$ and $\langle b, g(b) \rangle$ so that $f(a) = b$. Because f returns only one value, there is a single possible pair $\langle f(a), g(f(a)) \rangle$ that fits the bill. Having matched these two tuples, we can relate the first component (input) of the first to the second component (output) of the second, yielding $\langle a, g(f(a)) \rangle$.

We can do the same for any two binary relations on compatible sets, accepting any matches that occur (as there may be more than one).

If $\mathcal{F}$ is a binary relation on $\mathcal{A}$ and $\mathcal{B}$ and $\mathcal{G}$ is a binary relation on $\mathcal{B}$ and $\mathcal{C}$, then their composition $\mathcal{G} \circ \mathcal{F}$ (“ $\mathcal{G}$ after $\mathcal{F}$ ”) is the binary relation on $\mathcal{A}$ and $\mathcal{C}$ given by:

$$a (\mathcal{G} \circ \mathcal{F}) c \text{ if and only if there exists a } b \in \mathcal{B} \text{ with } a \mathcal{F} b \text{ and } b \mathcal{G} c. \quad (16.1)$$

We often drop the $\circ$ and just write $\mathcal{G}\mathcal{F}$.

In terms of the predicates, if f is the predicate for relation $\mathcal{F}$ and g is the predicate for relation $\mathcal{G}$, the predicate gf for relation $\mathcal{G}\mathcal{F}$ is

$$gf(a, b) = \bigvee_{b \in \mathcal{B}} (f(a, b) \wedge g(b, c)), \quad (16.2)$$

where $\vee$ is logical-or and $\wedge$ is logical-and as usual.

We have already seen an example of composition when we constructed the transitive closure of the adjacency relationship in a simple, undirected graph. We start with the relation $\circ\!\!\!\circ$; remember that, while we wrote this as an infix operator, it is actually a subset of $\mathcal{N} \times \mathcal{N}$ containing pairs of nodes. Using composition of relations, we define for each $k \in [1..)$

$$\circ\!\!\!\circ^k = \begin{cases} \circ\!\!\!\circ & \text{if } k = 1 \\ \circ\!\!\!\circ^{k-1} \circ \circ\!\!\!\circ & \text{if } k > 1 \end{cases} .$$

Then $n \circ\!\!\!\circ^k n'$ if nodes n, n' are connected by a path of k successive edges. The transitive closure $\bullet\!\!\!\bullet$ is given by

$$\bullet\!\!\!\bullet = \bigcup_{k=1}^{\infty} \circ\!\!\!\circ,$$

containing all pairs of nodes that are either equal or are connected by a path in the graph of one or more edges.

We can generalize the composition of binary relations to n -ary relations with $n > 2$, but the operation is more flexible. The result is called a (relation) **join**.

Let $\mathcal{R}$ be a relation of arity $m \geq 2$ and $\mathcal{S}$ be a relation of arity $n \geq 2$. Then a ℓ -join of $\mathcal{R}$ and $\mathcal{S}$ is an $(m + n - \ell)$ -ary relation join_{ℓ} where

$$\begin{aligned} \langle r_1, \dots, r_{m-\ell}, x_1, \dots, x_{\ell}, s_1, \dots, s_{n-\ell} \rangle &\in \text{join}_{\ell}(\mathcal{R}, \mathcal{S}) \\ &\text{if and only if} \\ \langle r_1, \dots, r_{m-\ell}, x_1, \dots, x_{\ell} \rangle &\in \mathcal{R} \text{ and } \langle x_1, \dots, x_{\ell}, s_1, \dots, s_{n-\ell} \rangle \in \mathcal{S}. \end{aligned} \tag{16.3}$$

This relation combines the instances in two relations when they have ℓ particular attributes in common.

Puzzle 130. Show how the composition of two binary relations $\mathcal{G}$ and $\mathcal{F}$ can be written as $\text{proj}_{13}^{\rightarrow}(\text{join}_1(\mathcal{F}, \mathcal{G}))$, where $\text{proj}_{13}^{\rightarrow}$ is the **image** of the projection function (see page 581.)

A more direct example applies to the database tables of Example 16.9. In that example, we saw that the data in one table (relation) can point to a row in another table (relation) via a foreign key. We use a *join* of two (or more) tables to get the combined attributes as though they were all in one table originally. For instance, in SQL we might do the following query on the tables of Example 16.9:

```
SELECT title, release, first_name, last_name, opening
  from Movies join People On Movies.director = People.id;
```

which would produce a table with rows like

title	released	first_name	last_name	opening
Star Wars	25 May 1977	George	Lucas	1.55
Star Wars	13 Aug 1982	George	Lucas	15.47

that include the director's name from the **People** table. The **Movies join People** indicates to use the join of the two relations and the **On Movies.director = People.id** clause says to use the director id as the common attribute. (The remaining attributes from the join are not shown for space; the **SELECT** does a projection onto the listed attributes only.)

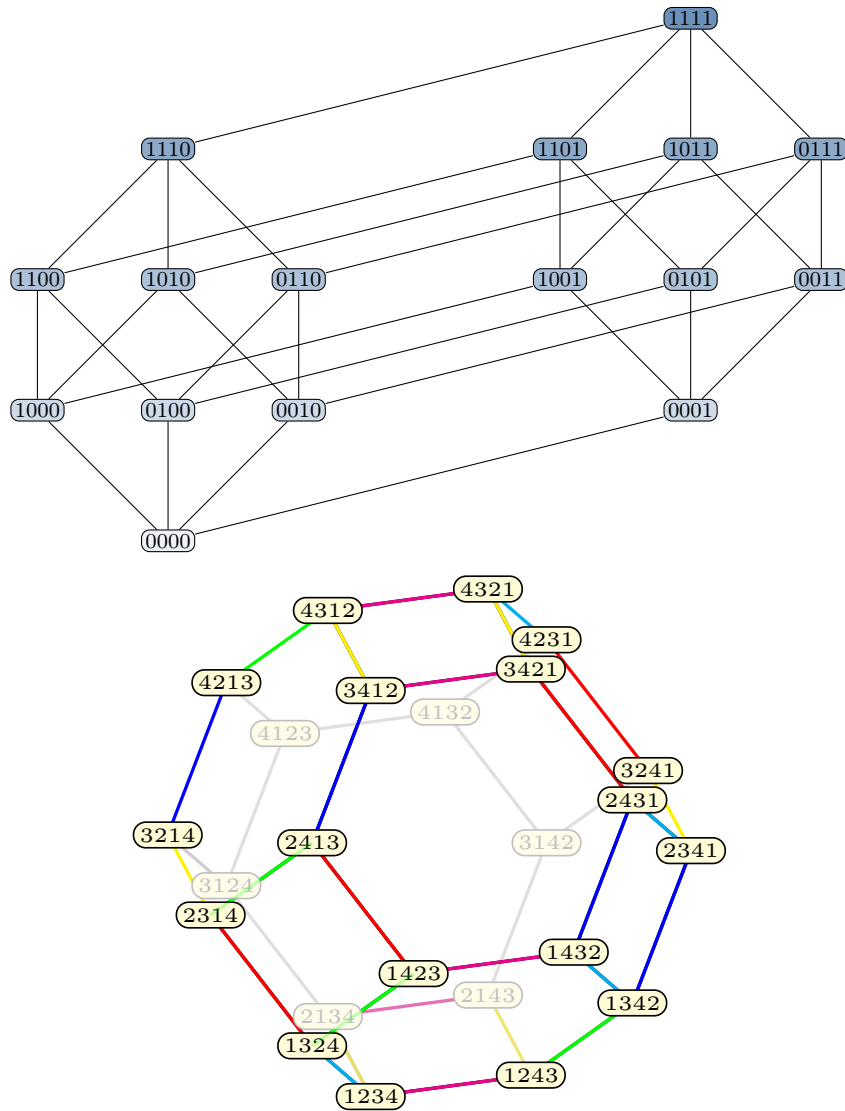


FIGURE 16.3. Representation of structure in two familiar sets as graphs. (Top) Subsets of $[1 \dots 4]$ and (Bottom) Permutations of $\langle 1, 2, 3, 4 \rangle$.

Algebraic Structures

17

Chapter

Contents

17.1	Associative Operators: Monoids, Semirings, and Beyond	702
17.2	Graphs and Matrices: An Enlightening Connection	731
17.3	Vector Spaces and Linearity	745
17.4	Orders	779

A background theme throughout this chapter has been the many ways that functions help us describe, probe, and transform the *structure* of mathematical objects. But so far, this notion of “structure” has been rather vague; it’s time to put some meat on those bones and enjoy the soup.

To whet our palettes, we begin with a deeper look at two familiar sets: the set $\mathcal{2}^{[1..n]}$ of all 2^n subsets of $[1..n]$ and the set $\mathcal{S}_n$ containing all $n!$ permutations of $\langle 1, 2, \dots, n \rangle$. We can view these sets as more than just collections of undifferentiated objects. Their elements have meaning, and the relationships among the elements constitute the structure of the set.

To make the structure of $\mathcal{2}^{[1..n]}$ and $\mathcal{S}_n$ more concrete, Figure 16.3 gives a visualization of both sets for $n = 4$. Each set is represented as a graph with one node per element; the graph’s edges and the spatial positions of the nodes help reveal the sets’ structures.

The top panel of Figure 16.3 depicts $\mathcal{2}^{[1..n]}$, where the subset in each node is labeled by a binary string whose digits indicate which elements are included in corresponding the subset of $[1..4]$. For example, 1111 is the whole set, 1010 is the set $\{1, 3\}$, 0001 is the set $\{4\}$, and 0000 the empty set. There is an edge between $\text{node}(\mathcal{A})$ and $\text{node}(\mathcal{B})$ if $\mathcal{A}$ and $\mathcal{B}$ differ by exactly one element, with the larger set at a higher vertical coordinate. Thus $\mathcal{A} \subset \mathcal{B}$ if and only if there is a path in the graph from $\text{node}(\mathcal{A})$ to $\text{node}(\mathcal{B})$ whose vertical coordinate only increases.¹ Subsets $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ have the same cardinality² if and only if their nodes lie at the same vertical coordinate. So each row of nodes gives the subsets of fixed cardinality, with the empty set on the bottom and the full set $[1..n]$ on top. Similarly, each set and its complement have nodes that lie on a line through the origin (center point of the figure), so subsets $\mathcal{A}_1$ and $\mathcal{A}_2$ are complements if and only if for some x, y , $\text{node}(\mathcal{A}_1)$ is at position $\langle x, y \rangle$ and $\text{node}(\mathcal{A}_2)$ is at position $\langle -x, -y \rangle$.

The arrangement of the nodes reveals more about the set as well. The 16 nodes in the graph divide into two cubes, with the cube on the left giving all the subsets not containing 4 and the cube on the right giving all the subsets containing 4. Each of the cubes is equivalent to the picture we would obtain for $\mathcal{2}^{[1..3]}$. In fact, the nodes can be divided into two (stretched) cubes in four different ways, with one cube in each pair giving the subsets containing a particular element and the other giving the subsets not containing that element.³ On each such cube, each pair of opposing faces split the subsets in the cube into those that contain and do not contain another element. So we can see how $\mathcal{2}^{[1..4]}$ is built from copies of $\mathcal{2}^{[1..3]}$ which in turn are build from copies of $\mathcal{2}^{[1..2]}$ which in turn are build from copies of $\mathcal{2}^{[1..1]}$.

¹This graph describes the order relation among subsets; it is called a *Hasse diagram*.

²Number of elements

³Look for instance at all the nodes containing $\{1\}$, which connect in a (stretched) cube. The remaining nodes are similarly connected and do not contain $\{1\}$.

The bottom panel of Figure 16.3 shows a three-dimensional polytope called the *permutahedron*, depicted on the two-dimensional page.⁴ Each possible permutation of $\langle 1, 2, 3, 4 \rangle$ corresponds to a node in the graph, a vertex of the polytope labeled with the permutation. There is an edge between two permutations' nodes if the permutations differ only by swapping one pair of numbers that *differ by 1*. Thus, $\langle 1, 2, 3, 4 \rangle$ and $\langle 1, 2, 4, 3 \rangle$ are connected by an edge because 4 and 3 are swapped and $4 - 3 = 1$, but $\langle 1, 2, 3, 4 \rangle$ is connected to neither $\langle 4, 2, 3, 1 \rangle$ nor $\langle 2, 4, 3, 1 \rangle$ because in the former $4 - 1 \neq 1$ and in the latter more than one swap is required. The color and direction of the edges corresponds to which pair of components is swapped, as shown. There are 14 faces; 6 are squares and 8 are hexagons.

The distance between the nodes in the picture roughly reflects the “distance” between the corresponding permutations in terms of the number of swaps required to go from one to the other. The nodes on each hexagonal face gives all the permutations that keep 1 fixed or all the permutations that keep 4 fixed, with parallel hexagonal faces fixing 1 and 4 in the same position. For instance, the hexagon facing out of the page (3412, 3421, 2431, 1432, 1423, 2413) contains all permutations that fix 4 in the second component, and its opposite face (4123, 4132, 3142, 2143, 2134, 3124) fixing 1 in the second component. Slicing the polytope by four planes parallel to a particular hexagonal face gives four hexagons (not all connected by edges) containing all permutations for which, respectively, 1, 2, 3, and 4 are fixed in the same component. So again we can see how $\mathcal{S}_4$ is built from various combinations of simpler permutations. The relationships depicted in the diagram thus capture the structure of $\mathcal{S}_4$.

Formally, a set $\mathcal{A}$ considered with some structure is a tuple $\langle \mathcal{A}, \dots \rangle$ whose first component is the set and the remaining components are objects (e.g., functions, relations, operators) that specify the structure. For instance, the set $\mathcal{2}^{[1..4]}$ with the structure in Figure 16.3 is a tuple $\langle \mathcal{2}^{[1..4]}, \text{pos}, \text{adjacent?} \rangle$, where $\text{pos}: \mathcal{2}^{[1..4]} \rightarrow \mathbb{R}^2$ maps each subset to the position of its node and $\text{adjacent?}: \mathcal{2}^{[1..4]} \times \mathcal{2}^{[1..4]} \rightarrow \mathbb{2}$ returns 1 if a pair of subsets differ by one element and 0 otherwise. The functions⁵ **pos** and **adjacent?** describe the graph in the diagram and represent the relationships described above. In practice, when the structure is understood from context, we refer to the set and treat the underlying tuple as implicit. For example, a Cartesian product of two arbitrary sets $\mathcal{A}_1 \times \mathcal{A}_2$ has a default structure $\langle \mathcal{A}_1 \times \mathcal{A}_2, \text{proj}_1, \text{proj}_2 \rangle$ where the projection functions **proj**₁ and **proj**₂ extract the components of the product and thus tell us how a particular element is built from the elements of $\mathcal{A}_1$ and $\mathcal{A}_2$: $p \in \mathcal{A}_1 \times \mathcal{A}_2 = \langle \text{proj}_1 p, \text{proj}_2 p \rangle$. An equivalent way to describe the structure on this

⁴The coordinate of permutahedron vertex $\langle i, j, k, \ell \rangle$, for instance, is $\langle i, j, k, \ell \rangle$. Even though there are four dimensions to each coordinate, this is a three-dimensional figure because all the coordinates of each vertex add up to 10.

⁵**adjacent?** is equivalent to a binary relation on $\mathcal{2}^{[1..4]}$.

sets is via two binary relations on the product $\sim_1$ and $\sim_2$, where

$$p \sim_1 q \text{ if and only if } \text{proj}_1(p) = \text{proj}_1(q) \quad (17.1)$$

$$p \sim_2 q \text{ if and only if } \text{proj}_2(p) = \text{proj}_2(q). \quad (17.2)$$

Similarly, a Cartesian product of more than two sets $\mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_n$ has default structure $\langle \mathcal{A}_1 \times \mathcal{A}_2, \text{proj}_1, \text{proj}_2, \dots, \text{proj}_n \rangle$ giving the projection functions onto each component of the n -tuple or equivalently n binary relations.

Given two sets $\mathcal{A}$ and $\mathcal{B}$ with structure and a function $f: \mathcal{A} \rightarrow \mathcal{B}$, a frequently useful question is how the structure of $\mathcal{A}$ mapped under f relates to the structure of $\mathcal{B}$, if at all. An important special case is when the function $f: \mathcal{A} \rightarrow \mathcal{B}$ *preserves* the structure of $\mathcal{A}$ in $\mathcal{B}$: if a relation $\mathcal{R}$ is part of the structure on $\mathcal{A}$, there is a relation $\mathcal{S}$ in the structure on $\mathcal{B}$ such that

$$\langle a_1, a_2, \dots, a_n \rangle \in \mathcal{R} \implies \langle f(a_1), f(a_2), \dots, f(a_n) \rangle \in \mathcal{S} \quad (17.3)$$

That is, elements of $\mathcal{A}$ related by structure are still related after mapping under f . We will see many examples of this below, but to make it more concrete immediately, consider for a fixed element $z \in \mathcal{A}_3$, the $f_z: \mathcal{A}_1 \times \mathcal{A}_2 \rightarrow \mathcal{A}_1 \times \mathcal{A}_2 \times \mathcal{A}_3$ where

$$f_z(p) = \langle \text{proj}_1(p), \text{proj}_2(p), z \rangle. \quad (17.4)$$

With the default structures on these sets, notice that for $p, q \in \mathcal{A}_1 \times \mathcal{A}_2$, $\text{proj}_1(p) = \text{proj}_1(q)$ implies $\text{proj}_1(f_z(p)) = \text{proj}_1(f_z(q))$.⁶ Similarly, $\text{proj}_2(p) = \text{proj}_2(q)$ implies $\text{proj}_2(f_z(p)) = \text{proj}_2(f_z(q))$ and by definition, for all $p, q \in \mathcal{A}_1 \times \mathcal{A}_2$, $\text{proj}_3(f_z(p)) = \text{proj}_3(f_z(q))$. Hence, the structure on $\mathcal{A}_1 \times \mathcal{A}_2$ is preserved by the function f_z , and indeed, this range of this function is essentially a *copy* of $\mathcal{A}_1 \times \mathcal{A}_2$ embedded in the larger set.

⁶These two uses of proj_1 are *different* functions, the first on $\mathcal{A}_1 \times \mathcal{A}_2$ and the second on $\mathcal{A}_1 \times \mathcal{A}_2 \times \mathcal{A}_3$.

A function that preserves structure in this way is called a **homomorphism**, a word that derives from “homo” and “morphism” for “same form”, or “same structure.” As we have seen earlier, in Example 13.11 for instance, homomorphisms can be “lossy,” giving a compressed or limited picture of one object in another context. In the previous example, f_z is a homomorphism from $\mathcal{A}_1 \times \mathcal{A}_2$ to $\mathcal{A}_1 \times \mathcal{A}_2 \times \mathcal{A}_3$ which gives a picture of the former inside the latter, and each choice of $z \in \mathcal{A}_3$ gives a different homomorphism. An important special case occurs when a homomorphism is a bijection whose inverse function is also a homomorphism. We call this an **isomorphism**, which we can loosely translate as “equivalent structure.” An isomorphism is a structure-preserving

function that is invertible, making the two sets interchangeable for many practical purposes.

As an example of this distinction, consider two sets of tuples: $\mathcal{A}_1 \times \mathcal{A}_2$ and $\mathcal{A}_2 \times \mathcal{A}_1$, where $\mathcal{A}_1 \neq \mathcal{A}_2$. As sets $\mathcal{A}_1 \times \mathcal{A}_2$ and $\mathcal{A}_2 \times \mathcal{A}_1$ are *not equal*, but the distinction between them only lies in how we order the two sets in the product. The function

$$s = \text{permute}_{21} \quad (\mathcal{A}_1 \times \mathcal{A}_2 \rightarrow \mathcal{A}_2 \times \mathcal{A}_1)$$

swaps the order of the elements and has inverse

$$s^{-1} = \text{permute}_{21} \quad (\mathcal{A}_2 \times \mathcal{A}_1 \rightarrow \mathcal{A}_1 \times \mathcal{A}_2).$$

Both s and s^{-1} are homomorphisms. One way to see this is that the following diagram commutes for $i \in \{1, 2\}$

$$\begin{array}{ccc} \mathcal{A}_1 \times \mathcal{A}_2 & \xrightarrow{s} & \mathcal{A}_2 \times \mathcal{A}_1 \\ & \searrow \text{proj}_i & \downarrow \text{proj}_{3-i} \\ & & \mathcal{A}_i \end{array}$$

(Replacing s with s^{-1} just reverses the top, horizontal arrow giving an analogous diagram.) Or as equations,

$$\begin{aligned} \text{proj}_1(\langle a, b \rangle) &= \text{proj}_2(s(\langle a, b \rangle)) \\ \text{proj}_2(\langle a, b \rangle) &= \text{proj}_1(s(\langle a, b \rangle)), \end{aligned}$$

which can also be written in terms of s^{-1} . Thus, although they are not equal, $\mathcal{A}_1 \times \mathcal{A}_2$ and $\mathcal{A}_2 \times \mathcal{A}_1$ are *isomorphic* (with both s and s^{-1} as isomorphisms). In terms of the structure that defines the tuples, the two sets are essentially interchangeable. As long as we are consistent, it does not matter which we use in practice. We will see more concrete examples below.

In this section, we will look at several different kinds of structure and the functions that preserve them. These provide us with patterns that match in a tremendous variety of situations and give us powerful tools for understanding disparate types of objects. We often describe structure using *algebraic laws*, which are functions or relations that describe properties and relationships among objects.

As a start, let's consider the familiar operation of addition. This a function $+: \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ defined for various sets $\mathcal{A}$, including the real and natural numbers,

that we write as an infix operator. The function comes equipped with something extra: algebraic laws that govern its use, including

$$a + b = b + a \quad (\text{Commutativity}) \quad (17.5)$$

$$a + (b + c) = (a + b) + c \quad (\text{Associativity}) \quad (17.6)$$

$$a + 0 = a \quad (\text{Identity/Unit}) \quad (17.7)$$

$$a + (-a) = 0. \quad (\text{Inverse}). \quad (17.8)$$

Variants of these four particular laws show up quite often. An operator is **commutative** when you get the same output after swapping the order of the arguments. An operator is **associative** if when applying it two or more times consecutively, we get the same result for any grouping of the calculations. So we can write $a + b + c$ unambiguously in this case. The third law requires the existence of an “**identity element**”⁷ (0 for addition); when it is plugged into one argument of the binary operator, the operation equals id as a function of the other argument. The last law here posits that for each value a , an **inverse** element $-a$ exists so that the sum of the two gives the identity element. If we apply an operator to a value and its **inverse**, we get the identity element. All four of these laws hold for the addition of real numbers, but they are so familiar that we might not think much about them.

⁷Other names sometimes used for identity element are “unit element” or “neutral element”.

Multiplication, division, subtraction, exponentiation all have laws that govern them, such as $2^{m+n} = 2^m 2^n$. And the laws from different operators often interact. For instance, the distributive law $a(b + c) = ab + ac$ and the annihilation law $a0 = 0$ entangle addition and multiplication.

Algebraic laws are one way to give structure to a set. A set is just a collection of objects in and of itself; it is how those objects relate to each other that gives them meaning. For instance, the $+$ operator above can be defined by a 3-ary relation $\mathcal{P}$ on $\mathcal{A}$ where $\langle a, b, c \rangle \in \mathcal{P}$ if and only if $a + b = c$. Similarly for example, the symmetric binary relation $\mathcal{C}$ on $\mathcal{A}$ for which $\langle a, b \rangle \in \mathcal{C}$ when $a + b = b + a$ describes the commutativity property when $\mathcal{C} = \mathcal{A} \times \mathcal{A}$. Numbers relate in other ways as well, such as ordering properties, sign and magnitude, and distance; all of these are governed by algebraic laws.

Much of our mathematical analysis can be seen as the study of sets equipped with some structure, and of functions between sets that preserve that structure, i.e., homomorphisms. In the case of arithmetic, it is so familiar as to be implicit, but once you see it, you realize it is ubiquitous.

Let’s consider a different type of object in this light: the set $\mathbb{R}^3$ whose elements

are 3-tuples of numbers. We might reason that since we can add numbers together, if we have two such tuples then we can add them together component by component. This gives us an operator, which we'll again call $+$, defined by $+: \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$ with:

$$\langle a_1, a_2, a_3 \rangle + \langle b_1, b_2, b_3 \rangle = \langle a_1 + b_1, a_2 + b_2, a_3 + b_3 \rangle.$$

This is a function that takes two 3-tuples and returns a new 3-tuple; for example, $\langle 1, 0, 3 \rangle + \langle 4, 2, 5 \rangle = \langle 5, 2, 8 \rangle$. If we define $\mathbf{0} = \langle 0, 0, 0 \rangle$, then for $u, v, w \in \mathbb{R}^3$:

$$\begin{aligned} u + v &= v + u \\ u + (v + w) &= (u + v) + w \\ v + \mathbf{0} &= v, \end{aligned}$$

which follows directly from the corresponding laws for numeric addition. We can also scale all the components of a tuple by any number c , giving scaling and negation operators that we denote with juxtaposition and a prefix “-”:

$$\begin{aligned} c\langle a_1, a_2, a_3 \rangle &= \langle ca_1, ca_2, ca_3 \rangle \\ -\langle a_1, a_2, a_3 \rangle &= \langle -a_1, -a_2, -a_3 \rangle. \end{aligned}$$

It follows that

$$v + (-v) = \mathbf{0}.$$

The values like c in this “scaling law” are thus called *scalars*. Taken together, these equations yield *the same laws* for $+$ on $\mathbb{R}^3$ as for addition with numbers, though with a different (though related) identity element, and a new scaling operation with its own law above. Endowed with these operators and algebraic laws, the set of tuples $\mathbb{R}^3$ becomes a *space*⁸ of **vectors**. We can do the same for tuples of any length, which gives us the structure of a “vector space” on every $\mathbb{R}^n$.

Consider what we get here for 1-tuples of numbers. the addition laws looks the same as for the addition of numbers with identity element $\langle 0 \rangle$, and the scaling laws look the same as our familiar laws for numeric multiplication. (The distributive and annihilation properties hold here as well, though we did not write them down above.) The function $f: \mathbb{R}^1 \rightarrow \mathbb{R}$ defined by $f(\langle x \rangle) = x$ is invertible, with inverse

⁸In mathematics, a *space* is one name for a set endowed with some additional structure.

$f^{-1}(x) = \langle x \rangle$. This also preserves our structure on $\mathbb{R}^1$

$$f(\langle x \rangle + \langle y \rangle) = x + y$$

$$f(c\langle x \rangle) = cx$$

$\dots,$

as does the inverse function in the other direction, and hence is an isomorphism. The set of 1-tuples of real numbers is *isomorphic* to the set of real numbers, and for practical purposes, we can elide the distinction between them, exactly as we decided to do in Chapter 15.

In this section, we will look at three different kinds of structure (and their homomorphisms) that arise so frequently that we gain a lot of power by recognizing them. These are: associative operators, vector spaces, and order relations.

17.1 Associative Operators: Monoids, Semirings, and Beyond

The familiar arithmetic on numbers is governed by quite a few different laws, so we begin here with a much simpler structure that will serve as a building block. We start by considering sets that are equipped with a single binary operator that has only two requirements: it is *associative* and has an *identity element*. A familiar example will set the stage.

Example 17.1 Lists

Let $\mathcal{A}$ be a non-empty set. Then $\text{List}(\mathcal{A})$, commonly denoted $\mathcal{A}^*$, is the set whose elements are *finite*-length tuples of elements from $\mathcal{A}$. For example, with $\mathbb{2} := \{0, 1\}$, $\text{List}(\mathbb{2})$ contains all finite-length binary strings

$$\langle \rangle, \langle 0 \rangle, \langle 1 \rangle, \langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 0, 0, 0 \rangle, \dots,$$

where as usual $\langle \rangle$ is the empty list. Two analogues in programming languages are: $\text{List}(\mathbb{Z})$ as the set of integer arrays, and if $\mathcal{A}$ is the set of ASCII characters, $\text{List}(\mathcal{A})$ as the set of ASCII strings.

We can endow the set $\text{List}(\mathcal{A})$ with structure by defining a binary operator $:: : \text{List}(\mathcal{A}) \times \text{List}(\mathcal{A}) \rightarrow \text{List}(\mathcal{A})$ that concatenates two lists. For example, $\langle 0, 0 \rangle :: \langle 1, 0, 1, 1 \rangle = \langle 0, 0, 1, 0, 1, 1 \rangle$. This operator satisfies three useful algebraic

We could define functions on $\text{List}(\mathcal{A})$ that correspond to the various list and string manipulations that we often use in programming.

laws, which you can confirm for yourself:

$$\langle \rangle :: \ell = \ell \quad (17.9)$$

$$\ell :: \langle \rangle = \ell \quad (17.10)$$

$$\ell_1 :: (\ell_2 :: \ell_3) = (\ell_1 :: \ell_2) :: \ell_3. \quad (17.11)$$

where $\ell, \ell_1, \ell_2, \ell_3 \in \text{List}(\mathcal{A})$. The first two equations tell us that the empty list is an **identity element** for $::$, like 0 is for addition. The third equation tells us that $::$ is **associative**; when performing a series of pairwise concatenations, it does not matter which pair we concatenate first. So we can write $\ell_1 :: \ell_2 :: \ell_3$ unambiguously, and similarly for any number of terms. Notice that $::$ is *not* commutative; $\langle 1 \rangle :: \langle 1, 0 \rangle \neq \langle 1, 0 \rangle :: \langle 1 \rangle$.

Puzzle 131. What is the difference between $\text{List}(\mathbb{R})$ and the set of sequences $\text{Seq}(\mathbb{R})$ as defined on page 562?

MONOIDS. A common situation: we have a collection of things and a way of combining two things in the collection that always produces yet another thing in the collection. There are only two provisos. First, if we combine multiple things – necessarily pairwise – we produce the same thing ultimately no matter which pair we combine first.⁹ Second, there is a special “empty” or “neutral” thing that when combined with any other thing leaves the other thing unchanged. The lists in the previous example fits this template, and as we will see, a great variety of “things” and combination operations do as well.

⁹Note that the *order* of the things *within* any particular pair may matter.

A **monoid** $\langle \mathcal{M}, \diamond, e \rangle$ consists of a set $\mathcal{M}$ equipped with a binary operator $\diamond: \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$ and a special element $e \in \mathcal{M}$ that satisfy:

1. $\diamond$ is associative: $m_1 \diamond (m_2 \diamond m_3) = (m_1 \diamond m_2) \diamond m_3$ for every $m_1, m_2, m_3 \in \mathcal{M}$, and
2. e is an **identity element**: $e \diamond m = e = m \diamond e$ for every $m \in \mathcal{M}$.

$\diamond$ need not be commutative, but if it is we call this a *commutative monoid*.

The monoid is the set *with* the extra structure of the operator and identity element, $\langle \mathcal{M}, \diamond, e \rangle$, but colloquially, it is common to just refer to $\mathcal{M}$ as a monoid when the operator and identity element are understood.

The associativity property implies that we can write $m_1 \diamond m_2 \diamond \cdots \diamond m_n$ for any n because any order of resolving the pairwise $\diamond$ leads to the same result. Associativity

lets us build a complex thing with the operation without worrying about *how* we built it. The recipe for building $m_1 \diamond m_2 \diamond \cdots \diamond m_n$ is just that: which ingredients went into the pot and in which order. (With a commutative monoid, we wouldn't even care about order.) Without associativity, however, we'd have to keep track the steps we used to build it, which pairs we combined in which order, giving exponentially many (in fact $\frac{1}{n} \binom{2n-2}{n-1}$) different recipes for each tuple of ingredients $\langle m_1, m_2, \dots, m_n \rangle$. Monoids capture a basic notion of composability, and associativity lets us abstract away somewhat from the internal details of the things we are combining.

The term “identity element” stems from the fact that both the functions $\langle m \rangle \mapsto m \diamond e$ and $\langle m \rangle \mapsto e \diamond m$ equal the identity function id . The identity element is *unique*. If e' were some other element that had the properties of an identity element, then $e' = e \diamond e' = e$.

In the previous example, we saw that $\langle \text{List}(\mathcal{A}), ::, \langle \rangle \rangle$ is a monoid. This describes several data containers we use in programming such as lists, arrays, and strings. Numbers and Boolean values also have monoidal structure. For instance, the natural numbers with addition (or multiplication) as the operator and Booleans with logical-and (or logical-or or logical-exclusive-or) as the operator. Other examples abound. A computer scientist might describe the monoid properties as an API¹⁰ diverse types of objects can support; we can work in a unified way with any type of objects that support the interface.

¹⁰Application Programming Interface

Puzzle 132. Explain why these are monoids, specifying identity elements if missing:

1. $\langle \mathbb{N}, +, 0 \rangle$
2. $\langle \mathbb{N}, \cdot, 1 \rangle$
3. The set of Booleans $\mathbb{B} = \{\perp, \top\}$ with each of the operators $\wedge$, $\vee$, and ∇ . What are the corresponding identity elements?
4. $\langle \mathbb{N}, \max, ??? \rangle$
5. $\langle [-\infty, \_], \max, ??? \rangle$. Note $-\infty$ is part of the set.
6. $\langle [_, \infty], \min, ??? \rangle$. Note ∞ is part of the set.

Example 17.2 Dictionaries. Many programming languages define a type of associative array in which we can look up a value associated with a type of *key*, often strings. These have various names in different languages (e.g., maps, tables, hash maps, hash tables). In Python for instance, these are called *dictionaries*, e.g.,

```
dict1 = { 'a': 10, 'b': 20, 'c': 248, 'oh!': 7 }
dict2 = { 'a': -10, 'x': 100, 'c': 4096, 'z': 0 }
dict3 = {}
```

and we can look up the value associated with a key with, for instance, `dict1['a']` which gives 10 or `dict2['z']` which gives 0. `dict3` is an empty dictionary with no keys or values.

At times, we want to *merge* two dictionaries to produce a new dictionary with combined sets of keys and reasonably assigned values for each key. A standard merging strategy is to take as keys the union of keys for both dictionaries, and for the values of any keys in both, use the second one specified. In Python, the operator `|` merges dictionaries, so we have:

```
dict2 | dict1 = {'a': 10, 'x': 100, 'c': 248, 'z': 0, 'b': 20, 'oh!': 7}
dict1 | dict2 = {'a': -10, 'b': 20, 'c': 4096, 'oh!': 7, 'x': 100, 'z': 0}
dict1 | dict3 = {'a': 10, 'b': 20, 'c': 248, 'oh!': 7}
```

This operation is associative, though not commutative, and merging with empty dictionary on either side acts like `id`. The set of dictionaries is thus a monoid with merging as the operator and the empty dictionary as the identity element. This monoid is *not* commutative because the merging strategy distinguishes between the dictionaries given first or second in prioritizing the values of shared keys.

An application of the previous monoid arises when we are configuring software, in any of several ways: configuration files, app settings/preferences, options for command-line programs, and various system setup “wizards.” Like a dictionary, the configuration has a set of keys, describing the available options and values giving the settings for each option. The configuration/settings/options can be drawn from various sources: built-in defaults, system defaults, site preferences, user preferences, and particular choices made at runtime. The settings that are ultimately used by the program are obtained by merging these dictionaries.

Example 17.3 Subsets. Let $\mathcal{X}$ be a set and $2^{\mathcal{X}}$ its power set, i.e., the set containing all its subsets. Both union ($\cup$) and intersection ($\cap$) are associative operators on $2^{\mathcal{X}}$. For instance, $\mathcal{A} \cup (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cap \mathcal{C}$ because the combined set contains the elements of all three constituents.

Notice that for any $\mathcal{A} \subseteq \mathcal{X}$

$$\mathcal{A} \cup \{\} = \mathcal{A}$$

$$\{\} \cup \mathcal{A} = \mathcal{A}$$

and

$$\mathcal{A} \cap \mathcal{X} = \mathcal{A}$$

$$\mathcal{X} \cap \mathcal{A} = \mathcal{A}.$$

So, $\langle 2^{\mathcal{X}}, \cup, \{\} \rangle$ and $\langle 2^{\mathcal{X}}, \cap, \mathcal{X} \rangle$ are both monoids.

Example 17.4 Multisets

Example 10.14 introduced the idea of multisets, or evocatively *bags*. A multiset on an base set $\mathcal{A}$ is like a set where elements can be repeated; formally, it is a function $\mathcal{A} \rightarrow \mathbb{N}$ that returns the number of copies of a value that are contained in the multiset.

Following up on the last two examples, multisets form a monoid, both as a set of mathematical objects and as a data structure in programming.

In the mathematical case, let $\mathcal{M}(\mathcal{A})$ be the set of all multisets on base set $\mathcal{A}$. Using the multiset sum ($m = m_1 + m_2$ as functions) as the operation and the zero function ($m = \text{const}_0$) as identity element makes $\mathcal{M}(\mathcal{A})$ a monoid. Associativity follows from the associativity of addition.

In a programming context, we can represent a multiset as a dictionary whose keys are in the base set and whose values are natural numbers. Summing the multisets corresponds to merging the dictionaries with a merging strategy that adds the values of any keys shared by the two dictionaries being merged.

Example 17.5 Merge Sorts

Consider the set $\text{List}(\mathbb{R})$ (or in general, any $\text{List}(\mathcal{A})$ that is ordered) with an operator $\text{merge}: \text{List}(\mathbb{R}) \times \text{List}(\mathbb{R}) \rightarrow \text{List}(\mathbb{R})$, where $\text{merge}(\ell_1, \ell_2)$ is a list with all the components of ℓ_1 and ℓ_2 listed in increasing order. This an associative operation with $\langle \rangle$ as the identity element, so $\langle \text{List}(\mathbb{R}), \text{merge}, \langle \rangle \rangle$ is a monoid on $\text{List}(\mathbb{R})$ different from the monoid on the same set in Example 17.1.

Example 17.6 Shuffles. Let $\mathcal{D}$ be the set of $52!$ orderings of a standard deck of cards. Every shuffle acts on $\mathcal{D}$ by permuting the ordering in some way to produce another element of $\mathcal{D}$.

Let $\mathcal{S}$ be the set of shuffles. The “do nothing” shuffle leaves the deck unchanged. We can combine two shuffles by doing one shuffle, then the other. This operation is not commutative, but it is associative (for the same reason function composition is). Hence, shuffles form a monoid.

Puzzle 133. Function Composition as a Monoid.

Show that for any set $\mathcal{A}$, the set $\mathcal{A} \rightarrow \mathcal{A}$ of functions that map $\mathcal{A}$ to itself forms a *monoid* with composition $\circ$ as the operator and id as the identity element.

One benefit recognizing a monoid structure is that we can use it to speed or improve computations in various ways. A helpful trick along these lines is fast computation of powers.

Example 17.7 Exponentiation by Squaring

If $\langle \mathcal{M}, \diamond, e \rangle$ is a monoid, we can define natural powers of any $m \in \mathcal{M}$ by $m^0 = e$ and $m^{n+1} = m \diamond m^n$, giving

$$m^n = \overbrace{m \diamond \cdots \diamond m}^{n \text{ times}}.$$

Associativity of the monoid operation gives us for $n \in [1..)$

$$\begin{aligned} m^{2n} &= (m \diamond m)^n \\ m^{2n+1} &= m \diamond (m \diamond m)^n. \end{aligned}$$

This gives us an algorithm

```
fast_monoid_power(x, n: natural):
    if n is 0:
        return IDENTITY_ELEMENT
    if n is even:
        return fast_monoid_power(MONOID_OP(x, x), n / 2)
    return MONOID_OP(x, fast_monoid_power(MONOID_OP(x, x), (n - 1) / 2))
```

Applying this to the $\langle \mathbb{N}, \cdot, 1 \rangle$, we can compute $k^{24} = (k^2)^{12} = ((k^2)^2)^6 = (((k^2)^2)^2)^3 = ((k^2)^2)^2(((k^2)^2)^2)^2$, which requires only five applications of the monoid operation. (The last step only requires 2 as $((k^2)^2)^2$ is reused.) Similarly, in $\langle \mathbb{N}, +, 0 \rangle$, $24k = (2 \cdot 2 \cdot 2k)(2 \cdot 2 \cdot 2 \cdot 2k)$. The same applies to any monoid: computing the n th “power” in roughly $\log(n)$ operations. This includes joining string, matrix powers, polynomial powers, rotations of shapes, $\dots$, and in Part I, the kind of an independent join transformed by a so-called *monoidal statistic*, $\psi(K \star n)$. With just a bit of infrastructure, all these computations can be done with essentially *the same code*.

Consider, for instance, the monoid on $\mathbb{N} \times \mathbb{N}$ with operation $\diamond$ given by

$$\langle a, b \rangle \diamond \langle m, n \rangle = \langle am + bn, an + b(m + n) \rangle \quad (17.12)$$

and with identity element $\langle 1, 0 \rangle$. That this is associative can be checked (somewhat tediously). Then, $\text{proj}_2 \langle 0, 1 \rangle^n$ gives the n th [Fibonacci number](#), which our `fast_monoid_power` algorithm gives us in a logarithmic number of operations.

Example 17.8 Parallel Computation

Suppose that $f: \mathcal{X} \rightarrow \mathcal{M}$, where $\mathcal{X}$ is a set of possible data values and $\langle \mathcal{M}, \diamond, e \rangle$ is a monoid that represents the results of some computation, and $\psi: \mathcal{M} \rightarrow \mathcal{Y}$. Given a large data set $\langle x_1, x_2, \dots, x_n \rangle$ of values in $\mathcal{X}$, we want to compute a summary result

$$\psi(f(x_1) \diamond f(x_2) \diamond \dots \diamond f(x_n)). \quad (17.13)$$

The role of ψ is to convert the This is the situation, for instance, if we want to compute summary statistics of a large data set, as in [Example 17.14](#).

A calculation of this form can be *parallelized* using the properties of the monoid operation. If we have s machines available, we can initialize s preliminary results as $r_1 = e, r_2 = e, \dots, r_s = e$. Typically s is much less than n . We can then pass the data in disjoint chunks to the s machines. For each data point x_i that machine j receives, it updates $r_j \leftarrow r_j \diamond f(x_i)$. If machine j receives data $x_{i_1}, x_{i_2}, \dots, x_{i_k}$, that machine computes $r_s = r_s$. When all the data have been processed, the machines pass their results to a designated machine which computes $\psi(r_1 \diamond r_2 \diamond \dots \diamond r_s)$ giving the final result. This is the essence of the “Map-Reduce” framework.

Example 17.9 Folds

Referring to Example 17.1, consider $\text{List}(\mathcal{M})$ where $\langle \mathcal{M}, \diamond, e \rangle$ is a monoid. Define the function $\text{fold}: \text{List}(\mathcal{M}) \rightarrow \mathcal{M}$ by

$$\text{fold}(\langle \rangle) = e \quad (17.14)$$

$$\text{fold}(\langle m \rangle :: \ell) = m \diamond \text{fold}(\ell). \quad (17.15)$$

It follows that

$$\text{fold}(\langle m_1, m_2, \dots, m_n \rangle) = m_1 \diamond m_2 \diamond \dots \diamond m_n, \quad (17.16)$$

and thus that

$$\text{fold}(\ell_1 :: \ell_2) = \text{fold}(\ell_1) \diamond \text{fold}(\ell_2). \quad (17.17)$$

We have, for instance:

1. For $\mathcal{M} = \langle \mathbb{N}, +, 0 \rangle$, **fold** gives the sum of the list's elements.
2. For $\mathcal{M} = \langle \mathbb{N}, \cdot, 1 \rangle$, **fold** gives the product of the list's elements.
3. For $\mathcal{M} = \langle \mathbb{N}, \max, 0 \rangle$, **fold** gives the maximum of the list's elements, with 0 for the empty list.
4. For $\mathcal{M} = \langle \mathbb{B}, \wedge, \top \rangle$, **fold** returns true if every element of the list is true.
5. For $\mathcal{M} = \langle \mathbb{B}, \vee, \perp \rangle$, **fold** returns true if some element of the list is true.
6. For $\mathcal{M} = \langle \text{List}(\mathcal{A}), ::, \langle \rangle \rangle$, **fold** concatenates all the component lists into one; e.g., $\text{fold}(\langle \langle 1, 2, 3 \rangle, \langle 3, 4, 5, 6 \rangle, \langle 10, 20 \rangle, \langle \rangle \rangle) = \langle 1, 2, 3, 3, 4, 5, 6, 10, 20 \rangle$.

You should check these claims.

As a final example, consider the set of predicates $\mathcal{A} \rightarrow \mathbb{B}$. As Example 17.11 below shows, we can make this into a monoid $\langle \mathcal{A} \rightarrow \mathbb{B}, \otimes, u \rangle$ in at least two ways: (i) The “every” monoid with $(p \otimes q)(a) = p(a) \wedge q(a)$ and $u = \text{const}_{\top}$, corresponding to item #4 for the codomain; or (ii) The “some” monoid with $(p \otimes q)(a) = p(a) \vee q(a)$ and $u = \text{const}_{\perp}$, corresponding to item #5 for the codomain. In case (i), **fold** takes a list of predicates and returns a new predicate that returns true when *every* predicate in the list returns true. In case (ii), **fold** takes a list of predicates and returns a new predicate that returns true when *at least one* predicate in the list returns true.

We also see that computation of **fold** is parallelizable in the sense discussed in the previous example.

As with other objects we've seen, we can create new monoids from other monoids in various ways.

Example 17.10 Products of Monoids

If we have two sets of things that we can combine monoidally, then we can combine tuples of things from both sets monoidally as well. We just apply the corresponding operators to each component.

Suppose that we have two monoids $\langle \mathcal{M}_1, \diamond, e_1 \rangle$ and $\langle \mathcal{M}_2, \boxdot, e_2 \rangle$. Form the set of pairs $\mathcal{M}_1 \times \mathcal{M}_2$ and combine pairs with

$$\langle m_1, m_2 \rangle \bowtie \langle m'_1, m'_2 \rangle = \langle m_1 \diamond m'_1, m_2 \boxdot m'_2 \rangle$$

with

$$\begin{aligned} \langle m_1, m_2 \rangle \bowtie \langle e_1, e_2 \rangle &= \langle m_1, m_2 \rangle \\ \langle e_1, e_2 \rangle \bowtie \langle m_1, m_2 \rangle &= \langle m_1, m_2 \rangle, \end{aligned}$$

so $\langle \mathcal{M}_1 \times \mathcal{M}_2, \bowtie, \langle e_1, e_2 \rangle \rangle$ is a monoid. The same idea works for tuples of any dimension. Hence, the *Cartesian product of monoids is a monoid*.

Example 17.11 Functions Returning Monoidal Values

Suppose $\langle \mathcal{M}, \diamond, e \rangle$ is a monoid and consider the set of functions $\mathcal{A} \rightarrow \mathcal{M}$ for a set $\mathcal{A}$. Let $u = \text{const}_e$ and define operator $\otimes$ by

$$(f \otimes g)(a) = f(a) \diamond g(a).$$

We have $f \otimes u = f = u \otimes f$ because

$$\begin{aligned} (f \otimes u)(a) &= f(a) \diamond u(a) = f(a) \diamond e = f(a) \\ (u \otimes f)(a) &= u(a) \diamond f(a) = e \diamond f(a) = f(a). \end{aligned}$$

We get associativity of $\otimes$ from the associativity of $\diamond$:

$$\begin{aligned} (f \otimes (g \otimes h))(a) &= (f(a) \diamond (g(a) \diamond h(a))) \\ &= ((f(a) \diamond g(a)) \diamond h(a)) = ((f \otimes g) \otimes h)(a). \end{aligned}$$

So, $\langle \mathcal{A} \rightarrow \mathcal{M}, \otimes, u \rangle$ is a monoid.

We say that $\otimes$ has *lifted* the operation $\diamond$ from $\mathcal{M}$ to $\mathcal{A} \rightarrow \mathcal{M}$.

Example 17.12 Dual Monoid If $\langle \mathcal{M}, \diamond, e \rangle$ is a monoid, we can define a new

monoid $\langle \mathcal{M}, \square, e \rangle$ from the original where

$$m_1 \square m_2 = m_2 \diamond m_1.$$

This satisfies all the monoid axioms because the original does. This is called the *dual* to the original monoid. A commutative monoid is its own dual.

Monoids appear in many places and different contexts. The next example illustrates this by revealing several monoids in the familiar context of building diagrams. We then look at two examples of how monoids arise as tools for computations that produce non-monoidal values.

Example 17.13 Diagrams

A simple representation of two-dimensional diagram is a list of “primitive” shapes, like circles, polygons, lines, points, and curves. Several different types of monoids arise in this context, including:

1. We might want to “stack” diagrams $d_1 \diamond d_2$, the contents of diagram d_1 on top of diagram d_2 . This just involves concatenating the primitives in the two lists, *but in reverse order*. This is the dual monoid (see previous example) to the ordinary list.
2. We can translate, scale, and rotate diagrams by operating on the coordinates of the primitives in the list. These are linear transformations for which the operation of composition makes them a monoid.
3. Style attributes specifying color, shading, etc. are associated with a diagram as a dictionary mapping attributes to their values. As in Example 17.2, these form a monoid.
4. The operation of putting another diagram *beside* another requires that we can find how far a diagram extends in any direction. The *envelope* of a diagram d is a function $\mathbb{R}^2 \rightarrow \mathbb{R}$ that takes a vector v and returns a scaling c of that vector so that every point in d projected onto v equals av for $a \leq c$.

We can combine the envelopes of two diagrams by taking the maximum of these functions. This gives an associative operation on envelopes, and we can add a special “empty” envelope as the identity, giving us a monoid.

This example is inspired by [Yorgey2012]

Example 17.14 Summary Statistics

While numeric addition gives us a monoid, *averaging* does not. To see why, notice that averaging fails both our requirements for a monoid. It is not associative

$$\frac{1}{2} \left(\frac{x_1 + x_2}{2} \right) + \frac{1}{2} x_3 = \frac{1}{4} x_1 + \frac{1}{4} x_2 + \frac{1}{2} x_3 \neq \frac{1}{2} x_1 + \frac{1}{4} x_2 + \frac{1}{4} x_3 = \frac{1}{2} x_1 + \frac{1}{2} \left(\frac{x_2 + x_3}{2} \right),$$

and it has no identity element

$$\frac{x_1 + x_2}{2} = x_1 \implies x_1 = x_2.$$

Moreover, if we are averaging say integers, the result need not even be an integer.

Averaging may not be a monoidal operation, but it does derive from one. Following Example 17.10, we can form the product monoid $\mathcal{A}$ from $\langle \mathbb{R}, +, 0 \rangle$ and $\langle \mathbb{N}, +0 \rangle$. An element of this monoid is a pair $\langle x, n \rangle$ and the operation, which we will denote as $+$, is given by $\langle x, n \rangle + \langle y, m \rangle = \langle x + y, n + m \rangle$.

Now suppose we have a list of numbers $\langle x_1, x_2, \dots, x_n \rangle$. We can express an average of these numbers as a function of monoid; the computation proceeds in three steps

1. Insert every number in the list into $\mathcal{A}$ with $\langle x \rangle \mapsto \langle x, 1 \rangle$. This gives us a list $\langle \langle x_1, 1 \rangle, \langle x_2, 1 \rangle, \dots, \langle x_n, 1 \rangle \rangle$
2. We apply the $+$ operator of $\mathcal{A}$ to these values giving

$$\langle x_1, 1 \rangle + \langle x_2, 1 \rangle + \dots + \langle x_n, 1 \rangle = \langle x_1 + \dots + x_n, n \rangle.$$

3. Apply the function $\langle y, m \rangle \mapsto y/m$ to the value from #2, yielding $(x_1 + \dots + x_n)/n$, the average.

Notice that #2 is just the fold function for $\mathcal{A}$ from the previous Example 17.9.

Averages and many other summary statistics can be derived in this way as the output of a function from some monoid. We first wrap our inputs in a monoidal value, apply the monoid operation, and then transform the result into our summary statistic.

Puzzle 134. Following up on the previous example, a common summary statistic

is the *sample variance*. For a list of numbers $\langle x_1, x_2, \dots, x_n \rangle$, its sample variance is

$$s_n^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x}_n)^2,$$

where $\bar{x}_n = (x_1 + \dots + x_n)/n$.

Show how to derive the sample variance as a function of a monoid. Can we you specify a single monoid from which to compute the average *and* sample variance?

As described in Example 17.8, monoidal computations can be *parallelized*. The previous examples illustrate this. Although the summary statistics are not themselves monoidal values in general (though some are), we can derive them from monoidal computations on our data. If we have a list of numbers, we can break up the list in any way we like, wrap the numbers in appropriate monoidal values, apply the operator to every sub-list *in parallel*, combine the monoidal results of all sub-lists into one – when they have all been computed, and then apply our function to compute the summary statistic or fold. This parallelizability of the monoid operator stems from its associativity. Try out this procedure for a simple list, like $\langle 1, 2, \dots, 10 \rangle$.

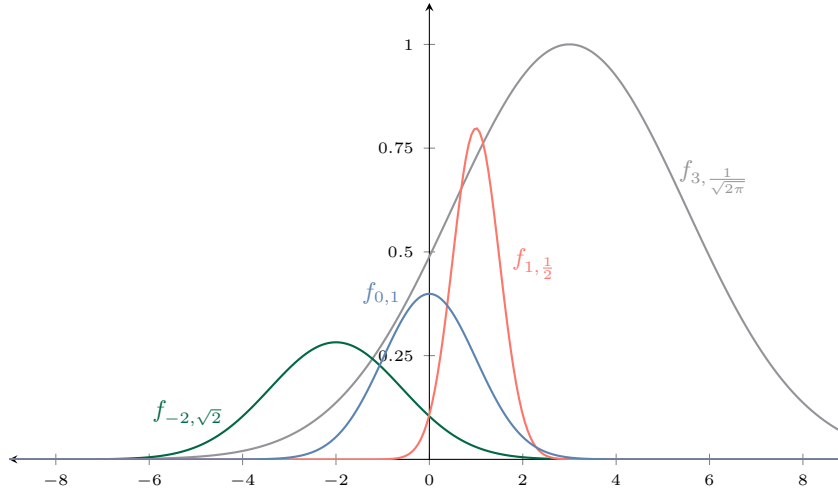


FIGURE 17.1. Four functions from the family generated by a monoid action in Example 17.15.

Example 17.15 Actions

Let $\mathcal{M} = \mathbb{R} \times (0_-)$ be the monoid derived from the monoids $\langle \mathbb{R}, +, 0 \rangle$ and $\langle (0_-), \cdot, 1 \rangle$ with operation $\odot$ given by

$$\langle a, r \rangle \odot \langle c, s \rangle = \langle a + rc, rs \rangle$$

and identity element $\langle 0, 1 \rangle$. This is associative but not commutative

$$\begin{aligned}\langle b, t \rangle \odot (\langle a, r \rangle \odot \langle c, s \rangle) &= \langle b + t(a + rc), rst \rangle \\ &= \langle b + at + crt, rst \rangle \\ &= \langle (b + ta) + rtc, rst \rangle = (\langle b, t \rangle \odot \langle a, r \rangle) \odot \langle c, s \rangle.\end{aligned}$$

Let $\mathcal{F} = \mathbb{R} \rightarrow \mathbb{R}$. The monoid operation $\odot$ on $\mathcal{M}$ induces a transformation $\odot: \mathcal{F} \rightarrow \mathcal{F}$ for which we use the same symbol as the monoid operator:

$$\langle c, s \rangle \odot f = \langle x \rangle \mapsto \frac{1}{s} f\left(\frac{x - c}{s}\right)$$

called an *action* of $\mathcal{M}$ on $\mathcal{F}$. The element $\langle c, s \rangle$ of $\mathcal{M}$ is *acting* on the function f to produce a new function of the same type. Note that we *overload* the monoid operator here, for reasons illustrated by equation (*) below. For simplicity, write $f_{c,s} := \langle c, s \rangle \odot f$.

This action behaves as we might hope. The action of the *identity* element on $\mathcal{M}$ is the *identity* function on $\mathcal{F}$, i.e., $f_{0,1} = f$, and

$$(\langle a, r \rangle \odot \langle c, s \rangle) \odot f = \langle b, t \rangle \odot (\langle a, r \rangle \odot f), \quad (*)$$

so we get the same function if we act then operate as if we operate then act.

The image of this action on a particular function f is a family of functions $f_{c,s}: \mathbb{R} \rightarrow \mathbb{R}$ for $\langle c, s \rangle \in \mathcal{M}$ consisting of all the re-centerings and scalings of f . For example, for $f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$, we get the family

$$f_{c,s}(x) = \frac{1}{s\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-c}{s}\right)^2},$$

several examples of which are shown in Figure 17.1.

The monoid $\mathcal{M}$ has a special property: every element has an inverse:

$$\langle -c/s, 1/s \rangle \odot \langle c, s \rangle = \langle 0, 1 \rangle = \langle c, s \rangle \odot \langle -c/s, 1/s \rangle,$$

so in particular for any $\langle c, s \rangle \in \mathcal{M}$, $f_{0,1} = \langle -c/s, 1/s \rangle \odot f_{c,s}$ and we can get any member of the family from any other with one action.

In general, if $\langle \mathcal{M}, \odot, e \rangle$ is a monoid, a (left) **action** of $\mathcal{M}$ on a set $\mathcal{X}$ is a function $a: \mathcal{M} \times \mathcal{X} \rightarrow \mathcal{X}$ that is compatible with the structure on $\mathcal{M}$, meaning:¹¹

1. $a(e, x) = x$ for any $x \in \mathcal{X}$, and

See Example 10.15 for more on families.

¹¹Similarly a right monoid action maps $\mathcal{X} \times \mathcal{M} \rightarrow \mathcal{X}$ with the monoid element as the right argument.

2. $a(m_2, a(m_1, x)) = a(m_2 \diamond m_1, x)$ for all $m_1, m_2 \in \mathcal{M}$ and $x \in \mathcal{X}$.

Put another way, for each m , define a function $a_m: \mathcal{X} \rightarrow \mathcal{X}$ by $a_m(x) = a(m, x)$. The above equations show that $a_e = \text{id}$ and $a_{m_2 \diamond m_1} = a_{m_2} \circ a_{m_1}$. It is often convenient to depict a monoid action by *overloading* the monoid operator instead of using an explicit named function $a()$, writing $m \diamond x$ to mean $a(m, x)$. The above properties of the action are then written as $e \diamond x = x$ and $m_2 \diamond (m_1 \diamond x) = (m_2 \diamond m_1) \diamond x$.

Earlier in this section, we talked about the idea of structure and of functions that preserve that structure. Monoids are a good example. The axioms of a monoid – the associative operator and identity element – comprise a monoid’s *structure* on the underlying set. (We can equivalently express this structure in terms of relations: a monoid $\langle \mathcal{M}, \diamond, e \rangle$ is determined by a unary relation $\{e\}$ on $\mathcal{M}$ and a ternary relation $\mathcal{O}$ on $\mathcal{M}$, where $\langle m_1, m_2, m_3 \rangle \in \mathcal{O}$ when $m_1 \diamond m_2 = m_3$.) Functions that preserve this structure will be functions from one monoid to another that respects the structure of both. As an example, let’s return to our favorite monoid – lists.

Example 17.16 List Lengths

Consider the monoid $\langle \text{List}(\mathcal{A}), ::, \langle \rangle \rangle$ from Example 17.1, and define a function $\text{length}: \text{List}(\mathcal{A}) \rightarrow \mathbb{N}$ that returns the length of the input list. Notice:

$$\text{length}(\langle \rangle) = 0 \quad (17.18)$$

$$\text{length}(\ell_1 :: \ell_2) = \text{length}(\ell_1) + \text{length}(\ell_2). \quad (17.19)$$

As noted in the previous puzzle, $\langle \mathbb{N}, +, 0 \rangle$ is itself a monoid. The length function maps the identity element of $\text{List}(\mathcal{A})$ to the identity element of $\mathbb{N}$, and it preserves the monoid operation in $\text{List}(\mathcal{A})$ via the monoid operation in $\mathbb{N}$.

Thus, length function preserves the monoid structure of $\text{List}(\mathcal{A})$. We can view it as a summary of the list that is consistent with how lists are constructed.

Consider again the set $\text{List}(\{1\})$. It contains exactly one list of length n for every $n \in [0..)$, so we can map each $n \in [0..)$ to exactly one list in a way that preserves the monoid structure. So $\text{List}(\{1\})$ is just a version of the $\langle \mathbb{N}, +, 0 \rangle$ monoid, in essence a copy of the natural numbers. $\text{List}(\{1\})$ is called the unary number representation of $\mathbb{N}$.

Another example is what enabled NASA engineers on the Apollo program to get so much out of their slide rules. Consider the monoids $\langle \mathbb{R}, +, 0 \rangle$ (the real numbers with addition) and $\langle (0_+), \cdot, 1 \rangle$ (the positive real numbers with multiplication). The function $\exp: \mathbb{R} \rightarrow (0_+)$ where $\exp(x) = e^x$ and its inverse function $\ln: (0_+) \rightarrow \mathbb{R}$

each respect the structure of these monoids. They preserve the respective identity elements

$$\begin{aligned}\exp(0) &= 1 \\ \ln(1) &= 0\end{aligned}$$

and the operators

$$\begin{aligned}\exp(a + b) &= \exp(a) \cdot \exp(b) \\ \ln(a \cdot b) &= \ln(a) + \ln(b).\end{aligned}$$

We can do our multiplication of positive numbers directly or take logs and convert multiplication to addition on real numbers. One practical use case for this transformation is when we are multiplying many positive numbers that might be small. Without some special efforts, the numerical representation of the product on a computer will *underflow*, giving an incorrect result. If we convert to a log scale, however, we can get correct results with a bit of care. Indeed, this example is even richer still, as we will see below.

Generalizing the previous examples, we define a homomorphism – structure-preserving function – between two monoids.

Suppose we have monoids $\langle \mathcal{M}, \diamond, e \rangle$ and $\langle \mathcal{N}, \square, u \rangle$. A function $f: \mathcal{M} \rightarrow \mathcal{N}$ is called a **monoid homomorphism** if preserves the identity elements and the operators:

$$\begin{aligned}f(e) &= u \\ f(m \diamond m') &= f(m) \square f(m'),\end{aligned}$$

If f is a bijection, we call it an **monoid isomorphism**.

A monoid homomorphism gives a picture of one monoid using the elements and operation of another. The picture may be a blurry summary or a sharp copy. The `length` function from the previous example is a monoid homomorphism between $\text{List}(\mathcal{A})$ and $\mathbb{N}$. When $\mathcal{A} = \{0, 1\}$, the picture is a blurry summary because many binary strings have the same length; when $\mathcal{A} = \{1\}$, it is a sharp copy as we saw earlier.

In Example 17.9, equations (17.17) and (17.14) show that the `fold` function is a monoid homomorphism from $\text{List}(\mathcal{M})$ to $\mathcal{M}$. In Example 17.15, we saw that $a_e = \text{id}$ and $a_{m_2 \diamond m_1} = a_{m_2} \circ a_{m_1}$, so the mapping $\langle m \rangle \mapsto a_m$ is thus a monoid homomorphism

from $\mathcal{M}$ to $\mathcal{X} \rightarrow \mathcal{X}$ (cf. Puzzle 133).

Puzzle 135. Suppose $\mathcal{A}$ is a finite set with $m = \#\mathcal{A}$ arranged in a fixed order $\langle a_1, a_2, \dots, a_m \rangle$. For $a \in \mathcal{A}$ and $w \in \text{List}(\mathcal{A})$, let $\text{count}_a(w) = \#\{i \mid w_i = a\}$ count the number of occurrences of a in the list w . Show that $\text{freq}: \text{List}(\mathcal{A}) \rightarrow \mathbb{N}^m$, defined by

$$\text{freq}(w) = \langle \text{count}_{a_1}(w), \text{count}_{a_2}(w), \dots, \text{count}_{a_m}(w) \rangle,$$

is a monoid homomorphism. What are the identity element and operator for the codomain monoid?

The last puzzle is another case where monoids enable parallel computation. We can compute the frequencies in parallel over both sub-lists of words and subsets of $\mathcal{A}$.

Example 17.17 Partial Permutations

Let $\mathcal{S}_n$ denote the set of permutations of n items, i.e., bijections $[1..n] \rightarrow [1..n]$.

The inclusion $\text{incl}: \mathcal{S}_n \rightarrow \mathcal{S}_{n+1}$ returns a permutation that leaves $n+1$ untouched. This is a homomorphism because following one permutation by another in $\mathcal{S}_n$ leads to the same operation in $\mathcal{S}_{n+1}$. It reveals the structure of the smaller permutation within the set of larger permutations.

Example 17.18 Generators

Let $\mathcal{G} = \{a, b, c\}$ be a finite set whose elements we treat as symbols. Let $\mathcal{M}$ be the set of strings of the form $a \cdots ab \cdots bc \cdots c$ for some number of repetitions of each, including the empty string. Write such a string as $a^j b^k c^\ell$ where the (natural number) exponents indicate the number of times a , b , or c is repeated. We define an operation $\diamond$ on $\mathcal{M}$ with

$$a^j b^k c^\ell \diamond a^r b^s c^t = a^{j+r} b^{k+s} c^{\ell+t},$$

which makes the empty string the identity element. $\mathcal{M}$ is a commutative monoid that is “generated” by the symbols in $\mathcal{G}$.

Let $g: \text{List}(\mathcal{G}) \rightarrow \mathcal{M}$ be defined so that $g(\langle \rangle)$ is the empty string and on a list $\langle g_1, \dots, g_n \rangle$ with j a ’s, k b ’s, and ℓ ’s, g returns the string $a^j b^k c^\ell$. Then g is a monoid homomorphism. Indeed, we could have defined the monoid $\mathcal{M}$ from the

beginning as the monoid on $\text{range}(g)$ that makes g a homomorphism.

Let's use this approach to add some constraints. Suppose we want $\mathbf{a} \diamond \mathbf{b}$ to be the identity (empty string). Define a function g_1 on $\text{List}(\mathcal{G})$ that maps $\langle \rangle$ to the empty string and a list with j , k , and ℓ $\mathbf{a}$'s, $\mathbf{b}$'s, and $\mathbf{c}$'s to the string $\mathbf{a}^{j-\min(j,k)}\mathbf{b}^{k-\min(j,k)}\mathbf{c}^\ell$. If we require that g_1 be a monoid homomorphism, this determines an operation that makes $\text{range}(g_1)$ a commutative monoid for which our constraint holds (as well as all the equations implied by that constraint).

Try to define a function g_2 and corresponding monoid that specifies a different constraint (or more than one), such as $\mathbf{b} \diamond \mathbf{b} = e$ or $\mathbf{a} \diamond \mathbf{b} \diamond \mathbf{c} = e$.

Example 17.19 Kinds. Recall that FRPs form a monoid with independent join $\star$ as the operation and the empty FRP `empty` as the identity element. Kinds also form a monoid with respect to independent join, with the empty kind $\langle \rangle$ as the identity element. The function `kind` maps an FRP to its kind, and it preserves the monoid structure:

$$\text{kind}(\text{empty}) = \langle \rangle \quad (17.20)$$

$$\text{kind}(X \star Y) = \text{kind}(X) \star \text{kind}(Y). \quad (17.21)$$

(In other words, `kind` is a monoid homomorphism.)

In fact, the `kind` function preserves other kinds of structure as well. For instance, if X is an FRP and M is a conditional FRP, we have

$$\text{kind}(X \triangleright M) = \text{kind}(X) \triangleright (\text{kind} \circ M), \quad (17.22)$$

where $\text{kind} \circ M$ is the conditional kind consistent with M . Similarly, if φ is a compatible statistic,

$$\text{kind}(\varphi(X)) = \varphi(\text{kind}(X)). \quad (17.23)$$

These relationships are a powerful for reasoning about FRPs and their kinds.

Example 17.20. In Puzzles 95 and 96, Table 10.5, and Example 10.21, we defined a function `lift` (operator $\hat{}$). For real numbers x, y , $\hat{x}$ is the sequence

Based on Part I

Part of the "Sequences and Streams" series.

$x, 0, 0, \dots$, and

$$\begin{aligned}\widehat{x+y} &= \widehat{x} + \widehat{y} \\ \widehat{xy} &= \widehat{x} \widehat{y} \\ \widehat{x^{-1}} &= (\widehat{x})^{-1},\end{aligned}$$

where $x \neq y$ implies that $\widehat{x} \neq \widehat{y}$.

Given a sequence $a \in \text{Seq}(\mathbb{R})$, we have seen that $a = \sum_{n=0}^{\infty} a_n z^n$, where z is defined in equation (10.19) from Example 10.21. Define the function $h: \text{Seq}(\mathbb{R}) \rightarrow \text{Seq}(\mathbb{R})$ by

$$h(a) = \sum_{n=0}^{\infty} a_n \frac{z^n}{n!}. \quad (17.24)$$

We will use this “exponential power series” representation in various counting problems. The function h satisfies

$$h(\widehat{1}) = \widehat{1} \quad (17.25)$$

$$h(a \otimes b) = h(a)h(b), \quad (17.26)$$

where on the left and right we have the Shuffle Product and Product from Table 10.5, respectively. This h is invertible and is thus an isomorphism. This tells us that we can use the Shuffle Product on the ordinary representation to represent the Product on the exponential representation.

SEMIRINGS. We saw that the set of natural numbers $\mathbb{N}$ can be considered a monoid in *two distinct ways*: with addition and 0 or multiplication and 1 as operator and identity element. The two monoidal structures are actually *entangled* in our familiar arithmetic through laws that govern how addition and multiplication interact:

$$0 \cdot n = 0 \quad (\text{Annihilation Laws}) \quad (17.27)$$

$$n \cdot 0 = 0 \quad (17.28)$$

$$k \cdot (m + n) = k \cdot m + k \cdot n \quad (\text{Distributive Laws}) \quad (17.29)$$

$$(m + n) \cdot k = m \cdot k + n \cdot k. \quad (17.30)$$

Taken together, addition and multiplication give $\mathbb{N}$ even more structure than that of a monoid, which suggests that we consider them together from the outset.

This is the motivating example of a common phenomenon. A set that can be

described as a monoid in two distinct ways where those two monoid operations interact like addition and multiplication is called a **semiring**.

The sets of integers and real numbers both form a semiring with the same operators and identities as the natural numbers, though these sets have extra structure because every value has an additive inverse ($n + (-n) = 0$) and non-zero real numbers have a multiplicative inverse ($a \cdot 1/a = 1$). Boolean logic gives another semiring. We equip the Booleans $\mathbb{B}$ with operators logical-or ($\vee$) as “addition” and logical-and ($\wedge$) as “multiplication”. False ($\perp$) and true ($\top$) play the roles of 0 and 1 because

$$\perp \vee b = b, \quad (17.31)$$

$$\top \wedge b = b, \quad (17.32)$$

and

$$\perp \wedge b = \perp = b \wedge \perp. \quad (17.33)$$

The distributive properties follow by confirming the equation

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$

for all eight values (each $\perp$ or $\top$) of a, b, c .

Our definition of a semiring generalizes the natural numbers in one useful respect: we do not require that the “multiplication” operation be commutative. Using $\boxplus$ and $\boxtimes$ as the generic “addition” and “multiplication” operators in a semiring, we require that $\boxplus$ be commutative (i.e., $a \boxplus b = b \boxplus a$) but not necessarily multiplication (i.e., we allow $a \boxtimes b$ to be unequal to $b \boxtimes a$). A semiring for which $\boxtimes$ is commutative is called a *commutative semiring*. So the natural numbers $\mathbb{N}$, the integers $\mathbb{Z}$, the real numbers $\mathbb{R}$, and Boolean logic $\mathbb{B}$ with the operations and identities described above are all commutative semirings. But there are many useful examples that are not commutative as we will see.

We say that $\langle \mathcal{S}, \boxplus, \boxtimes, \mathbf{0}, \mathbf{1} \rangle$ is a **semiring** when $\mathcal{S}$ is a set with two special elements, denoted by $\mathbf{0}$ and $\mathbf{1}$, and two operators $\boxplus, \boxtimes: \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ that satisfy

1. $\langle \mathcal{S}, \boxplus, \mathbf{0} \rangle$ is a *commutative monoid*
2. $\langle \mathcal{S}, \boxtimes, \mathbf{1} \rangle$ is a monoid
3. $\mathbf{0}$ annihilates under $\boxtimes$: $x \boxtimes \mathbf{0} = \mathbf{0} = \mathbf{0} \boxtimes x$

4. $\boxdot$ distributes over $\boxplus$:

$$a \boxdot (b \boxplus c) = (a \boxdot b) \boxplus (a \boxdot c) \quad (17.34)$$

$$(b \boxplus c) \boxdot a = (b \boxdot a) \boxplus (c \boxdot a). \quad (17.35)$$

The operator $\boxdot$ need not be commutative; if it is, we have a *commutative semiring*.

When the operators and identity elements are understood from context, we often just loosely refer to $\mathcal{S}$ as the semiring. Note that $\mathbf{0}$ and $\mathbf{1}$ here are just convenient *names* for particular *elements* of the set $\mathcal{S}$, which we call the zero and unit elements respectively. They need not represent numbers in general. Again, we use $\boxplus$ and $\boxdot$ as generic operators here; for specific semirings, we will replace them by particular operators with those roles. We will assume that $\boxdot$ binds more tightly than $\boxplus$, so $a \boxdot b \boxplus c \boxdot d = (a \boxdot b) \boxplus (c \boxdot d)$.

The distributive properties (17.34) and (17.35) imply that every expression in a semiring can be written as a sum of products. In the commutative case, each implies the other. In the non-commutative case, both are required, though a weaker notion is sometimes useful, keeping only (17.34) for a *left semiring* or (17.35) for a *right semiring*.

Puzzle 136. Consider $\langle 2, \oplus, \cdot, 0, 1 \rangle$ where $\cdot$ is ordinary multiplication and where $\oplus$ is bitwise exclusive-or with $1 \oplus 0 = 1 = 0 \oplus 1$ and $1 \oplus 1 = 0 = 0 \oplus 0$. Confirm that this is a semiring. Is it a commutative semiring?

Semirings have application in a wide range of problems, including optimization, statistical and machine learning, probabilistic graphical models, signal processing, data compression, cryptography, geometry, network algorithms, satisfiability/constraint satisfaction, and database queries. Once you see the pattern, you will start noticing semirings pop up in unexpected places. We will start by identifying diverse examples of semirings and then we will explore some of their uses.

Example 17.21 The Semiring of Subsets

Working with sets and set operations often feels like a kind of arithmetic, and this is not a coincidence. Let $\mathcal{A}$ be a set and $2^{\mathcal{A}}$ its power set, the set of all $\mathcal{A}$'s

subsets. Then $\langle 2^{\mathcal{A}}, \cup, \cap, \{\}, \mathcal{A} \rangle$ is a commutative semiring. For $\mathcal{C} \subseteq \mathcal{A}$,

$$\begin{aligned}\mathcal{C} \cup \{\} &= \mathcal{C} \\ \mathcal{C} \cap \mathcal{A} &= \mathcal{C},\end{aligned}$$

showing that $\{\}$ is the zero element and $\mathcal{A}$ the unit element.

As $\mathcal{C}$ can have no elements in common with the empty set, which has no elements, $\mathcal{C} \cap \{\} = \{\}$, which is the annihilation property. The distributive property

$$\mathcal{C} \cap (\mathcal{B} \cup \mathcal{D}) = (\mathcal{C} \cap \mathcal{B}) \cup (\mathcal{C} \cap \mathcal{D})$$

reduces to a logical statement for any $a \in \mathcal{A}$:

$$\begin{aligned}a \in \mathcal{C} \cap (\mathcal{B} \cup \mathcal{D}) &\iff (a \in \mathcal{C}) \wedge (a \in \mathcal{B} \vee a \in \mathcal{D}) \\ a \in (\mathcal{C} \cap \mathcal{B}) \cup (\mathcal{C} \cap \mathcal{D}) &\iff (a \in \mathcal{C} \wedge a \in \mathcal{B}) \vee (a \in \mathcal{C} \wedge a \in \mathcal{D}).\end{aligned}$$

The right-hand sides follow from the distributive property of the Boolean semiring.

Example 17.22 The Log Semiring

Numerical calculations that involve products of many, potentially small positive numbers $\prod_i p_i$ can fail because of *underflow*, when a number becomes too small to be represented accurately. The exponent of floating-point numbers is represented with far fewer bits than the fractional part, and a product can drive the exponent out of its range quite easily. An alternative is to use the *log semiring* on $[-\infty, \infty]$, with infinities included, that computes on a logarithmic scale:

$$x \boxplus y = \ln(e^x + e^y) \tag{17.36}$$

$$x \boxtimes y = x + y \tag{17.37}$$

Here, the zero element is $-\infty$ and the unit element 0. Products become sums and are numerically relatively inexpensive.

The $\boxplus$ operation can be rewritten in a particularly stable way

$$x \boxplus y = \max(x, y) + \ln(e^{x-\max(x,y)} + e^{y-\max(x,y)}) \quad (17.38)$$

$$= \max(x, y) + \ln(1 + e^{-(\max(x,y)-\min(x,y))}) \quad (17.39)$$

$$\bigoplus_{i=1}^n x_i = \max_i x_i + \ln\left(\sum_i e^{x_i - \max_j x_j}\right), \quad (17.40)$$

which ensures that the exponents are always negative. (Underflow in that term does not destabilize the calculation because of the first term.)

Numerical calculations in this semiring give more stable and accurate results when underflow is a concern. Note that the $\boxplus$ operation is numerically less efficient because of the logarithm and exponentials, but often calculations can be arranged with products ($\boxtimes$) done in batches, making the overall process efficient. This semiring is frequently useful in fitting high-dimensional statistical models, among other use cases.

Puzzle 137. Confirm that the log semiring is a semiring.

Example 17.23 Square Matrices

Although we will discuss matrix multiplication in detail in the next Section, here we look ahead a bit. Let $\mathcal{M}_n$ denote the set of real $n \times n$ matrices. Then, setting $\boxplus$ to be entrywise addition and $\boxtimes$ as matrix multiplication, gives us a semiring. The additive identity $\mathbf{0}$ is the matrix of all zeroes, and $\mathbf{1}$ as the matrix with 1s on the diagonal and 0s elsewhere, called the “identity matrix.”

Unlike previous examples, this semiring is *not* commutative because matrix multiplication is not commutative. For instance, when $n = 2$,

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 7 & 10 \\ 3 & 4 \end{bmatrix} \neq \begin{bmatrix} 1 & 4 \\ 3 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}.$$

The next Section will explain matrix multiplication in detail.

Depends lightly on linear algebra or Section 17.2

Example 17.24 The Semiring of Relations

Uses ideas from Chapter 16

For a set $\mathcal{S}$, every subset $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a (homogeneous) binary relation on $\mathcal{S}$. Let $\Delta = \{\langle s, s \rangle \mid s \in \mathcal{S}\}$ be the “diagonal” relation. Then $\langle \mathcal{P}^{\mathcal{S} \times \mathcal{S}}, \cup, \circ, \{\}, \Delta \rangle$ is a semiring, where $\circ$ is the composition of relations defined in equation (16.1). This semiring is not commutative.

Union with the empty set does not change a relation, so $\{\}$ is the zero element. For any relation $\mathcal{R}$, $\langle s_1, s_2 \rangle$ belongs to the composition $\mathcal{R} \circ \Delta$ if there is an s' with $\langle s_1, s' \rangle \in \mathcal{R}$ and $\langle s', s_2 \rangle \in \Delta$. But this only happens for $s' = s_2$, so $\mathcal{R} \circ \Delta = \mathcal{R}$. A similar argument with the terms in the opposite order shows that Δ is the unit element, an identity element for composition.

The composition with the empty relation is itself empty, giving the annihilation property. The distributive property requires that

$$\mathcal{R}_1 \circ (\mathcal{R}_2 \cup \mathcal{R}_3) = (\mathcal{R}_1 \circ \mathcal{R}_2) \cup (\mathcal{R}_1 \circ \mathcal{R}_3).$$

We can see that this holds term by term. If $\langle s_1, s_2 \rangle$ belongs to the left hand side, then there is an s' with $\langle s_1, s' \rangle \in \mathcal{R}_1$ and $\langle s', s_2 \rangle$ in either $\mathcal{R}_2$ or $\mathcal{R}_3$. Then this must belong to the right-hand relation. The argument works in reverse as well.

Example 17.25 Regular Languages

Computer scientists define a language as a *set of strings* with characters drawn from some finite “alphabet” set $\mathcal{A}$. So, a language is a subset of $\text{List}(\mathcal{A})$.

A *regular language* can be described by what is called a *regular expression* – a way to specify a pattern by composing simpler rules, or equivalently recognized by a kind of computational machine called a finite automaton.

The regular languages on alphabet $\mathcal{A}$ form a semiring $\mathcal{RL}(\mathcal{A})$ on the set $\mathcal{P}^{\text{List}(\mathcal{A})}$ with

- $\boxplus = \cup$, with $\mathcal{L}_1 \boxplus \mathcal{L}_2$ the set of strings in $\mathcal{L}_1$ or $\mathcal{L}_2$. The empty language $\{\}$ is the corresponding identity element, as it does not add any strings;
- $L_1 \boxdot L_2$ is the set containing each string in L_1 concatenated with each string in L_2 ; that is, $L_1 \boxdot L_2 = \{s_1 s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}$. The set containing only the empty string is the identity element (which is different from the empty set) since the empty string is the identity for concatenation.

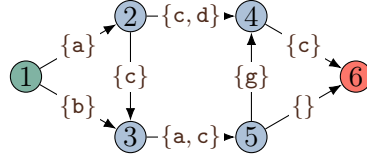
This is *not* a commutative semiring; concatenation depends on order.

Each edge in the directed graph below is labeled by a regular language in the semiring. Along any path, we combine the languages with $\boxdot$, and across paths

See 17.1 on $\text{List}(\mathcal{A})$.

The regular expression `ac*zz*` matches any string that starts with a, is followed by zero or more c's and then by one or more z's.

we aggregate the languages with $\boxplus$.



For example, path $1 \rightarrow 3 \rightarrow 5 \rightarrow 6$ gives the language $\{b\} \boxtimes \{a, c\} \boxtimes \{\} = \{ba, bc\}$, and path $1 \rightarrow 2 \rightarrow 4 \rightarrow 6$ gives the language $\{a\} \boxtimes \{c, d\} \boxtimes \{c\} = \{acc, adc\}$. The graph as a whole represents the language that aggregates (with $\boxplus$) over the languages of all possible paths from node 1 to node 6:

$$\begin{aligned}
 & (\{a\} \boxtimes \{c, d\} \boxtimes \{c\}) \boxplus (\{a\} \boxtimes \{c\} \boxtimes \{a, c\} \boxtimes \{\}) \\
 & \boxplus (\{a\} \boxtimes \{c\} \boxtimes \{a, c\} \boxtimes \{g\} \boxtimes \{c\}) \boxplus (\{b\} \boxtimes \{a, c\} \boxtimes \{\}) \\
 & \boxplus (\{b\} \boxtimes \{a, c\} \boxtimes \{g\} \boxtimes \{c\}) \\
 & = \{acc, adc\} \cup \{\} \cup \{acagc, accgc\} \cup \{\} \cup \{bagc, bcgc\} \\
 & = \{acc, adc, acagc, accgc, bagc, bcgc\}.
 \end{aligned}$$

Example 17.26 Numeric Semirings

There is a variety of useful semirings on the real numbers extended by including $-\infty$ or ∞ :

- *Min-Sum*. The set $(-\infty, \infty]$, which includes ∞ , is a commutative semiring with $\boxplus = \min$, $\boxtimes = +$, $\mathbf{0} = \infty$, and $\mathbf{1} = 0$.
- *Max-Sum*. The set $[-\infty, \infty)$, which includes $-\infty$, is a commutative semiring with $\boxplus = \max$, $\boxtimes = +$, $\mathbf{0} = -\infty$, and $\mathbf{1} = 0$.
- *Max-Min*. The set $[-\infty, \infty]$, which includes $\pm\infty$, is a commutative semiring with $\boxplus = \max$, $\boxtimes = \min$, $\mathbf{0} = -\infty$, and $\mathbf{1} = \infty$.
- *Min-Product*. The set $(0, \infty]$, which includes ∞ , is a commutative semiring with $\boxplus = \min$, $\boxtimes = \cdot$, $\mathbf{0} = \infty$, and $\mathbf{1} = 1$.
- *Max-Product*. The set $[0, \infty)$ is a commutative semiring with $\boxplus = \max$ the maximum of two numbers, $\boxtimes = \cdot$ regular multiplication, $\mathbf{0} = 0$, and $\mathbf{1} = 1$.

The first two are sometimes called, respectively, the Tropical and Arctic semirings.

An expression like $4 \boxdot 1 \boxplus 7 \boxdot 9 \boxplus 3 \boxdot 100$ has values 5 (Min-Sum), 103 (Max-Sum), 7 (Max-Min), 4 (Min-Product), and 300 (Max-Product) in these semirings. The choice of semiring allows us to select for different features of the terms. We will get a sharper sense of how these semirings are useful in the graph-based illustrations below (see page 728).

Example 17.27 Polynomials

Suppose $\langle \mathcal{S}, \boxplus, \boxdot, \mathbf{0}, \mathbf{1} \rangle$ is a semiring. If $x \in \mathcal{S}$, write x^k for $x \boxdot \cdots \boxdot x$ with k terms and x^0 for $\mathbf{1}$. Then a function $p: \mathcal{S} \rightarrow \mathcal{S}$ of the form

$$p(x) = a_0 \boxdot x^0 \boxplus a_1 \boxdot x^1 \boxplus \cdots \boxplus a_d \boxdot x^d,$$

for $a_0, a_1, \dots, a_d \in \mathcal{S}$ and $d \in [0..)$, is called a *polynomial*. The values $a_0, a_1, \dots, a_d$ are called the *coefficients* of the polynomial and d is its *degree*. Notice that $p(\mathbf{0}) = a_0$ and $p(\mathbf{1}) = a_0 \boxplus a_1 \boxplus \cdots \boxplus a_d$.

In the ordinary semiring of real numbers, we get the polynomials familiar from high-school algebra. For comparison, let's consider polynomials in other semirings we have defined earlier:

- *Min-Sum*. For $k \in [1..)$, $x^k = x + x + \cdots + x = kx$, where the right-hand side is ordinary numeric multiplication. Hence:

$$a_0 \boxdot x^0 \boxplus a_1 \boxdot x^1 \boxplus \cdots \boxplus a_d \boxdot x^d = \min \{a_0, a_1 + x, a_2 + 2x, \dots, a_d + dx\}$$

- *Max-Sum*. This semiring is similar, with max replacing min:

$$a_0 \boxdot x^0 \boxplus a_1 \boxdot x^1 \boxplus \cdots \boxplus a_d \boxdot x^d = \max \{a_0, a_1 + x, a_2 + 2x, \dots, a_d + dx\}$$

- *Boolean Logic*. In this semiring, $s \boxplus s = s$ and $s \boxdot s = s$ for any s , so $x^k = x$. Then:

$$\begin{aligned} a_0 \boxdot x^0 \boxplus a_1 \boxdot x^1 \boxplus \cdots \boxplus a_d \boxdot x^d &= a_0 \vee (a_1 \wedge x) \vee (a_2 \wedge x) \vee \cdots \vee (a_d \wedge x) \\ &= a_0 \vee ((a_1 \vee \cdots \vee a_d) \wedge x). \end{aligned} \quad (17.41)$$

If this polynomial is p , then $p(\perp) = a_0$ and $p(\top) = a_0 \vee a_1 \vee \cdots \vee a_d$.

- *Regular Languages*. Consider $\mathcal{RL}(\mathcal{A})$ for a finite alphabet $\mathcal{A}$ and fix

languages $a_0, a_1, \dots, a_d$. For any x in $\mathcal{RL}(\mathcal{A})$,

$$\begin{aligned} a_0 \sqcup x^0 \boxplus a_1 \sqcup x^1 \boxplus \dots \boxplus a_d \sqcup x^d \\ = a_0 \cup (a_1 \sqcup x) \cup (a_2 \sqcup x \sqcup x) \dots \cup (a_d \sqcup \overbrace{x \sqcup \dots \sqcup x}^{d \text{ times}}) \end{aligned}$$

is the set of strings containing either: a string in a_0 or a string in a_1 followed by a string in x , or a string in a_2 followed by two successive strings in x , or a string in a_3 followed by three successive strings in x , or so on, up to a string in a_d followed by d successive strings in x . For example if all the a_i 's are the singleton set {"on"} and x is the singleton set {"and on"}, then this language contains the strings "on", "on and on", "on and on and on", through "on and on ... and on" which includes d "and"s.

In fact, for any semiring $\mathcal{S}$, the set of polynomials as defined above $\mathcal{P}(\mathcal{S})$ is *also a semiring*. With slight abuse of notation, we can use the same operator names for both semirings, giving polynomials $p \boxplus q$ and $p \sqcup q$ from polynomials p, q . Suppose that p and q have respective coefficients $a_0, \dots, a_k$ and $b_0, \dots, b_m$. We can take $d = \max(k, m)$ by setting $b_j = \mathbf{0}$ if $j > m$ or $a_j = \mathbf{0}$ if $j > k$. Then by the distributive property of semirings, we can define polynomials $p \boxplus q$ and $p \sqcup q$:

$$(p \boxplus q)(x) = p(x) \boxplus q(x) \quad (17.42)$$

$$= (a_0 \boxplus b_0) \sqcup x^0 \boxplus (a_1 \boxplus b_1) \sqcup x^1 \boxplus \dots \boxplus (a_d \boxplus b_d) \sqcup x^d \quad (17.43)$$

$$(p \sqcup q)(x) = p(x) \sqcup q(x) \quad (17.44)$$

$$= (a_0 \sqcup b_0) \sqcup x^0 \boxplus (a_1 \sqcup b_0 \boxplus a_0 \sqcup b_1) \sqcup x^1 \quad (17.45)$$

$$\boxplus \dots \boxplus (a_d \sqcup b_0 \boxplus a_{d-1} \sqcup b_1 \boxplus \dots \boxplus a_0 \sqcup b_d) \sqcup x^d \quad (17.46)$$

$$= \bigoplus_{i=0}^d (a_i \sqcup b_0 \boxplus a_{i-1} \sqcup b_1 \boxplus \dots \boxplus a_1 \sqcup b_{i-1} \boxplus a_0 \sqcup b_i) \sqcup x^i. \quad (17.47)$$

This looks just like the sum and product of ordinary polynomials, replacing ordinary addition and multiplication by the semiring operations. (The coefficient $a_i \sqcup b_0 \boxplus a_{i-1} \sqcup b_1 \boxplus \dots \boxplus a_1 \sqcup b_{i-1} \boxplus a_0 \sqcup b_i$ is called the *convolution* of the a 's and b 's.)

Example 17.28 Formal Power Series and Polynomials

Semiring $\mathcal{S}$ A formal power series in variables from a finite set $\mathcal{V}$ is a function $f: \text{List}(\mathcal{A}) \rightarrow \mathcal{S}$.

Polynomials with natural number coefficients over a variable x forms a semiring with $\boxplus$ equal to term-by-term addition of the coefficients and $\boxtimes$ equal to multiplication of the expressions

$$\begin{aligned} (a_0 + a_1x + a_2x^2 + \dots + a_dx^d) \boxtimes (b_0 + b_1x + b_2x^2 + \dots + b_dx^d) \\ = a_0b_0 + (a_1b_0 + b_1a_0)x + (a_2b_0 + a_1b_1 + a_0b_2)x^2 + \\ \dots + (a_db_0 + a_{d-1}b_1 + \dots a_0b_d)x^d. \end{aligned}$$

Here $\mathbf{0}$ is the 0-polynomial and $\mathbf{1} = 1$. (This extends easily to coefficients in any semiring. For integer or real coefficients, this comprises a ring.)

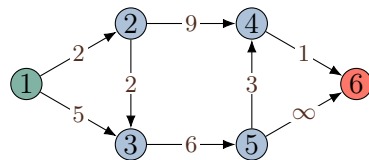
Example 17.29 Sequences

Similarly, the sequences defined in Example 10.20 and Puzzle 95 form a semiring in several ways with $\boxplus$ equal to the term by term addition and $\mathbf{0} = \text{repeat}(\mathbf{0}) = \mathbf{0}^*$. For $\boxtimes$, we can use any of the products defined in Example 10.21. For the regular product or shuffle product, $\mathbf{1} = \hat{\mathbf{1}} = \text{lift}(\mathbf{1})$ is the identity; for the parallel product setting $\mathbf{1}$ to $\text{repeat}(\mathbf{1})$ does the job.

Part of the “Sequences and Streams” series.

We will see applications for some of these (and others) as we proceed. For some immediate gratification, see Section 17.2. See also the following discussion for more on the extra structure in the integer and real number semirings.

First, we decorate the edges with values from the Min-Sum semiring. Think of these as distances or costs in moving from source to target.



Look at every path from node 1 to node 6. Along each path, we combine distances with $\boxtimes$ and *across* paths we aggregate with $\boxplus$. This gives us

$$\begin{aligned} (2 \boxtimes 9 \boxtimes 1) \boxplus (2 \boxtimes 2 \boxtimes 6 \boxtimes 3 \boxtimes 1) \boxplus (2 \boxtimes 2 \boxtimes 6 \boxtimes \infty) \\ \boxplus (5 \boxtimes 6 \boxtimes 3 \boxtimes 1) \boxplus (5 \boxtimes 6 \boxtimes \infty) \\ = \min(12, 14, \infty, 15, \infty) = 12. \end{aligned}$$

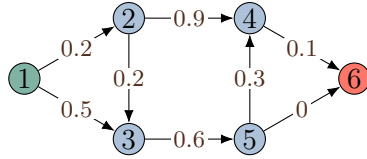
The Min-Sum semiring operations gives us the minimum distance/cost over all paths from node 1 to node 6.

Second, we use the Max-Min semiring and think of the edge weights as representing the capacity of flow on a channel along that edge. Using the same graph as before, we combine capacities along the paths from 1 to 6 using the semiring operations:

$$\begin{aligned}
 & (2 \boxminus 9 \boxminus 1) \boxplus (2 \boxminus 2 \boxminus 6 \boxminus 3 \boxminus 1) \boxplus (2 \boxminus 2 \boxminus 6 \boxminus \infty) \\
 & \boxplus (5 \boxminus 6 \boxminus 3 \boxminus 1) \boxplus (5 \boxminus 6 \boxminus \infty) \\
 & = \max(1, 1, 2, 1, 5) = 5.
 \end{aligned}$$

The Max-Min semiring operations gives us optimal capacity of flow from nodes 1 to 6.

Third, we use the Max-Product semiring and decorate the edges with the *reliability* of the connection represented by the edge. Here, reliability is measured by a number between 0 and 1 (i.e., a probability). We use a variant of the previous graph.



Aggregating again over paths from node 1 to node 6, we get

$$\begin{aligned}
 & (0.2 \boxtimes 0.9 \boxtimes 0.1) \boxplus (0.2 \boxtimes 0.2 \boxtimes 0.6 \boxtimes 0.3 \boxtimes 0.1) \boxplus (0.2 \boxtimes 0.2 \boxtimes 0.6 \boxtimes 0) \\
 & \boxplus (0.5 \boxtimes 0.6 \boxtimes 0.3 \boxtimes 0.1) \boxplus (0.5 \boxtimes 0.6 \boxtimes 0) \\
 & = \max(0.018, 0.00072, 0, 0.009, 0) = 0.018.
 \end{aligned}$$

We get the reliability of the most reliable path from node 1 to 6.

One common way semirings arise is when we realize that some operation can be described with the analogue of subtraction or division. Then we can replace an operation on numbers with an operation on semirings, and by changing the semiring, we often get interesting new techniques. This is the case with matrix multiplication as described in the next section. This also occurs, for example, with backpropagation in neural networks, dynamic programming, and automatic differentiation. Here's one more example.

Example 17.30 The Binomial Theorem

The basic version of the Binomial theorem states that for numbers x, y and $n \in [0..)$,

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k. \quad (17.48)$$

This holds because the number of terms $x^{n-k}y^k$ in the product is the number of ways of choosing y in k of the terms and x in the rest, i.e., the number of subsets of cardinality k of $[1..n]$. The derivation of this identity only uses the semiring structure of numbers, so the Binomial theorem extends to any semiring.

If $\langle \mathcal{S}, \boxplus, \boxminus, \mathbf{0}, \mathbf{1} \rangle$ is a semiring, $x, y \in \mathcal{S}$, and $xy = yx$, the same argument works:

$$(x \boxplus y)^n = \boxplus_{k=0}^n \binom{n}{k} (x^{n-k} \boxminus y^k), \quad (17.49)$$

where $js = s \boxplus \cdots \boxplus s$ with j terms and $s^0 = \mathbf{1}$, $s^1 = s$, and $s^m = s \boxminus \cdots \boxminus s$ with m terms.

We can apply this, for instance, to matrix multiplication, sequence products, polynomial products and much more.

Puzzle 138. We have seen several examples of structure (e.g., monoids, graphs) and of structure-preserving functions – *homomorphisms* – between sets equipped with that structure.

Building on these earlier examples, how might you define a *semiring homomorphism*? It will be a function from one semiring to another that respects both of the operations $\boxplus$ and $\boxminus$ and the identity elements for those operations.

ADDING INVERSES. Monoids and semirings arise frequently, and sometimes they have extra structure that we can exploit. Of particular interest is the existence of *inverse elements*. A value m has an inverse for a binary operation $\diamond$, if there is another value – typically denoted m^{-1} – for which $m \diamond m^{-1} = e = m^{-1} \diamond m$, where e is the identity element. When we add the requirement that inverse elements exist in monoids and semirings, the additional structure gives us additional descriptive power. We will not go into depth on these structures, but it is worth being aware of them. So here is a quick overview:

- A monoid $\langle \mathcal{M}, \diamond, e \rangle$ in which every element m has an *inverse element* m^{-1} such that $m \diamond m^{-1} = e = m^{-1} \diamond m$ is called a **group**.

For example: The integers $\langle \mathbb{Z}, +, 0 \rangle$ and real numbers $\langle \mathbb{R}, +, 0 \rangle$ are both groups with inverse $-x$ for value x . We get subtraction as well as addition.

- A semiring $\langle \mathcal{S}, \boxplus, \boxminus, \mathbf{0}, \mathbf{1} \rangle$ in which every $\langle \mathcal{S}, \boxplus, \mathbf{0} \rangle$ is a group (a monoid with inverses) is called a **ring**. If the semiring is commutative, then the ring is commutative.

Following the previous item, the integers $(\langle \mathbb{Z}, +, \cdot, 0, 1 \rangle)$ and real numbers $(\langle \mathbb{R}, +, \cdot, 0, 1 \rangle)$ are both rings, allowing subtraction as well as addition, both of which interact with multiplication.

- A commutative ring $\langle \mathcal{R}, \boxplus, \boxminus, \mathbf{0}, \mathbf{1} \rangle$ in which every element $r \neq \mathbf{0}$ has a *multiplicative* inverse $r^{-1} \equiv 1/r$ ($r \cdot (1/r) = 1 = (1/r) \cdot r$) is called a **field**.

The real numbers form a field, allowing division of non-zero values. Another example is the set of bits $\langle \mathbb{2}, \oplus, \cdot, 0, 1 \rangle$ with $\oplus$ addition modulo 2 (or equivalently bitwise exclusive-or) and $\cdot$ multiplication.

17.2 Graphs and Matrices: An Enlightening Connection

In this Section, we tie together several of the threads we’ve pulled so far in this Chapter – functions, operators, graphs, algebraic structures – into a satisfying bundle. We define an algebraic structure on a family of graphs that lets us solve a range of interesting problems. The graphs in this family are in one-to-one correspondence with the tables of numbers called *matrices* that are used in linear algebra, and the operations on the graphs correspond to operations on matrices, including “matrix multiplication.” Because these operations depend only on a notion of addition and multiplication, they can be defined in any semiring, which greatly expands their scope of application. Whether or not you have seen matrices before, this enlightening introduction will give you a conceptual interpretation of matrices and their operations and sense of how they can be used.

For reference, graphs are discussed earlier in this section (e.g., Examples 10.18 and 10.19) and in Interlude G. In particular, Example 10.18 explains the data needed to specify a directed graph in more detail; we use this below. More formal detail on matrices and vectors is covered in Section 17.3. “Matrix multiplication” arises in linear algebra as a mechanical procedure for composing linear transformations. The steps in the procedure have a conceptual basis,¹² but I recommend starting with the treatment here for a more intuitive foundation. In particular, matrix multiplication

¹²Pun intended.

emerges as a way to aggregate paths between nodes in a graph whose edges have associated values.

Assume throughout this Section that $m, n \in [0..)$ and that $\langle \mathcal{S}, \boxplus, \boxminus, \mathbf{0}, \mathbf{1} \rangle$ is a semiring as defined in the previous Section. The standard semiring of real numbers $\langle \mathbb{R}, +, \cdot, 0, 1 \rangle$ is the default case, which we indicate by writing loosely “ $\mathcal{S} = \mathbb{R}$ ”. Let $\mathfrak{r}$ (for “row”) and $\mathfrak{c}$ (for “column”) denote distinct but arbitrary values to be used as tags in a disjoint union.

$\mathcal{S}$ -MATRIX GRAPHS. An $m \times n$ (read “ m by n ”) $\mathcal{S}$ -matrix M is a table of values in $\mathcal{S}$ with m rows and n columns. The entry in row i and column j is denoted M_i^j . For the default $\mathcal{S} = \mathbb{R}$ case, we just call this a *matrix*. Here are a 3×4 matrix, a 3×2 $\mathbb{B}$ -matrix (Boolean logic semiring), a 3×1 $\mathcal{RL}$ -matrix (regular language semiring), and a 1×2 matrix:

$$\begin{bmatrix} 1 & 3 & 0 & 0 \\ 6 & 2 & 4 & 5 \\ 0 & 8 & 9 & 7 \end{bmatrix} \quad \begin{bmatrix} \perp & \top \\ \top & \perp \\ \top & \top \end{bmatrix} \quad \begin{bmatrix} \{\mathfrak{a}, \mathfrak{c}\} \\ \{\mathfrak{g}\} \\ \{\mathfrak{t}, \mathfrak{u}\} \end{bmatrix} \quad \begin{bmatrix} 2 & 5 \end{bmatrix} \quad (17.50)$$

We allow m and n to be zero here. This might seem odd: what good are a 3×0 , 0×0 , or 0×3 tables of values?

$$\begin{array}{c} | \\ \\ \\ \end{array} \quad \bullet \quad \text{---} \quad (17.51)$$

Nonetheless, these will prove useful in practice, even if we are not especially good at depicting them.

We define an $\mathcal{S}$ -matrix graph to be a special kind of directed, bipartite graph,¹³ where each edge has a value in $\mathcal{S}$ assigned to it. The sources are called “row” nodes, the targets “column” nodes. The graph is *complete*: there is an edge from every “row” node to every “column” node. Specifically, an $m \times n$ (“ m by n ”) **$\mathcal{S}$ -matrix graph** $\langle m, n, G, A \rangle$ is a directed graph $G = \langle \mathcal{N}, \mathcal{E}, \varphi \rangle$ with $\mathcal{N} = [1..m] \sqcup [1..n]$, $\mathcal{E} = [1..m] \times [1..n]$, and $\varphi(i, j) = \langle \langle i, \mathfrak{r} \rangle, \langle j, \mathfrak{c} \rangle \rangle$ equipped with an $m \times n$ $\mathcal{S}$ -matrix A that associates the value $A_i^j \in \mathcal{S}$ with the edge $\langle i, j \rangle$ for $i \in [1..m]$ and $j \in [1..n]$. For the default $\mathcal{S} = \mathbb{R}$ case, we just call this a *matrix graph*. We often refer to the $\mathcal{S}$ -matrix graph $\langle m, n, G, A \rangle$ as G for short.

Figure 17.2 shows the matrix graphs associated with the matrices in (17.50), where we omit any edges whose value is $\mathbf{0}$. Three special cases of $\mathcal{S}$ -matrix graphs

¹³A graph where some nodes can only be sources and the rest can only be targets. See Puzzle 93.

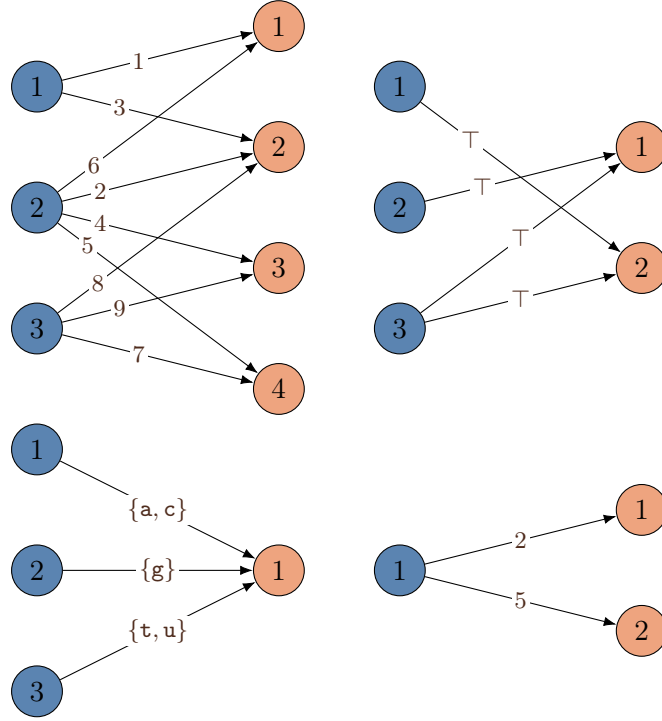


FIGURE 17.2. The matrix graph and $\mathbb{B}$ -matrix graph associated with the example matrices in (17.50). Edges weighted with 0 in the corresponding semiring are not shown.

suggest some optional but evocative names: (i) an $m \times 1$ $\mathcal{S}$ -matrix graph is an $\mathcal{S}$ -vector graph (of dimension m); (ii) an $1 \times n$ $\mathcal{S}$ -matrix graph is an $\mathcal{S}$ -covector graph (of dimension n); and (iii) a 1×1 $\mathcal{S}$ -matrix graph can be called a $\mathcal{S}$ -scalar graph. Moreover, we allow an $m \times n$ $\mathcal{S}$ -matrix graph to have $m = 0$ or $n = 0$ (or both). In these cases, there are simply no row or column nodes (or both), and therefore no edges. Figure 17.3 shows the graphs in (17.51) that have this property. The Figure highlights the missing nodes in these cases, though we will not generally do that.

We will define three operations on $\mathcal{S}$ -matrix graphs, each of which produces an output graph from one or more input graphs. These operations all have a tangible interpretation in terms of the graphs, which makes it easy to visualize and compute them. We can then define the matrix associated with the output graph as the result of a corresponding operations on the matrices associated with the input graphs. The three operations are

1. Product. Join two graphs left to right, identifying the “column” nodes of the first graph with the “row” nodes of the second graph. We aggregate values across all paths between each pair of outer nodes, producing a new graph.
2. Direct Sum. Stack two graphs atop one another, adding 0 edges from each

“row” node in one graph to each “column” node in the other graph.

3. Transpose. Swap the position of “row” and “column” nodes and reverse the arrows.

The product will define matrix multiplication, and the other two operations expand our toolkit in interesting ways. We begin with the product.



FIGURE 17.3. The $\mathcal{S}$ -matrix graphs with zero dimensions associated with the matrices in (17.51). Empty parts of the three graphs (all of the 0×0) are shown here in gray.

THE PRODUCT OF $\mathcal{S}$ -MATRIX GRAPHS. Let $\langle m, n, G, A \rangle$ and $\langle n, p, H, B \rangle$ be $\mathcal{S}$ -matrix graphs. Notice that the number of G ’s “column” nodes in G equals the number of H ’s “row” nodes. We will define the product GH to be a new $\mathcal{S}$ -matrix graph $\langle m, p, C \rangle$. For this to be defined, the number of G ’s “column” nodes in G must equal the number of H ’s “row” nodes. The key question: What is C ? We will define this matrix to be the *matrix product* AB .

To compute the product GH , we first join the graphs, with G on the left, identifying the “column” nodes of G with the “row” nodes of H . Each node on the left will be a “row” node in the product; each node on the right will be a “column” node in the product. For each such pair, we aggregate the values over *all directed paths* between them. Along each path, we combine the edge values with $\boxtimes$, and across paths, we aggregate the resulting products with $\boxplus$. (Aggregating zero terms with $\boxplus$ is defined to give $\mathbf{0}$.) The resulting value is assigned to the edge between this pair of nodes in the product.

Figure 17.4 illustrates the product of two matrix graphs G and H . The identified nodes (the “column” nodes of G and the “row” nodes of H) are shown in gray. $\boxtimes$ is just multiplication in this case, and $\boxplus$ is addition. The Figure shows the paths isolate for each pair of nodes and the resulting calculation along with the product graph. Although we usually do not show $\mathbf{0}$ -valued edges for clarity, here we do to

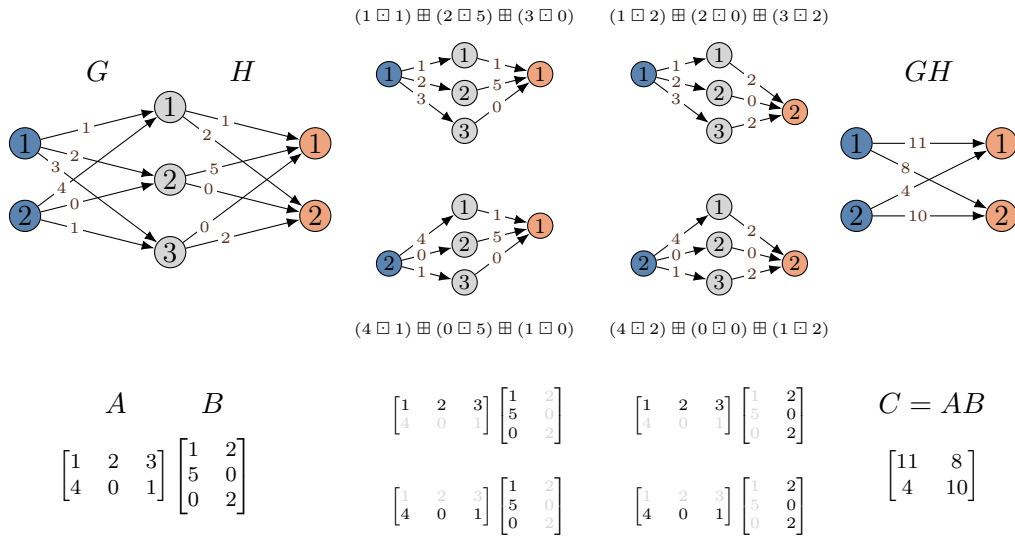


FIGURE 17.4. Illustration of the matrix graph product of two $\langle \mathbb{R}, +, \cdot, 0, 1 \rangle$ matrix graphs. Edges with zero values are shown in the intermediate graphs to ease matching. On the left, the two graphs being multiplied are joined at their common nodes (in gray), with the corresponding matrices shown below. In the middle, the aggregated paths are isolated for each pair of final nodes, with the calculation made explicit and the matrix parts highlighted below. And on the right is the product graph, with its corresponding matrix below.

make it easier to track the steps. The matrices corresponding to the graphs are shown below, with the relevant parts highlighted for part of the calculations. The matrix C associated with the product of graphs is defined as the product of matrices A and B . Notice that each of the four sub-calculations depicted in the middle column corresponds to a product of *subgraphs* of the original graphs: specifically, the 1×3 and a 3×1 $\mathcal{S}$ -matrix graph obtained by selecting one “row” node of G and one “column” node of H .

Figure 17.5 shows a similar example for two $\langle \mathbb{B}, \vee, \wedge, \perp, \top \rangle$ -matrix graphs.¹⁴ The process is the same, though here we do not show the edges with $\perp$ (false) value to avoid clutter. If we imagine the nodes in the graphs represent cities and that there is an edge from city i to city k if there is a route from i to k . Then the product of G and H tells us if there is a route from any of G ’s “row cities” to each of H ’s “column cities” through the cities they have in common. More generally, a Boolean-logic-matrix graph represents a **relation** on the set of “row” nodes and the set of “column” nodes, i.e., a subset of $[1..m] \times [1..n]$.¹⁵ A pair $\langle i, j \rangle$ belongs to this relation when there is a $\top$ edge from “row” node i to “column” node j . We can see then that the product of two Boolean-logic-matrix graphs is just the **composition of the two relations** as

¹⁴We call $\langle \mathbb{B}, \vee, \wedge, \perp, \top \rangle$ the *Boolean logic semiring*.

¹⁵See Chapter 16.

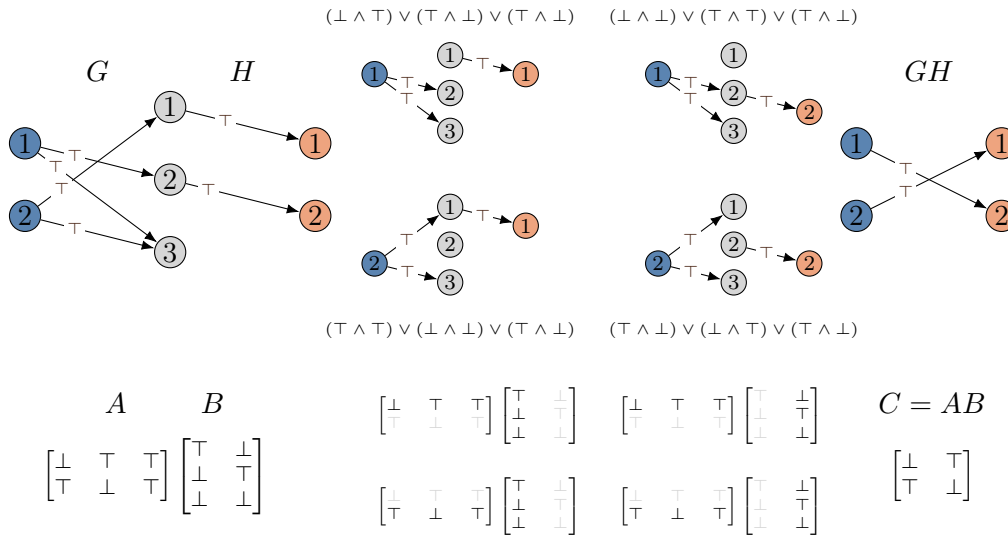


FIGURE 17.5. Illustration of the matrix graph product of two $\langle \mathbb{B}, \vee, \wedge, \perp, \top \rangle$ matrix graphs. Edges with value $\top$ are shown; edges with value $\perp$ are not. See the text and previous figure for an explanation of the layout here.

defined in equation (16.1). There is a $\top$ -edge between nodes i and k in the product if there is an intermediate (gray) node j with $\top$ -edge $\langle i, j \rangle$ in the first graph and $\langle j, k \rangle$ in the second graph.

Remember that the product of two graphs is well defined when the graphs are *compatible*: the number of “column” nodes in the first graph must equal the number of “row” nodes in the second graph. These are the nodes that are identified and become the intermediate nodes in the paths from the new “row” nodes to the new “column” nodes. Thus, the product of $m \times p$ and a $p \times n$ graph is an $m \times n$ graph. In particular, a covector graph of dimension m times a vector graph of dimension m is a scalar graph. An $0 \times p$ times a $p \times n$ gives a $0 \times n$ and similarly, $m \times p$ times $p \times 0$ gives $m \times 0$. The odd looking case of a $m \times 0$ times a $0 \times n$ still works; the result is an $m \times n$ graph with all edges $\mathbf{0}$ as seen below.

Puzzle 139. Create an $m \times 1$ graph and a $1 \times n$ graph for any $m, n > 1$ and any semiring of your choosing. What is the product of that graph? What does the corresponding matrix look like?

In general, the edge in GH from “row” node i to “column” node j is computed by aggregating a score for every path from i to j in the graph joining G and H . Each such path goes through one of the intermediate (identified, gray) nodes, and the

weights along the path are combined with $\boxdot$. So, the edge from i to j has value

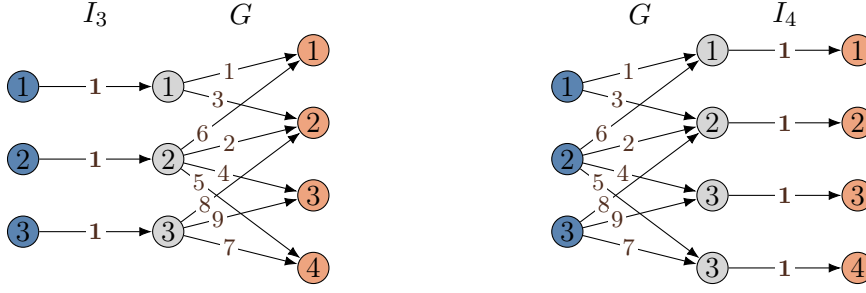
$$\bigoplus_{k=1}^p (A_i^k \boxdot B_k^j).$$

If $p = 0$, the $\bigoplus$ is empty and thus has value $\mathbf{0}$. (We give $\boxdot$ higher precedence than $\bigoplus$ in what follows.) Each term in this “sum” operates over a matched pair of a raised index (superscript) with a lowered index (subscript), here k .

As we have seen throughout this section, there are several immediate questions we should ask whenever we introduce a new operation. First, is it associative? This means that given three $\mathcal{S}$ -matrix graphs $\langle m, p, G, A \rangle$, $\langle p, q, H, B \rangle$, and $\langle q, n, K, C \rangle$, we should get the same $m \times n$ graph from $(GH)K$ and $G(HK)$. This follows from the semiring axioms, and in particular associativity of $\bigoplus$ and $\boxdot$ commutativity of $\bigoplus$, and the distributive property. The edge $\langle i, j \rangle$ in $(GH)K$ has associated value:

$$\begin{aligned} & \bigoplus_{\ell=1}^q \left(\bigoplus_{k=1}^p (A_i^k \boxdot B_k^\ell) \right) \boxdot C_\ell^j \\ &= \bigoplus_{\ell=1}^q \bigoplus_{k=1}^p \left((A_i^k \boxdot B_k^\ell) \boxdot C_\ell^j \right) && \text{Distributive property} \\ &= \bigoplus_{\ell=1}^q \bigoplus_{k=1}^p \left(A_i^k \boxdot (B_k^\ell \boxdot C_\ell^j) \right) && \text{Associativity of } \boxdot \\ &= \bigoplus_{k=1}^p \bigoplus_{\ell=1}^q \left(A_i^k \boxdot (B_k^\ell \boxdot C_\ell^j) \right) && \text{Commutativity of } \bigoplus \\ &= \bigoplus_{k=1}^p \left(A_i^k \boxdot \bigoplus_{\ell=1}^q (B_k^\ell \boxdot C_\ell^j) \right), && \text{Distributive property} \end{aligned}$$

which is the value on the $\langle i, j \rangle$ edge of $G(HK)$. Second, does the operation have an identity element? That is, we seek a graph – or graphs – such that $GI = G$ or $IG = G$. The compatibility condition makes it clear that the best we can hope for is an identity graph for each number of nodes. And unlike most things in life, here we get the best we could hope for. For each $n \in [0..)$, define the graph I_n to be the $n \times n$ $\mathcal{S}$ -matrix graph where $\mathbf{1}$ is assigned to each edge $\langle i, i \rangle$ for $i \in [1..n]$ and $\mathbf{0}$ is assigned to the remaining edges. For example, $I_3G = G = GI_4$ as follows:



because the **1** edges in I leave each edge weight untouched. Third, is the operation commutative? We can see immediately that the answer is no. HG need not even be defined, even if GH is. Moreover, $\square$ need not be commutative, so combining paths in the opposite direction can give different results.

Our graph product defines a matrix product. We define the matrix AB to have entries corresponding to the edge values for the graph GH .

If A is an $m \times p$ $\mathcal{S}$ -matrix and B is an $p \times n$ $\mathcal{S}$ -matrix, then the product AB is an $m \times n$ $\mathcal{S}$ -matrix whose entry in row i and column j is

$$(AB)_i^j = \bigoplus_{k=1}^p A_i^k \square B_k^j, \quad (17.52)$$

for $i \in [1 \dots m]$ and $j \in [1 \dots n]$. If $p = 0$, the empty $\bigoplus$ has value **0**.

This is an associative but not commutative product that is defined for compatible matrices. For each $n \in [0 \dots \infty)$, the identity graph above corresponds to the **identity matrix** I , where

$$I_i^j = \mathbf{1} \cdot \{i = j\}.$$

We use a subscript I_n to indicate the dimension only when needed to disambiguate.

Below, we will examine interpretations of this matrix product in different semirings and with different types of matrices. But before that, we define the other graph operations.

THE DIRECT SUM OF $\mathcal{S}$ -MATRIX GRAPHS. Given two such graphs, say G and H , their *direct sum* is the $\mathcal{S}$ -matrix graph $G \oplus H$ defined as follows. First, we simply stack G and H atop each other with their “row” nodes aligned on the left and “column” nodes aligned on the right. Second, we fill in the missing edges, putting a **0** edge from every “row” node in G to every “column” node in H and from every “row”

node in H to every “column” node in G . In the pictures, we then renumber the nodes and edges in the direct sum in order with G ’s nodes followed by H ’s nodes.

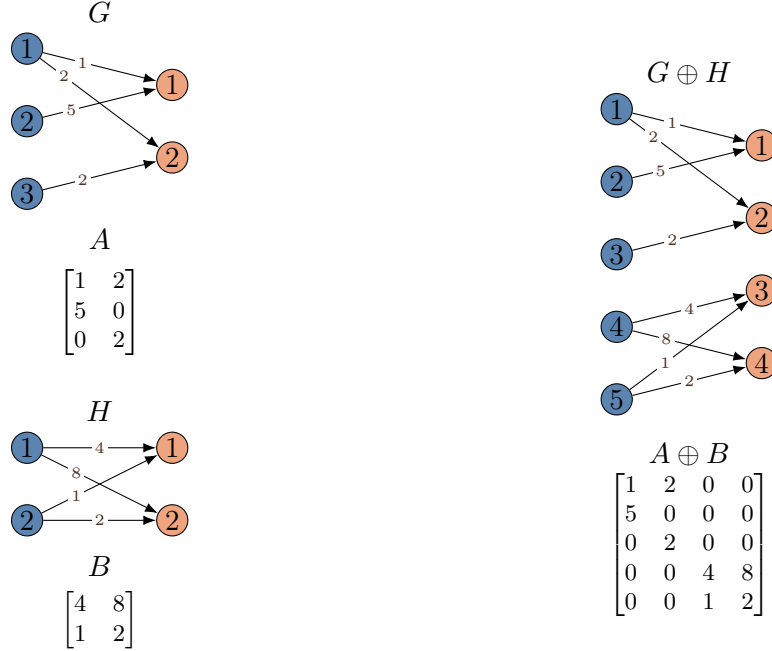


FIGURE 17.6. Illustration of the direct sum of two $\langle \mathbb{R}, +, \cdot, 0, 1 \rangle$ matrix graphs. Edges with zero values are not shown. On the left, the two graphs being joined, with the corresponding matrices shown below. On the right is the direct sum graph, with its corresponding matrix below.

Figure 17.6 shows an example of the graph direct sum with the corresponding matrices. Because $\mathbf{0}$ -edges are not shown (though they exist), the direct sum of the graphs simply stacks the two graphs and rennumbers the nodes. The matrix $A \oplus B$ has a copy of A in the upper left 3×2 block and a copy of B in the lower right 2×2 block, with the rest filled in with $\mathbf{0}$.

If $\langle m_1, n_1, G, A \rangle$ and $\langle m_2, n_2, H, B \rangle$ are $\mathcal{S}$ -matrix graphs, then $\langle m_1, n_1, G, A \rangle \oplus \langle m_2, n_2, H, B \rangle$, or $G \oplus H$ for short, is an $(m_1 + m_2) \times (n_1 + n_2)$ $\mathcal{S}$ -matrix graph. Letting $m = m_1 + m_2$ and $n = n_1 + n_2$, the node set of this graph is $[1 \dots m] \sqcup [1 \dots n]$ and edges set is $[1 \dots m] \times [1 \dots n]$. The edge values are given by the matrix $A \oplus B$, the *direct sum* of A and B , with

$$(A \oplus B)_i^j = \begin{cases} A_i^j & \text{if } i \in [1 \dots m_1] \text{ and } j \in [1 \dots n_1] \\ B_{i-m_1}^{j-n_1} & \text{if } i \in (m_1 \dots m] \text{ and } j \in (n_1 \dots n] \\ \mathbf{0} & \text{otherwise.} \end{cases} \quad (17.53)$$

The direct sum is associative but not quite commutative: $G \oplus H$ and $H \oplus G$ differ only in which graphs nodes are listed first, a simple rearrangement. The direct sum does have an identity element, the 0×0 $\mathcal{S}$ -matrix graph (there is only one), because stacking on a graph with no nodes does not change the graph. The $m \times 0$ and $0 \times n$ $\mathcal{S}$ -matrix graphs are useful here. If H is an $m \times 0$ graph, then $G \oplus H$ adds m “row” nodes to G ; if H is a $0 \times n$ graph then $G \oplus H$ adds n “column” nodes to G ; Figure 17.7 shows examples of this with the corresponding matrices.

Puzzle 140. If $\langle 1, 1, G, [1] \rangle$ is an $\mathcal{S}$ -matrix graph, what is

$$\overbrace{G \oplus G \oplus \cdots \oplus G}^{k \text{ times}}?$$

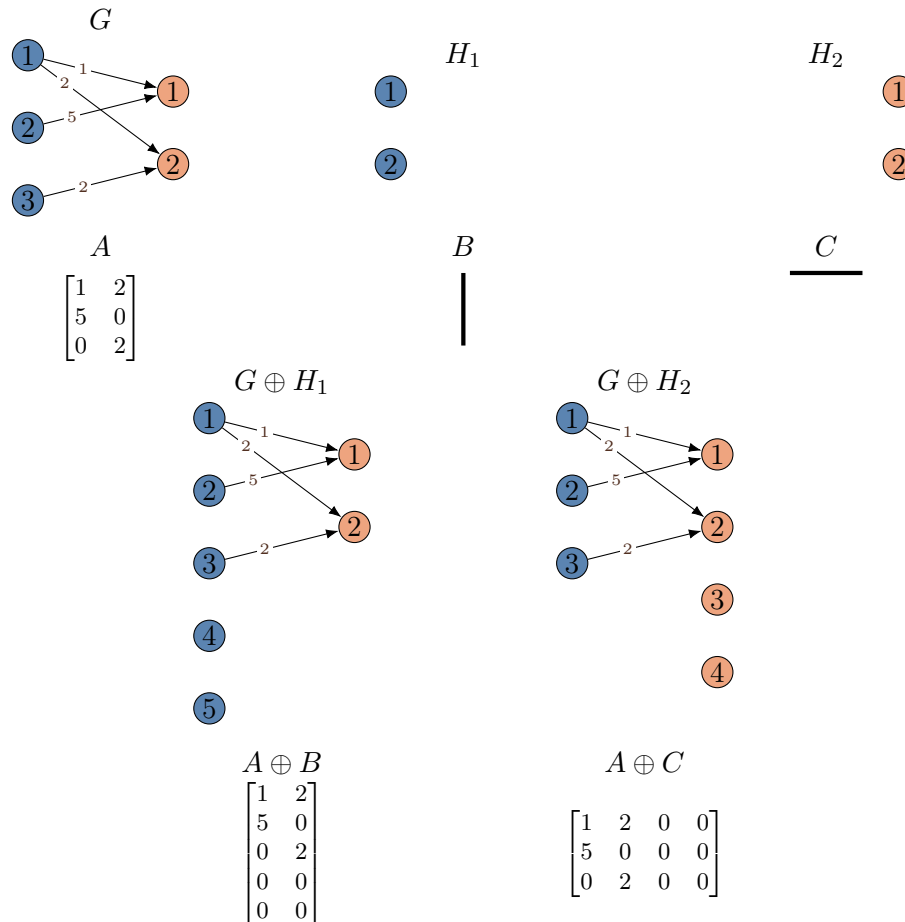


FIGURE 17.7. The direct sum of a matrix graphs with some dimensions zero. Edges with zero values are not shown.

The direct sum of graphs thus defines a direct sum operation on matrices. It creates a copy of the matrices in distinct blocks.

Suppose A is an $m_1 \times n_1$ $\mathcal{S}$ -matrix and B is an $m_2 \times n_2$ $\mathcal{S}$ -matrix. Let $m = m_1 + m_2$ and $n = n_1 + n_2$.

Then, $A \oplus B$ is an $m \times n$ $\mathcal{S}$ -matrix, with a copy of A in the $[1 \dots m_1] \times [1 \dots n_1]$ sub-matrix, a copy of B in the $(m_1 \dots m) \times (n_1 \dots n)$ submatrix, and $\mathbf{0}$ in all other entries. Equation (17.53) gives an equivalent description.

Puzzle 141. Referring to Figure 17.7, what is $H_1 \oplus H_2$? What is $B \oplus C$?

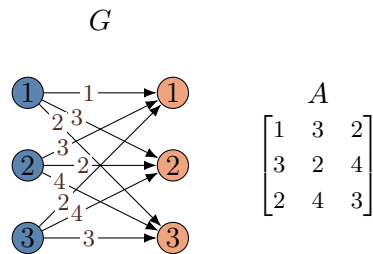
Puzzle 142. Suppose that, for $i \in \{1, 2\}$, A_i is $m_i \times p_i$ and B_i is $p_i \times n_i$. Is the following true?

$$(A_1 \oplus A_2)(B_1 \oplus B_2) = A_1 B_1 \oplus A_2 B_2.$$

Why or why not?

THE TRANSPOSE OF AN $\mathcal{S}$ -MATRIX GRAPH. The **transpose** operation exchanges “row” and “column” nodes. If $\langle m, n, G, A \rangle$ is an $\mathcal{S}$ -matrix graph, the transpose is an $n \times m$ $\mathcal{S}$ -matrix graph that reverses the arrows on all edges, though the associated values are unchanged. We show the graph with the new “row” nodes on the left as usual. Figure 17.8 shows an example.

We call an $\mathcal{S}$ -matrix graph G **symmetric** if $G = G^T$. A symmetric graph necessarily has the same number of “row” and “column” nodes and the value associated with edge $\langle i, j \rangle$ must equal that associated with $\langle j, i \rangle$. This gives the graph a “symmetric” appearance as seen in the following example:



The corresponding matrix has entries $A_i^j = A_j^i$.

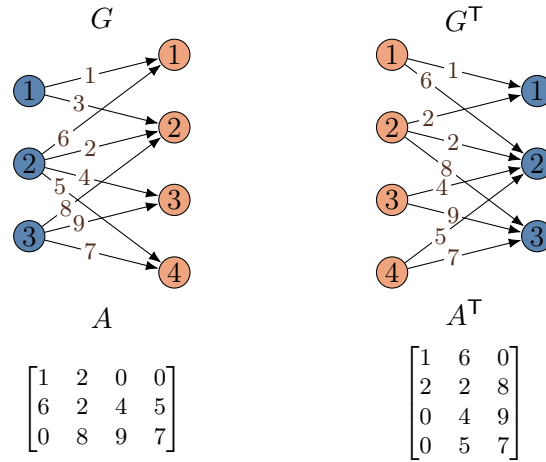


FIGURE 17.8. A Min-Sum-semiring matrix graph and its transpose, with the corresponding matrices displayed below.

The **transpose** of an $m \times n$ $\mathcal{S}$ -matrix is an $n \times m$ $\mathcal{S}$ -matrix A^T with entries

$$(A^T)_i^j = A_j^i, \quad (17.54)$$

for $i \in [1 \dots n]$ and $j \in [1 \dots m]$.

An $n \times n$ matrix is **symmetric** if $A = A^T$.

APPLICATIONS WITH VARIOUS SEMIRINGS. Varying the semiring gives different interpretations for these operations on graphs (and on the corresponding matrices).

With these operations in hand, we can now put them to use in different contexts and semirings. (Example 17.26 discusses several of the semirings we use here.) Working in the Min-Sum semiring, let the following matrices represent the costs (in hundreds of \$) of a one-way ticket between cities

$$\begin{array}{c}
 \begin{matrix} D & E & F \\ A & \begin{bmatrix} 7 & \infty & 9 \end{bmatrix} \\ B & \begin{bmatrix} 10 & \infty & 5 \end{bmatrix} \\ C & \begin{bmatrix} 4 & 8 & \infty \end{bmatrix} \end{matrix}
 &
 \begin{matrix}
 G & H & I \\
 D & \begin{bmatrix} 10 & 9 & 11 \end{bmatrix} \\
 E & \begin{bmatrix} 11 & 7 & \infty \end{bmatrix} \\
 F & \begin{bmatrix} 6 & 14 & 11 \end{bmatrix}
 \end{matrix}
 \end{array}$$

To compute the minimum cost from each of cities A, B, C to cities G, H, I, we take

the matrix product with $\boxplus = \min$ and $\boxdot = +$. This gives us

$$\begin{array}{c} \textcolor{red}{G} \quad \textcolor{red}{H} \quad \textcolor{red}{I} \\ \textcolor{red}{A} \begin{bmatrix} 17 & 16 & 18 \end{bmatrix} \\ \textcolor{red}{B} \begin{bmatrix} 11 & 19 & 16 \end{bmatrix} \\ \textcolor{red}{C} \begin{bmatrix} 14 & 13 & 15 \end{bmatrix} \end{array}$$

For instance, we look at all paths from A to H through each intermediate city, and find the minimum of $7 + 11$, $\infty + \infty$, and $9 + 11$, giving 18. For each pair of nodes, the product computes the minimum total cost, summed along each path between the nodes.

Next, consider a system that moves randomly among different states, making a transition at each tick of a clock. Let the matrix P below show the probability of making a transition from each state to each other state

$$\begin{array}{c} \textcolor{red}{1} \quad \textcolor{red}{2} \quad \textcolor{red}{3} \\ \textcolor{red}{1} \begin{bmatrix} 0.25 & 0.50 & 0.25 \end{bmatrix} \\ \textcolor{red}{2} \begin{bmatrix} 0.30 & 0.30 & 0.40 \end{bmatrix}, \\ \textcolor{red}{3} \begin{bmatrix} 0.80 & 0.00 & 0.20 \end{bmatrix} \end{array}$$

where the entry in row i and column j is the chance of the machine moving next to state j when it is currently in state i . (For this reason, each row sums to 1 because the machine must move to *some* state.) Think of the graph associated with this matrix and its product with itself $P^2 = PP$.

$$\begin{array}{c} \textcolor{red}{1} \quad \textcolor{red}{2} \quad \textcolor{red}{3} \\ \textcolor{red}{1} \begin{bmatrix} 0.4125 & 0.275 & 0.3125 \end{bmatrix} \\ \textcolor{red}{2} \begin{bmatrix} 0.485 & 0.24 & 0.275 \end{bmatrix} \\ \textcolor{red}{3} \begin{bmatrix} 0.36 & 0.4 & 0.24 \end{bmatrix} \end{array}$$

This product aggregates over all paths from one state to another through an intermediate state. Hence, each entry tells us the probability of moving from one state to another *in two steps*. Similarly, the product P^3 aggregates over all paths from one state to another through an two intermediate states, and the product P^n aggregates over all paths from one state to another through an $n - 1$ intermediate states. Hence, each entry of P^n tells us the probability of moving from one particular state to another *in n steps*.

We saw earlier that Boolean-logic matrices represent binary relations and the

product of two such matrices is the *composition* of those relations. Thus, we can think of composition of relations as aggregating conjunctions over paths.

A similar story works with the Max-Min semiring. We can view a matrix with values in this semiring as the capacities in a channel between two entities. The matrix product considers all possible paths between two nodes. The semiring product along a path gives the minimum capacity along that path, which is the best that path can support. So the matrix product computes the maximum flow along any given channel.

With the Max-Product semiring, we can use matrices containing a measure of the connection reliability between nodes. The matrix product then computes the reliability of paths.

And in the default $\mathbb{R}$ semiring, we can think of nodes as basis vectors and the matrix product as paths between the output basis vectors and input basis vectors with the score being the contribution of the input vector to the coefficient of the output vector. This is discussed in detail in the next section.

Puzzle 143. The graph view of matrix multiplication gives us insight into the structure of common computations. For each of the following cases, sketch some graphs that illustrate the product and state what the corresponding matrix looks like:

1. An $m \times 1$ vector graph times a $1 \times n$ covector graph.
2. An $m \times 1$ vector graph times a scalar graph.
3. A $1 \times n$ covector graph times an $n \times 1$ vector graph.
4. A $1 \times n$ covector graph times an $n \times m$ matrix graph times an $m \times 1$ vector graph.
5. We get an $m \times m$ **diagonal matrix** by taking the direct sum of the 1×1 matrix $[c]$ with itself m times. The product of an $m \times m$ diagonal matrix graph with an $m \times n$ matrix.
6. The product of an $m \times n$ matrix graph with an $n \times n$ diagonal matrix graph.

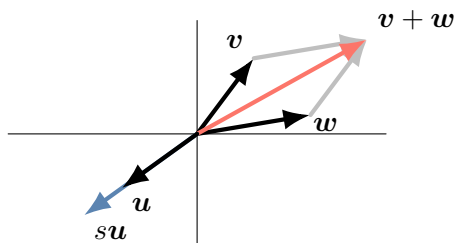


FIGURE 17.9. The operations of scaling and adding vectors illustrated.

17.3 Vector Spaces and Linearity

The idea of a *vector* was introduced to provide a geometric representation of quantities endowed with a *magnitude* and *direction*, such as complex numbers, forces, velocities, and so forth. In this setting, vectors are drawn as arrows whose length is their magnitude and whose direction is their ... direction. The two basic operations on these arrows is *adding them*, tail to tip, and *scaling them*, multiplying their magnitude by some constant factor while keeping the direction unchanged. See Figure 17.9. In adding the vectors, we shift one so its tail starts at the tip of the other; the sum is the vector from the origin to the combined tip. Notice that it does not matter which we shift onto which.

VECTORS AS TUPLES. While pictures like this can be useful, it is easier to work with vectors when we recognize that we can represent them by a *tuple* of numbers giving coordinates for their tip. For instance, $\mathbf{v} = \langle \frac{3}{4}, 1 \rangle$, $\mathbf{w} = \langle \frac{3}{2}, \frac{1}{4} \rangle$, and $\mathbf{v} + \mathbf{w} = \langle \frac{9}{4}, \frac{5}{4} \rangle$; and similarly, $\mathbf{u} = \langle -1, -0.72 \rangle$ with $s\mathbf{u} = \langle -s, -0.72s \rangle$. And we can see that our operations on arrows correspond to adding the respective components of the tuples and to scaling all the components of the tuples by the same number. What works in two dimensions, works in any number of dimensions, so we get numeric vectors by endowing $\mathbb{R}^n$ for any n with the *structure* of adding and scaling vectors:

$$\langle a_1, a_2, \dots, a_n \rangle + \langle b_1, b_2, \dots, b_n \rangle = \langle a_1 + b_1, a_2 + b_2, \dots, a_n + b_n \rangle \quad (17.55)$$

$$s\langle a_1, a_2, \dots, a_n \rangle = \langle ca_1, ca_2, \dots, ca_n \rangle, \quad (17.56)$$

for real number s .¹⁶ We *lifted* addition from the components (real numbers) to the vectors, overloading the operator as $+: \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$. This addition satisfies the expected properties: it is commutative $u + v = v + u$ and associative $u + (v + w) = (u + v) + w$ for $u, v, w \in \mathbb{R}^n$ and it has an identity element – the **zero vector**. The

¹⁶This is the source of the word *scalar*: a value that can be used to scale vectors.

zero vector of dimension n is just the tuple containing n zeros: $\langle 0, 0, \dots, 0 \rangle$. We usually denote it just by 0 when clear from context, but to distinguish it as a vector, we write $\mathbf{0}$.

VECTOR SPACES AND LINEAR TRANSFORMATIONS. The structure on $\mathbb{R}^n$ induced by vector addition and scaling makes $\mathbb{R}^n$ into what we call a *vector space*. The essence of that structure generalizes: anytime we recognize it on a set of objects we can treat those objects like vectors.

A set $\mathcal{V}$ endowed with addition ($\mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V}$) and scaling ($\mathbb{R} \times \mathcal{V} \rightarrow \mathcal{V}$) operations and a special element $\mathbf{0} \in \mathcal{V}$ is called a **vector space** when

1. addition is commutative and associative;
2. $\mathbf{0}$ is an identity element for addition:

$$v + \mathbf{0} = v = \mathbf{0} + v, \quad \text{for } v \in \mathcal{V}. \quad (17.57)$$

3. Scaling by 1 is id: for every $v \in \mathcal{V}$, $1v = v$.
4. Every $v \in \mathcal{V}$, has an *additive inverse* $-v \in \mathcal{V}$:

$$v + (-v) = \mathbf{0}. \quad (17.58)$$

5. Scaling is compatible with scalar multiplication:

$$(rs)v = r(sv), \quad \text{for } v \in \mathcal{V} \quad \text{and} \quad r, s \in \mathbb{R}. \quad (17.59)$$

6. Scalar multiplication and addition distributes over each other:

$$s(u + v) = su + sv \quad (17.60)$$

$$(s + r)u = su + ru, \quad (17.61)$$

for $u, v \in \mathcal{V}$ and $s, r \in \mathbb{R}$.

We often refer to generic elements of a vector space as *vectors* and to the real numbers that scale them as *scalars*.

These properties of vector addition and scaling¹⁷ make these operations on vectors act just like the vector operations on tuples of numbers. For instance, they imply that $-v = (-1)v$, $0v = \mathbf{0}$ and $s\mathbf{0} = \mathbf{0}$.

And indeed, $\mathbb{R}^n$ with component-wise addition and scaling of tuples forms a

¹⁷Scaling is also called *scalar multiplication*.

vector space that is consistent with the pictures of arrows that we started with. As vectors spaces, $\mathbb{R}^n$ has extra structure that does not follow from the vector-space axioms. In particular, on $\mathbb{R}^n$ we can define the *magnitude* of a vector:

$$|v| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}, \quad (17.62)$$

for $v = \langle v_1, \dots, v_n \rangle \in \mathbb{R}^n$. And from this we can compute the distance between two points in $u, v \in \mathbb{R}^n$ by $|v - u|$.¹⁸ Many vector spaces have such additional structure, but it is not required.

¹⁸In a general vector space, we define subtraction by $v - u = v + -u$.

Example 17.31 Functions Form a Vector Space

Consider the set $\mathcal{A} \rightarrow \mathcal{V}$ of functions from a set $\mathcal{A}$ to a vector space $\mathcal{V}$. We define the vector space operations by *lifting* them to $\mathcal{A} \rightarrow \mathcal{V}$ from $\mathcal{V}$ as follows:

$$\begin{aligned} (f + g)(a) &= f(a) + g(a) \\ (sf)(a) &= sf(a), \end{aligned}$$

for $f, g: \mathcal{A} \rightarrow \mathcal{V}$, $a \in \mathcal{A}$, and $s \in \mathbb{R}$ (see Aside 14). The identity element is const_0 , because $f(a) + \text{const}_0(a) = f(a) + 0 = f(a)$. All the vector properties we need are inherited from $\mathcal{V}$ because they hold for each $a \in \mathcal{A}$.

It follows that $\mathcal{A} \rightarrow \mathcal{V}$ is a vector space.

Example 17.32 Matrices Form a Vector Space

As we saw in Section 17.2, *matrices* are rectangular tables of numbers; an $m \times n$ matrix has m rows and n columns for $m, n \in \mathbb{N}$. For example,

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 4 \end{bmatrix}$$

are 2×2 , 3×3 , and 1×1 matrices.

Let $\mathcal{M}_{m \times n}$ be the set of $m \times n$ real matrices. Given two such matrices A

and B , we define addition and scaling entry-wise, e.g.,

$$\begin{aligned} \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} a' & b' \\ c' & d' \end{bmatrix} &= \begin{bmatrix} a + a' & b + b' \\ c + c' & d + d' \end{bmatrix} \\ s \begin{bmatrix} a & b \\ c & d \end{bmatrix} &= \begin{bmatrix} sa & sb \\ sc & sd \end{bmatrix}, \end{aligned}$$

with the matrix of all 0's as identity element. Because we are just adding and scaling numbers in each entry, $\mathcal{M}_{m \times n}$ inherits all the vector space properties with these operations and so is a vector space.

Note though that we can only add matrices of the *same shape*.

Example 17.33 A One-Dimensional Vector Space

The real numbers $\mathbb{R}$ form a vector space. We can view this as the set of 1-tuples, which is a vector space by construction. But notice that we can view the numbers themselves as the vectors; that is, we treat the set of numbers and the set of 1-tuples of numbers as isomorphic.

Example 17.34 Polynomials and Sequences

Polynomials are functions of type $\mathbb{R} \rightarrow \mathbb{R}$ in the form $(a_0 + a_1x + a_2x^2 + \cdots + a_nx^n)$, where n is the polynomial's *degree* and the a_i 's are its *coefficients*. Adding together two polynomials of any degree gives another polynomial, and multiplying a polynomial by a constant gives another polynomial (of the same degree unless the constant is 0). Negating a polynomial's coefficients gives its additive inverse, and the constant function const_0 is a 0-degree polynomial that acts as an identity element. These operations on polynomials thus satisfy the axioms of a vector space, as in Example 17.31, so the set of polynomials with real coefficients, $\text{Poly}(\mathbb{R})$, is a vector space.

Similarly, sequences (cf. Example 10.20) are functions $\mathbb{N} \rightarrow \mathbb{R}$ and so by Example 17.31, $\text{Seq}(\mathbb{R})$ is a vector space. In fact, Table 10.5 already defined the addition and scaling operations.

Example 17.35 A Trivial Space

A set with only one element forms a trivial vector space. Call the element $\mathbf{0}$ with $\mathbf{0} + \mathbf{0} = \mathbf{0}$ and $s\mathbf{0} = \mathbf{0}$. All the remaining properties are satisfied, so we have a

vector space.

Given the structure of a vector space, what kinds of functions *preserve* that structure? If we have a vector space $\mathcal{V}$ and two vectors $u, v \in \mathcal{V}$, the vector space operations allow us to form a **linear combination** $au + bv$ for scalars a, b . Any structure-preserving function $f: \mathcal{V} \rightarrow \mathcal{W}$ between vector spaces $\mathcal{V}$ and $\mathcal{W}$ must map a linear combination to a linear combination as

$$f(au + bv) = af(u) + bf(v), \quad (17.63)$$

for $u, v \in \mathcal{V}$ and $a, b \in \mathbb{R}$. Such an f will respect all the vector-space properties. For instance, $f(-v) = -f(v)$, and f preserves the identity elements $f(\mathbf{0}) = \mathbf{0}$ where the $\mathbf{0}$'s denote the identity elements in $\mathcal{V}$ and $\mathcal{W}$. Repeated application of equation (17.63), shows that a f preserves linear combinations $a_1\mathbf{v}^1 + a_2\mathbf{v}^2 + \cdots + a_n\mathbf{v}^n$ with any number of terms:¹⁹

$$f(a_1\mathbf{v}^1 + a_2\mathbf{v}^2 + \cdots + a_n\mathbf{v}^n) = a_1f(\mathbf{v}^1) + a_2f(\mathbf{v}^2) + \cdots + a_nf(\mathbf{v}^n). \quad (17.64)$$

We call f a **linear function**.

Example 17.36 Covectors

Which functions $\omega: \mathbb{R}^n \rightarrow \mathbb{R}$ with $n \in [1..)$ are linear? To determine this, let $\mathcal{E}_j$ for $j \in [1..n]$ be the set of vectors that can be non-zero only in the j th component. Every $v \in \mathcal{E}_j$ is of the form $v = a\mathbf{e}^j$ for some real a , where

$$\mathbf{e}^j = \overbrace{\langle 0, \dots, 0, 1, 0, \dots, 0 \rangle}^{j-1 \text{ components}}. \quad (17.65)$$

Because ω is linear, $\omega(a\mathbf{e}^j) = a\omega(\mathbf{e}^j)$. Now, if $v = \langle v_1, v_2, \dots, v_n \rangle \in \mathbb{R}^n$, we can write $v = v_1\mathbf{e}^1 + v_2\mathbf{e}^2 + \cdots + v_n\mathbf{e}^n$ because both sides have the same components. Linearity gives us

$$\omega(v) = \sum_{i=1}^n \omega(\mathbf{e}^i) v_i.$$

Every linear function $\mathbb{R}^n \rightarrow \mathbb{R}$ is of this form and is determined by the tuple of numbers $\langle \omega(\mathbf{e}^1), \dots, \omega(\mathbf{e}^n) \rangle$, which we also call ω . This tuple represents a function on vectors. This function – and the tuple that represents it – is called a **covector**. We index the *components* of vectors with subscripts and the components of covectors with superscripts; we index *collections* of vectors with

¹⁹If v is a vector, then v_j denotes its j th component; for a vector in an indexed collection, however, v^j is the j th vector and v_k^j is its k th component. When there are many indices about, we sometimes visually distinguish terms that are vectors (as opposed to numbers), like $\mathbf{v}^j$.

These superscripts are not exponents; more on such indexing shortly.

superscripts and collections of covectors with subscripts. Hence, we can write

$$\omega(v) = \sum_{i=1}^n \omega^i v_i. \quad (17.66)$$

Notice that the sum matches exactly one raised and one lowered script index.

From this argument, we actually get more. What are all the linear functions $L: \mathbb{R}^n \rightarrow \mathbb{R}^m$? Well, for every $v \in \mathbb{R}^n$, $L(v)$ is a vector in $\mathbb{R}^m$, so L has m component functions $L_1, \dots, L_m: \mathbb{R}^n \rightarrow \mathbb{R}$ that are *linear*. Each L_i is thus a covector, with components L_i^j , where $L_i = \sum_{j=1}^n L_i^j \mathbf{e}_j$. So for any vector $v \in \mathbb{R}^n$ and covector $w \in \mathbb{R}^m$, we can write

$$\begin{aligned} w(L(v)) &= w(\langle L_1(v), \dots, L_m(v) \rangle) \\ &= \sum_{i=1}^m w^i \left(\sum_{j=1}^n L_i^j v_j \right) \\ &= \sum_{i=1}^m \sum_{j=1}^n w^i L_i^j v_j. \end{aligned} \quad (17.67)$$

Again, notice how each sum matches a raised and lowered index. This implies

$$\begin{aligned} L(v) &= \left\langle \sum_{j=1}^n L_1^j v_j, \dots, \sum_{j=1}^n L_i^j v_j, \dots, \sum_{j=1}^n L_m^j v_j \right\rangle \\ w \circ L &= \sum_{i=1}^m w^i L_i^j \mathbf{e}_j. \end{aligned}$$

We will explore the implications of these equations shortly.

Puzzle 144. Is $\text{const}_0: \mathcal{V} \rightarrow \mathbb{R}$ a linear function on a vector space $\mathcal{V}$? Why?

If $\mathcal{V}$ is a vector space, then we refer to an element $v \in \mathcal{V}$ as a vector. A **covector** is a *linear* function $\mathcal{V} \rightarrow \mathbb{R}$. Think of a covector as extracting a feature from a vector in a way that is consistent with the vector-space structure. For instance, the covector $\mathbf{e}_4$ from the previous example extracts the fourth component of a vector, and the covector $\frac{1}{2}\mathbf{e}_1 + \frac{1}{2}\mathbf{e}_2$ computes the average of the first two components. Given a vector v , if we know the value of $w(v)$ for enough covectors, we know v . The interaction between vectors and covectors is fundamental.

There can be many indices to keep track of in our calculations, so to make it

easier we hold firmly to an indexing convention with two parts. First, for referring to *components*: we use subscripts to index vector components and superscripts to index covector components. Second, for referring to *collections* this reverses: we use superscripts to index collections of vectors and subscripts to index collections of covectors. The utility of this convention will become clearer as we proceed, but in the interim notice that every sum in what follows is over matched pairs of a raised index (superscript) and a lowered index (subscript). We will discuss this further below.

Vectors $\mathbf{v}^1, \dots, \mathbf{v}^n \in \mathcal{V}$ are said to be **linearly independent** if *no non-trivial linear combination of these vectors can equal zero*; that is,

$$a_1 \mathbf{v}^1 + a_2 \mathbf{v}^2 + \dots + a_n \mathbf{v}^n = \mathbf{0} \implies a_1 = a_2 = \dots = a_n = 0. \quad (17.68)$$

A set of vectors that is not linearly independent, is *linearly dependent*.

If a vector space $\mathcal{V}$ contains a set of n linearly independent vectors but no set of $n + 1$ linearly independent vectors, we say it has **dimension** n . A vector space need not have a finite dimension, and when it does we call it *finite dimensional*. For instance, the vector space $\mathbb{R}^n$ has dimension n , one dimension per component, but the vector space of polynomials is infinite dimensional because it contains the polynomials x^n for *any* degree n .

Puzzle 145. If $\mathcal{V}$ is a finite-dimensional vector space, let $\mathcal{V}^*$ denote the set of covectors on $\mathcal{V}$ (i.e., linear functions $\mathcal{V} \rightarrow \mathbb{R}$).

Then $\mathcal{V}^*$ forms a vector space. How? What are the addition and scaling operations? What is the identity element?

If $\mathcal{V}$ has dimension n , what is the dimension of $\mathcal{V}^*$? What does linear independence look like for a collection of covectors in $\mathcal{V}^*$?

If a vector space $\mathcal{V}$ has dimension n , then given a set of n linearly independent vectors $\mathbf{v}^1, \dots, \mathbf{v}^n$, we can express *any* $\mathbf{u} \in \mathcal{V}$ as a linear combination of those vectors. This follows because $\mathbf{u}, \mathbf{v}^1, \dots, \mathbf{v}^n$ are linearly *dependent*, having more than n vectors, so there exist numbers $a, a_1, \dots, a_n$ with $a \neq 0$ so that $a\mathbf{u} + a_1\mathbf{v}^1 + \dots + a_n\mathbf{v}^n = \mathbf{0}$. This implies that

$$\mathbf{u} = -(a_1/a)\mathbf{v}^1 - (a_2/a)\mathbf{v}^2 - \dots - a_n/a\mathbf{v}^n := b_1\mathbf{v}^1 + b_2\mathbf{v}^2 + \dots + b_n\mathbf{v}^n.$$

From this property, we say that $\mathbf{v}^1, \dots, \mathbf{v}^n$ comprise a *basis* for the vector space.

A set of n linearly independent vectors in an n -dimensional vector space $\mathcal{V}$ is called a **basis** for $\mathcal{V}$. If $\mathbf{v}^1, \mathbf{v}^2, \dots, \mathbf{v}^n$ forms a basis for $\mathcal{V}$, then for any $u \in \mathcal{V}$, we can write

$$u = a_1 \mathbf{v}^1 + a_2 \mathbf{v}^2 + \dots + a_n \mathbf{v}^n \quad (17.69)$$

for some choice of scalars $a_1, \dots, a_n$.

For any basis $\mathbf{v}^1, \mathbf{v}^2, \dots, \mathbf{v}^n$ for $\mathcal{V}$, we can define the **dual basis** consisting of covectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ for which

$$\mathbf{v}_i(\mathbf{v}^j) = \{i = j\} \quad (17.70)$$

That is, if $u = a_1 \mathbf{v}^1 + a_2 \mathbf{v}^2 + \dots + a_n \mathbf{v}^n$, $\mathbf{v}_i(u) = a_i$. Any covector $w: \mathcal{V} \rightarrow \mathbb{R}$ can be written

$$w = c^1 \mathbf{v}_1 + c^2 \mathbf{v}_2 + \dots + c^n \mathbf{v}_n \quad (17.71)$$

for some choice of scalars $c^1, \dots, c^n$.

In $\mathbb{R}^n$, the vectors $\mathbf{e}^j$ from equation (17.65), with j th component 1 and other components 0, forms a basis called the **standard basis** because it decomposes a vector component by component. That is, any $\mathbf{v} = \langle v_1, \dots, v_n \rangle \in \mathbb{R}^n$ can be written $\mathbf{v} = v_1 \mathbf{e}^1 + v_2 \mathbf{e}^2 + \dots + v_n \mathbf{e}^n$. There are corresponding covectors $\mathbf{e}_j$ so that for any $v \in \mathbb{R}^n$, $\mathbf{e}_j(v) = v_j$ extracts the j th component of v , and hence

$$\mathbf{e}_j(\mathbf{e}^i) = \{i = j\}. \quad (17.72)$$

The $\mathbf{e}_j$'s comprise the **standard dual basis**. As shown in Example 17.36, any covector on $\mathbb{R}^n$ can be represented by a tuple, and in this case $\mathbf{e}_j$ and $\mathbf{e}^j$ have *the same underlying tuples*. We will distinguish these (see below) by arranging vector tuples as single columns and covector tuples as single rows.

Puzzle 146. The standard basis is often convenient, but it is not unique. Any n linearly independent vectors form a basis for $\mathbb{R}^n$. If $\mathbf{v}^1 = \mathbf{e}^1$ and

$$\mathbf{v}^j = \langle \overbrace{1, 0, \dots, 0}^{j-1 \text{ components}}, 1, 0, \dots, 0 \rangle$$

for $j \in [2..n]$, do these vectors form a basis of $\mathbb{R}^n$? If so, how is vector $u = \langle u_1, \dots, u_n \rangle$ written as a linear combination of these basis vectors?

Example 17.37 Subspaces

In the vector space of polynomials $\text{Poly}(\mathbb{R})$ in Example 17.34, the polynomials $\langle x \rangle \mapsto x^n$ for $n \in \mathbb{N}$ are linearly independent because the only polynomial equal to const_0 is one whose coefficients are all zero.

The polynomials of degree 10 form a vector space $\text{Poly}_{10}(\mathbb{R})$ that is contained in $\text{Poly}(\mathbb{R})$. The polynomials $\langle x \rangle \mapsto x^n$ for $n \in [0..10]$ form a basis for this space, and a linear combination of these basis vectors is a polynomial

$$a_0 + a_1x + a_2x^2 + \cdots + a_{10}x^{10},$$

by which we can express any polynomial of degree 10, which includes all those with degree < 10 .

Define a function $p: \text{Poly}(\mathbb{R}) \twoheadrightarrow \text{Poly}_{10}(\mathbb{R})$ by

$$p\left(\sum_{k=0}^n a_k x^k\right) = \sum_{k=0}^{\min(n,10)} a_k x^k.$$

This is a linear function as you should confirm. Notice for instance that $p(ax^{17} + bx^{101}) = ap(x^{17}) + bp(x^{101})$. p maps a polynomial to the degree 10 polynomial that most closely approximates its input. The inclusion from $\text{Poly}_{10}(\mathbb{R})$ into $\text{Poly}(\mathbb{R})$ is a pre-inverse for p .

If $\mathcal{V}$ is a vector space and $\mathcal{W} \subseteq \mathcal{V}$ is also a vector space with the same operations and identity element, we say that $\mathcal{W}$ is a **subspace** of $\mathcal{V}$. It is a subset that *inherits* the structure of the containing space.

Given vector spaces $\mathcal{U}$ and $\mathcal{V}$, we can construct a new vectors space using tuples of vectors in $\mathcal{U} \times \mathcal{V}$: $\langle u, v \rangle + \langle u', v' \rangle = \langle u + u', v + v' \rangle$ and $c\langle u, v \rangle = \langle cu, cv \rangle$, for $u, u' \in \mathcal{U}$, $v, v' \in \mathcal{V}$, and scalar c . The components here need not be numbers, but it all works as expected. The resulting vector space is called the **direct sum** and usually denoted $\mathcal{U} \oplus \mathcal{V}$.²⁰ The direct sum of any number of vector spaces can be defined. There is a direct relationship between the direct sum of vector spaces and the direct sum operation we defined in Section 17.2, as we will see.

²⁰The underlying set is still $\mathcal{U} \times \mathcal{V}$; the $\oplus$ evokes the vector space structure directly.

TUPLES AGAIN: VECTORS, COVECTORS, AND MATRICES. Vector spaces can be built from a diverse range of objects, and recognizing the vector-space structure gives us an interface to those objects that can guide and sharpen our analysis. In most cases that we will deal with, the vector spaces will be *finite-dimensional*, and

this allows us to represent objects and operations in terms of *tuples of numbers*.

The key to this is equation (17.67). There we saw that any linear function $L: \mathbb{R}^n \rightarrow \mathbb{R}^m$ can be represented by the numbers L_i^j for $i \in [1..m], j \in [1..n]$. Arranging these numbers into a table with m rows and n columns and entry L_i^j in row i and column j gives us an $m \times n$ **matrix**. This matrix *represents* the linear function L in the sense that any calculation we can do with L can be expressed with the matrix. Specifically, for any covector w on $\mathbb{R}^m$ and any vector v on $\mathbb{R}^n$,

$$w(L(v)) = \sum_{i=1}^m \sum_{j=1}^n w^i L_i^j v_j,$$

This equation completely specifies L , and thus so does its matrix. The left side does not depend on a choice of basis; it evaluates the linear function $w \circ L$ on the vector v , giving a number. The right side does depend on choice of bases; both the matrix and the components of w and v are computed in terms of the standard bases on $\mathbb{R}^n$ and $\mathbb{R}^m$.

Next, we formalize and generalize this idea of finding a convenient *representation* of a linear function between vector spaces; convenient that is, for doing our calculations. For this purpose, we restrict our attention to *finite-dimensional* vector spaces. It is conventional (and convenient) to omit parentheses when evaluating *linear functions* unless needed for grouping. So, we can write Lv for $L(v)$ and wLv for $w(L(v))$. Similarly, it is convenient to write composition of linear functions by juxtaposition, omitting the $\circ$ operator. So, we can write KL for $K \circ L$ and wL for $w \circ L$.

Suppose that $\mathcal{V}$ and $\mathcal{W}$ are n and m dimensional vector spaces and $L: \mathcal{U} \rightarrow \mathcal{V}$ is linear. The key result is that **for each choice of bases** for $\mathcal{V}$ and $\mathcal{W}$ there is an $m \times n$ matrix $[L]$ such that linear calculations involving L can be translated exactly to linear calculations involving $[L]$. Indeed, we can view $[L]$ as a linear function $\mathbb{R}^n \rightarrow \mathbb{R}^m$. The translation looks like

$$\begin{array}{ccc} \mathcal{V} & \xrightarrow{L} & \mathcal{W} \\ \downarrow & & \downarrow \\ \mathbb{R}^n & \xrightarrow{[L]} & \mathbb{R}^m \end{array}$$

converting operations in $\mathcal{V}$ and $\mathcal{W}$ into operations on $\mathbb{R}^n$ and $\mathbb{R}^m$ using the *standard bases*. Moreover, this mapping from a linear function L to its matrix $[L]$ relative to specific bases respects both the vector-space structure and the laws of function composition. For instance, if $\mathcal{U}$ is a p -dimensional vector space on which we have chosen a basis and if $K: \mathcal{W} \rightarrow \mathcal{U}$ is linear, the following diagram commutes

$$\begin{array}{ccccc}
 \mathcal{V} & \xrightarrow{L} & \mathcal{W} & \xrightarrow{K} & \mathcal{U} \\
 \downarrow & & \downarrow & & \downarrow \\
 \mathbb{R}^n & \xrightarrow{[L]} & \mathbb{R}^m & \xrightarrow{[K]} & \mathbb{R}^p \\
 & \searrow & & \nearrow & \\
 & & [KL] & &
 \end{array}$$

giving $[KL] = [K][L]$. Similarly, on $\mathcal{V}$, $[\text{id}]$ translates to the identity on $\mathbb{R}^n$. Thus, **all n -dimensional vector spaces look like $\mathbb{R}^n$ with the standard basis** when it comes to evaluating and composing linear functions.

Now let's look more closely at where the matrix $[L]$ for a linear function L comes from. The essential idea is this: **each entry of $[L]$ gives the contribution to one component of L 's output vector from one component of its input vector.**

Let $\mathbf{x}^1, \dots, \mathbf{x}^n$ be the chosen basis for $\mathcal{V}$, and let $\mathbf{y}^1, \dots, \mathbf{y}^m$ be the chosen basis for $\mathcal{W}$ with corresponding dual basis $\mathbf{y}_1, \dots, \mathbf{y}_m$. Then, for any $v \in \mathcal{V}$, $Lv = \sum_{j=1}^n v_j L\mathbf{x}^j$, and each $L\mathbf{x}^j$ can be written as a linear combination of $\mathbf{y}^i$'s. To get the entries of the matrix $[L]$, we extract the component of each $\mathbf{y}^i$ from each $L\mathbf{x}^j$. Hence, we define

$$[L]_i^j = \mathbf{y}_i L\mathbf{x}^j. \quad (17.73)$$

This says that the number in the i th row and j th column of the matrix $[L]$ is component of $\mathbf{y}^i$ in the vector to which L maps $\mathbf{x}^j$. You can see from this that the matrix depends very concretely on the choice of bases for $\mathcal{V}$ and $\mathcal{W}$. If you change bases, you will generally get different numbers in the entries of the matrix.

From equation (17.73), we can compute for any covector on $\mathcal{W}$, $w = \sum_{i=1}^m w^i \mathbf{y}_i$, and any vector on $\mathcal{V}$, $v = \sum_{j=1}^n v_j \mathbf{x}^j$:

$$\begin{aligned}
 wLv &= wL \left(\sum_{j=1}^n v_j \mathbf{x}^j \right) \\
 &= w \left(\sum_{j=1}^n v_j L\mathbf{x}^j \right) \\
 &= \left(\sum_{i=1}^m w^i \mathbf{y}_i \right) \left(\sum_{j=1}^n v_j L\mathbf{x}^j \right) \\
 &= \sum_{i=1}^m w^i \mathbf{y}_i \left(\sum_{j=1}^n v_j L\mathbf{x}^j \right)
 \end{aligned}$$

$$\begin{aligned}
&= \sum_{i=1}^m \sum_{j=1}^n w^i \mathbf{y}_i L \mathbf{x}^j v_j \\
&= \sum_{i=1}^m \sum_{j=1}^n w^i [L]_i^j v_j,
\end{aligned} \tag{17.74}$$

using the linearity of L and the covectors $\mathbf{y}_i$. Equation (17.74) characterizes L completely, and the result of any linear calculation involving L can be derived from it. For instance, by choosing w to be each $\mathbf{y}_i$ in turn, we see that

$$Lv = \sum_{i=1}^m \left(\sum_{j=1}^n [L]_i^j v_j \right) \mathbf{y}^i$$

and by choosing v to be each $\mathbf{x}^j$ in turn that

$$wL = \sum_{j=1}^n \left(\sum_{i=1}^m w^i [L]_i^j \right) \mathbf{x}_j,$$

where the $\mathbf{x}_j$'s are the covectors dual to the $\mathbf{x}^j$'s.

This reveals that $[L]$ can itself be viewed as a linear function $\mathbb{R}^n \rightarrow \mathbb{R}^m$. If $a = \langle a_1, a_2, \dots, a_n \rangle \in \mathbb{R}^n$, then $[L]a$ is an m -tuple with i th component $\sum_{j=1}^n [L]_i^j a_j$, which is linear in a .

We can go the other way as well. If M is an $m \times n$ matrix – a rectangular table of real numbers, then for each choice of bases for $\mathcal{V}$ and $\mathcal{W}$, there is a linear function $L_M: \mathcal{V} \rightarrow \mathcal{W}$ with $M = [L_M]$. To find this, we simply mimic equation (17.74) and define

$$wL_M v = \sum_{i=1}^m \sum_{j=1}^n w^i M_i^j v_j,$$

for all covectors w on $\mathcal{W}$ and vectors v in $\mathcal{V}$, with components expressed *in the chosen bases*. This determines L_M uniquely.

Matrices represent linear functions

1. A matrix is a representation of a linear function between vector spaces with respect to a chosen bases on the input and output spaces.
2. Linear calculations with the function on the original vector spaces can be translated exactly to linear calculations with the matrix.
3. For every matrix M and each choice of bases on the domain and codomain vector spaces, there is a unique linear function from domain to codomain

whose matrix equals M .

4. A matrix can be viewed as a linear function from some $\mathbb{R}^n$ to some $\mathbb{R}^m$.
5. The matrix representing the composition of linear functions is the composition of the matrices representing the original functions.

This composition is expressed as a *product* of matrices, as we will see.

MATRIX ALGEBRA. The lesson so far is that we can translate calculations on general vector spaces to calculations with tuples of real numbers. In this light, we turn to some practical implementation of these calculations with matrices, vectors, and covectors on the vector-spaces $\mathbb{R}^n$ for various n . When working on the vector-spaces $\mathbb{R}^n$, we will by default use the standard bases (and standard dual bases) to represent matrices, vectors, and covectors unless otherwise stated.

Puzzle 147. What is the matrix representation of the identity function id on $\mathbb{R}^n$ with the standard bases?

We will represent matrices as rectangular tables of numbers delimited by $[]$'s. We write $m \times n$ matrix for a table with m rows and n columns. We represent a d -dimensional *vector* as a $1 \times d$ matrix, that is, a single *column*. We represent a d -dimensional *covector* as a $d \times 1$ matrix, that is, a single *row*. For example,

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

give the vector and covector, respectively, with contents $\langle 1, 2, 3, 4 \rangle$. Both vectors and covectors have an associated tuple of numbers,²¹ and it is often convenient to use the tuple to describe their contents. For instance, saying the vector $\langle 1, 2, 3, 4 \rangle$ or the covector $\langle 1, 2, 3, 4 \rangle$ tells us how those contents should be arranged and interpreted. Remember that despite this tuple representation, a covector represents a linear function and using equation (17.66), evaluating the covector $\langle w^1, w^2, w^3, w^4 \rangle$ at the vector $\langle v_1, v_2, v_3, v_4 \rangle$ gives $w^1 v_1 + w^2 v_2 + w^3 v_3 + w^4 v_4$. With our bracket notation

²¹Reminder: expressions of vector or covector components are always with respect to specific chosen bases.

this becomes

$$\begin{bmatrix} w^1 & w^2 & w^3 & w^4 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = w^1 v_1 + w^2 v_2 + w^3 v_3 + w^4 v_4,$$

where again we use juxtaposition without parentheses for the evaluation of linear functions.

It is time now for a more explicit comment on index placement. As mentioned earlier, we use subscripts for vector components and superscripts for covector components. And we use superscripts to index collections of vectors and subscripts to index collections of covectors. An $m \times n$ matrix M can be viewed either as a stack of m covectors (of dimension n) laid atop one another or as file of n vectors (of dimension m) laid side by side.²² Both views are useful and are reflected in the notation. We can think of this as a collection of covectors $M_1, M_2, \dots, M_m$ with M_i 's j th component M_i^j , or we can think of this as a collection of vectors $M^1, M^2, \dots, M^n$ with M^j 's i th component M_i^j . Our indexing convention is consistent both ways. This indexing convention has the additional nice feature that in sums when we apply a linear function or expand a vector or covector in a basis, we are summing over a *matched pair* of indices, one raised (superscript) and one lowered (subscript). This placement of the indices makes it easier to see what an equation means and easier to check your work.

²²Here "file" as in to columns on a chess board, "rank and file," or files in a cabinet.

Both of these ideas come together when computing Mv or wM for a vector v and covector w . Thinking of M as a stack of covectors M_i , then Mv is a stack of numbers $M_i v$ – a column. Thinking of M as a file of vectors M^j , then wM is a file of numbers – a row. We have

$$(Mv)_i = \sum_j M_i^j v_j \quad (17.75)$$

$$(wM)^j = \sum_i w^i M_i^j \quad (17.76)$$

$$wMv = \sum_i \sum_j w^i M_i^j v_j. \quad (17.77)$$

So, Mv is a vector with given components (lowered indices) in a column; wM is a covector with given components (raised indices) in a row; and wMv is a scalar. Here we see the matched pairs in each sum, and any indices not summed over occur on

both sides of the equation in the same position.

Equations (17.75), (17.76), and (17.77) describe what is commonly called *matrix-vector multiplication* (along with covector-matrix multiplication) but which we see is just the application and composition of linear functions. In terms of our bracket notation, these equations look like

$$\begin{aligned} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} &= \begin{bmatrix} v_1 + 2v_2 + 3v_3 \\ 4v_1 + 5v_2 + 6v_3 \end{bmatrix} \\ \begin{bmatrix} w^1 & w^2 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} &= \begin{bmatrix} w^1 + 4w^2 & 2w^1 + 5w^2 & 3w^1 + 6w^2 \end{bmatrix} \\ \begin{bmatrix} w^1 & w^2 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} &= w^1 v_1 + 2w^1 v_2 + 3w^1 v_3 + 4w^2 v_1 + 5w^2 v_2 + 6w^2 v_3. \end{aligned}$$

Notice that by associativity, in the left-hand side of the last equation, we can evaluate the pairs in either order and get the same result. More generally, we get

$$Mv = \begin{bmatrix} \sum_j M_1^j v_j \\ \sum_j M_2^j v_j \\ \dots \\ \sum_j M_m^j v_j \end{bmatrix} \quad wM = \begin{bmatrix} \sum_i w^i M_i^1 & \sum_i w^i M_i^2 & \dots & \sum_i w^i M_i^n \end{bmatrix}.$$

Notice that equation (17.77) generates all the others for suitable choice of covector w and vector v .

Puzzle 148. Find

$$\begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Answers:

(a) the vector $\langle 10, 10, 60 \rangle$ in a column in $\mathbb{R}^3$
 (b) the covector $\langle 0, 1, 2 \rangle$ in a row in $\mathbb{R}^3$

Vectors and covectors, despite being different types of objects, are intimately related. Not only do they have similar representations as tuples of numbers, but some vector spaces,²³ like $\mathbb{R}^n$, have an operation called the **transpose** that converts

²³This operation requires extra structure that not all vector spaces share. But $\mathbb{R}^n$ has it, as do most other vector spaces we deal with.

a vector to a covector of the same dimension and vice versa. The transpose operation is denoted with a raised label $^\top$. If v is an n -dimensional vector, then $v^\top$ is an n -dimensional covector; if w is an m -dimensional covector, then $w^\top$ is an m -dimensional vector. The transpose of a column is a row and vice versa. For any two vectors u and v of equal dimension, the transpose operation is defined to satisfy

$$u^\top v = v^\top u \quad (17.78)$$

and

$$v^\top v > 0 \quad \text{if and only if } v \neq \mathbf{0}. \quad (17.79)$$

In particular, if $\mathbf{e}^i$ is one of the standard basis vectors on $\mathbb{R}^n$, then we have $\mathbf{e}^{i^\top} = \mathbf{e}_i$, its dual covector, so $v^\top \mathbf{e}^i = \mathbf{e}^{i^\top} v = \mathbf{e}_i v = v_i$, the corresponding component of v in the standard basis. Hence, if u and v have tuple representations $\langle u_1, \dots, u_n \rangle$ and $\langle v_1, \dots, v_n \rangle$ in the standard basis, then

$$u^\top v = u_1 v_1 + u_2 v_2 + \dots + u_n v_n.$$

In particular,

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \iff v^\top = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}.$$

Hence, the transpose operator does not change the values in the underlying tuple of a vector or covector; it just changes our interpretation of those values and how we arrange them on the page.

If we think of a matrix M as a stack of covectors or a file of vectors, then $M^\top$ just exchanges vectors for covectors and vice versa – that is, it exchanges rows and columns. Thus, $(M^\top)_i^j = M_j^i$; the matrix's entries have been reflected about its main diagonal. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^\top = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^\top = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}.$$

This is exactly the same as the transpose operation discussed in Section 17.2. If M is an $m \times n$ matrix and v is an n -dimensional vector, then viewing M as a stack of m covectors, then Mv is an m -dimensional vector with components $M_i v$. It follows

from (17.78) and the associativity of function composition that for an m -dimensional vector u ,

$$(Mv)^T u = u^T (Mv) = \sum_{i=1}^m u_i (M_i v) = \left(\sum_{i=1}^m u_i M_i \right) v = v^T (M^T u) = v^T M^T u.$$

Since this holds for any u , we have that $(Mv)^T = v^T M^T$.

An $m \times n$ matrix is called **square** if $m = n$, in which case we say it has *dimension* n . A square matrix M is **symmetric** if $M = M^T$; that is $M_i^j = M_j^i$ for all i, j .

Puzzle 149. If A is a square matrix with dimension n and B is an $m \times n$ matrix, then all three of the following matrices are symmetric

1. $A + A^T$,
2. BB^T , and
3. $B^T B$.

Explain why.

The transpose operation gives us a type of product on pairs of vectors called the **dot product**. If u, v are two vectors in $\mathbb{R}^n$, we first convert one to a covector and then evaluate the covector at the other. The result is a number, and it does not matter which of the two vectors we apply the transpose to: $u^T v = v^T u$. The dot product is denoted by

$$u \cdot v = u^T v = v^T u. \quad (17.80)$$

This product has a geometric interpretation. First, the dot product lets us measure the “size” of a vector. Define

$$|v|^2 = v \cdot v. \quad (17.81)$$

Then $|v|$ is the *magnitude* of v , as in equation (17.62). When we depict a vector as an arrow, its magnitude is the length of that arrow. Notice that for any v and any real number r , $|rv| = |r||v|$; the length scales with the vector. In the special case where $|v| = 1$, we call v a **unit vector**. If v is any vector, then the scaled vector $\hat{v} = v/|v|$ is a unit vector that represents the *direction* of v .

Second, when we account for magnitudes, the dot product tells how the directions of two vectors relate. For vectors u and v ,

$$u \cdot v = |u||v| \cos(\theta), \quad (17.82)$$

where θ is the angle between their *directions*. When u and v point in perpendicular directions – that is, the angle between them is 90 degrees – then $u \cdot v = 0$, and we say that u and v are **orthogonal**. We have a useful fact

$$\begin{aligned} |u + v|^2 &= (u + v) \cdot (u + v) \\ &= u \cdot u + v \cdot u + u \cdot v + v \cdot v \\ &= u \cdot u + 2v \cdot u + v \cdot v. \end{aligned}$$

Hence,

$$|u + v|^2 = |u|^2 + |v|^2 + 2|u||v|\cos(\theta). \quad (17.83)$$

When u and v are orthogonal, we get *Pythagoras's Theorem*

$$|u + v|^2 = |u|^2 + |v|^2 \quad (u \text{ and } v \text{ orthogonal}) \quad (17.84)$$

generalized to n dimensions.

If $\mathbf{v}^1, \dots, \mathbf{v}^n$ are unit vectors with each pair orthogonal, i.e.,

$$\mathbf{v}^i \cdot \mathbf{v}^k = \{i = k\}, \quad (17.85)$$

we say that these vectors are **orthonormal**. The most important case of this is when the $\mathbf{v}^i$ form a *basis*, which we call an **orthonormal basis**. The standard basis $\mathbf{e}^1, \dots, \mathbf{e}^n$ for $\mathbb{R}^n$ is an example of an orthonormal basis. If we express a vectors x and y in an orthonormal basis, $x = \sum_{i=1}^n x_i \mathbf{v}^i$ and $y = \sum_{i=1}^n y_i \mathbf{v}^i$, then

$$\begin{aligned} x \cdot y &= \left(\sum_{i=1}^n x_i \mathbf{v}^i \right) \cdot \left(\sum_{k=1}^n y_k \mathbf{v}^k \right) \\ &= \sum_{i=1}^n \sum_{k=1}^n x_i y_k (\mathbf{v}^i \cdot \mathbf{v}^k) \\ &= \sum_{i=1}^n \sum_{k=1}^n x_i y_k \{i = k\} \\ &= \sum_{i=1}^n x_i y_i. \end{aligned} \quad (17.86)$$

In particular, $\mathbf{v}^i \cdot \mathbf{x} = x_i$, and we can read off the coefficients x_i in the expansion of x by taking successive dot products with the basis vectors. Note that the numeric values of x_i and y_i here depend on *which* basis we use. Put another way, the tuple

representation of a vector – and in particular the values of the tuple’s components – depend on the chosen basis. Although we use the standard basis by default when specifying the components of a vector, this is just a convention.

When working with vectors and linear functions in practice, we usually represent them in terms of some specific bases for the vector spaces, usually just the standard bases. This lets express our calculations in terms of matrices, which are concrete and convenient. Sometimes, however, we can simplify our calculations in some way by *changing* the bases we are using, at least temporarily. Changing the bases changes the numeric entries in our matrices, and it is often possible to find bases where the matrices have a particularly nice form that is either fast to compute with or easy to understand. Example 17.39 below shows an important illustration of this. In the following puzzle, you will explore how matrices and the tuple representations of vectors change when moving from basis to another.

Puzzle 150. Changing Bases. Suppose $\mathcal{X}$ is an n -dimensional vector space with a transpose operation on which we have two orthonormal bases $\mathbf{u}^i$ for $i \in [1..n]$ and $\mathbf{v}^i$ for $i \in [1..n]$.

We have a vector $x \in \mathcal{X}$ that has a tuple representation $\langle x_1, \dots, x_n \rangle$ for the basis $\mathbf{u}$. That is, $x = \sum_{i=1}^n x_i \mathbf{u}^i$. We would like to find x ’s tuple representation in the basis $\mathbf{v}$, $\langle x'_1, \dots, x'_n \rangle$, i.e., $x = \sum_{i=1}^n x'_i \mathbf{v}^i$.

Define the $n \times n$ matrix M by

$$M_i^j = \mathbf{u}^i \cdot \mathbf{v}^j. \quad (17.87)$$

Give an argument to demonstrate that

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = M \begin{bmatrix} x'_1 \\ \vdots \\ x'_n \end{bmatrix} \quad (17.88)$$

and

$$\begin{bmatrix} x'_1 \\ \vdots \\ x'_n \end{bmatrix} = M^T \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}. \quad (17.89)$$

Now, suppose that $L: \mathcal{X} \rightarrow \mathcal{Y}$ is a linear function and that $\mathcal{Y}$ is an m -dimensional vector space with a transpose operation. Analogously to $\mathcal{X}$, $\mathcal{Y}$ has

two orthonormal bases which we call the $\mathbf{s}$ and $\mathbf{t}$ bases. We compute the matrix A representing L with respect to the $\mathbf{u}$ basis on $\mathcal{X}$ and the $\mathbf{s}$ basis on $\mathcal{Y}$, and we want to find the matrix A' representing L with respect to the $\mathbf{v}$ basis on $\mathcal{X}$ and the $\mathbf{t}$ basis on $\mathcal{Y}$. We form the matrix M and the analogous matrix N that changes from $\mathbf{s}$ to $\mathbf{t}$. Use what you showed above to **explain why**

$$A' = N^\top A M \quad (17.90)$$

Remember that A takes an input expressed in the $\mathbf{u}$ basis and returns an output expressed in the $\mathbf{s}$ basis, but A' takes an input expressed in the $\mathbf{v}$ basis and returns an output expressed in the $\mathbf{t}$ basis.

If $v \neq \mathbf{0}$, we can define a covector $\langle x \rangle \mapsto \frac{v \cdot x}{v \cdot v}$ and a linear function $p_v: \mathbb{R}^n \rightarrow \mathbb{R}^n$ by

$$p_v(x) = \frac{v \cdot x}{v \cdot v} v. \quad (17.91)$$

This is called the **projection** onto v , and for any vector x , it gives the part of x along the line formed by v . See Figure 17.10 for several examples.

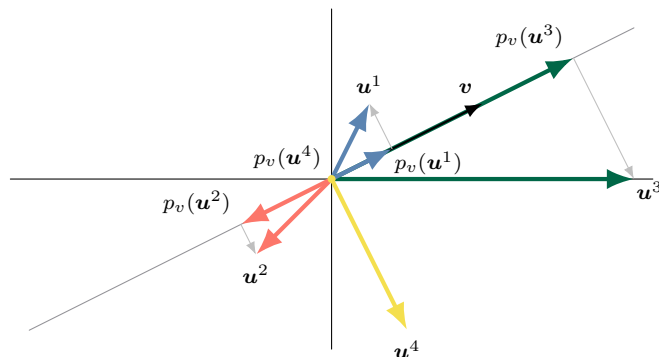


FIGURE 17.10. The projection of vectors $\mathbf{u}^1, \mathbf{u}^2, \mathbf{u}^3, \mathbf{u}^4$ onto a vector $\mathbf{v}$. Each vector is shown as an arrow from the origin. A thin gray arrow connects each vector to its projection, though $\mathbf{u}^4$ is orthogonal to $\mathbf{v}$ so its projection is the zero vector.

Puzzle 151. What is $p_v(v)$? What is $p_v(p_v(x))$? What is $p_v(x - p_v(x))$?

Answers: $\mathbf{0}$ (o) $!(x)^ad$ (q) $!a$ (e)

The projection p_v is useful for decomposing a vector x into a part that is *parallel* to v and a part that is *perpendicular* (i.e., orthogonal) to v :

$$x = p_v(x) + (x - p_v(x)).$$

You can see the second part in the gray arrows of Figure 17.10.

Puzzle 152. Draw a picture to explain why v and $x - p_v(x)$ are orthogonal.

Matrices represent linear functions, and the composition of linear functions is represented by the *product* of their matrices. We discussed *matrix multiplication* in detail in subsection 17.2, showing how to derive it by combining paths in graphs. Recall that if A is an $m \times p$ matrix and B is a $p \times n$ matrix, then their product AB is an $m \times n$ matrix with entries

$$(AB)_i^j = \sum_{k=1}^p A_i^k B_k^j. \quad (17.92)$$

(Notice again how the indexing convention works out here.) Here, we revisit matrix multiplication to connect the previous explanation to the ideas in this Section. In particular, we can think of matrix multiplication in terms of a stack of covectors or file of vectors, and we can think of it in terms of function composition.

Think of A as a stack of p -dimensional covectors $A_1, \dots, A_m$ and B as a file of p -dimensional vectors $B^1, \dots, B^n$. We evaluate each of the former on each of the latter, with each pair giving an entry of the product:

$$(AB)_i^j = A_i B^j, \quad (17.93)$$

which gives equation (17.92). Thus, the i, j th entry in AB is the result of applying the i th covector (row) in A to the j th vector (column) in B . In terms of the graphs in subsection 17.2, A_i is the subgraph of A 's graph from the i th input node with edges to all the output nodes, and B^j is the subgraph of B 's graph with all edges leading to the j th node. See Figure 17.11.

As we have seen, the product of two matrices A and B gives the matrix for the composition of the linear functions corresponding to A and B . Let's pull this apart by tracking an input vector through B and then through A . Equation (17.73) tells us that

$$(AB)_i^j = \mathbf{e}_i A B \mathbf{e}^j,$$

for $i \in [1 \dots m]$ and $j \in [1 \dots n]$. Considering this one step at a time, we have

$$\begin{aligned} B \mathbf{e}^j &= B_1^j \mathbf{e}^1 + \dots + B_p^j \mathbf{e}^p, \\ A \mathbf{e}^k &= A_1^k \mathbf{e}^1 + \dots + A_m^k \mathbf{e}^m, \end{aligned}$$

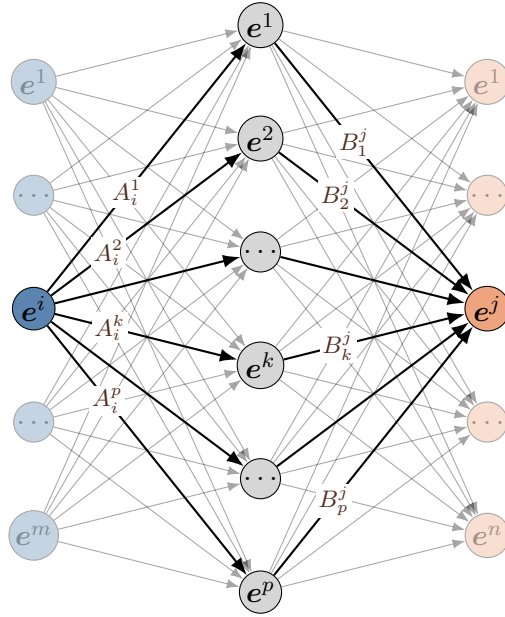


FIGURE 17.11. The graph interpretation of equation (17.93).

so

$$AB\mathbf{e}^j = \sum_{i=1}^m \sum_{k=1}^p A_i^k B_k^j \mathbf{e}^i$$

and

$$\mathbf{e}_i AB\mathbf{e}^j = \sum_{k=1}^p A_i^k B_k^j.$$

The j th basis vector in $\mathbb{R}^n$ is transformed by B into a linear combination of basis vectors in $\mathbb{R}^p$. Each of the latter basis vectors is in turn transformed into a linear combination of basis vectors in $\mathbb{R}^m$. If we look at the contribution to one of these output vectors from the original input vector, we combine $\mathbf{e}^j$'s contribution to each intermediate vector and each intermediate vector's contribution to $\mathbf{e}^i$ then add them up. This is just a version of the graph interpretation with a path from $\mathbf{e}^i$ through $\mathbf{e}^1, \dots, \mathbf{e}^p$ to $\mathbf{e}^j$.

This formula implies the useful fact that $(AB)^\top = B^\top A^\top$, which we can derive explicitly by viewing the matrices as stacks of covectors and files of vectors as before.

The identity function $\text{id}: \mathcal{V} \rightarrow \mathcal{V}$ is linear and from equation (17.73), its matrix $[\text{id}]$ satisfies

$$[\text{id}]_i^j = \mathbf{v}_i \text{id} \mathbf{u}^j = \{i = j\}.$$

This is the **identity matrix**, typically denoted by I or $\mathbb{I}$, occasionally with a subscript

to indicate its dimension. As the previous equation indicates, it is a square matrix with 1s on the main diagonal and 0 elsewhere

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \dots$$

Because matrix multiplication represents the composition of linear functions, if a matrix M represents an invertible function it is said to be an **invertible matrix** and it has an **inverse matrix** M^{-1} representing the inverse function and for which

$$MM^{-1} = I = M^{-1}M.$$

Notice that these are just the conditions defining inverse functions in (13.14) expressed in terms of matrices. An invertible matrix must be square, meaning that the domain and codomain vector spaces have the same dimension.

An invertible matrix M of dimension n satisfies $wMv \neq 0$ for every *non-zero* vector v of dimension n and every *non-zero* covector w of dimension n . If $wMv > 0$ for all such w, v , we say that M is **positive definite**. If instead it satisfies $wMv \geq 0$ for all such w, v , we say that M is **positive semi-definite**.

If M is symmetric and positive definite, then it is invertible, and we can find a *symmetric* matrices $M^{1/2}$, called its **symmetric square root**, and its inverse $M^{-1/2}$ for which

$$M^{1/2}M^{1/2} = M \tag{17.94}$$

$$M^{-1/2}M^{-1/2} = M^{-1} \tag{17.95}$$

The **determinant** of a square matrix M , denoted $\det M$, is a scalar associated with the matrix. (To get its value, we compute the image via M of the unit cube in $\mathbb{R}^n$ and take its signed volume, where the sign depends on its orientation.) $\det M$ is non-zero if and only if M is invertible. We have $\det I = 1$; $\det cM = c^n \det M$ where n is M 's dimension; and $\det J = -1$ if J is obtained from I by interchanging either one pair of rows or one pair of columns; and $\det(MQ) = \det M \cdot \det Q$ for square matrix Q of the same dimension. For a 2×2 matrix

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc,$$

and for an $n \times n$ *diagonal matrix*, which has non-zero entries only on the main diagonal

$$\det \begin{bmatrix} d_1 & 0 & 0 & \dots & 0 \\ 0 & d_2 & 0 & \dots & 0 \\ 0 & 0 & \ddots & \dots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & 0 & d_n \end{bmatrix} = d_1 d_2 \cdots d_n.$$

The **trace** of a square matrix is the sum of the diagonal terms

$$\text{Tr } M = \sum_i M_i^i.$$

A key property of the trace is that if the matrix products MQ and QM are both defined, then $\text{Tr}(QM) = \text{Tr}(MQ)$.

The next example many of these ideas on a novel vector space; it has a twist ending, so to speak.

Example 17.38 Not Your Imagination

Let $\mathcal{C}$ be the subset of square 2×2 matrices of the form

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix},$$

for real numbers a, b . We can make this a vector space as in Example 17.32, using entrywise addition and scaling by real numbers for the operations, i.e.,

$$u \begin{bmatrix} a & -b \\ b & a \end{bmatrix} + v \begin{bmatrix} c & -d \\ d & c \end{bmatrix} = \begin{bmatrix} ua + vc & -(ub + vd) \\ ub + vd & ua + vc \end{bmatrix}.$$

The matrix $\mathbf{0}$ with all zero entries is the zero vector and additive identity. We can also define a product $\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ via matrix multiplication, i.e.,

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} c & -d \\ d & c \end{bmatrix} = \begin{bmatrix} ac - bd & -(ad + bc) \\ ad + bc & ac - bd \end{bmatrix}.$$

As any value in $\mathcal{C}$ is specified by two numbers, this is a 2-dimensional vector space. A useful basis is

$$\mathbf{1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{i} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix},$$

and we can write any element as a linear combination of these two vectors:

$$a\mathbf{1} + b\mathbf{i} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}.$$

Notice that the value $\mathbf{1}$ is the multiplicative identity and

$$\mathbf{i}^2 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} = -\mathbf{1}$$

Negation gives an additive inverse to every element:

$$a\mathbf{1} + b\mathbf{i} + -a\mathbf{1} - b\mathbf{i} = 0\mathbf{1} + 0\mathbf{i} = \mathbf{0}.$$

Every non-zero $z \in \mathcal{C}$ has a multiplicative inverse as well, we can find it by taking the inverse of the underlying matrix

$$\begin{aligned} (a\mathbf{1} + b\mathbf{i})^{-1} &= \begin{bmatrix} a & -b \\ b & a \end{bmatrix}^{-1} = \begin{bmatrix} \frac{a}{a^2+b^2} & \frac{b}{a^2+b^2} \\ -\frac{b}{a^2+b^2} & \frac{a}{a^2+b^2} \end{bmatrix} \\ (a\mathbf{1} + b\mathbf{i}) \left(\frac{a}{a^2+b^2}\mathbf{1} - \frac{b}{a^2+b^2}\mathbf{i} \right) &= \mathbf{1}. \end{aligned}$$

This suggests an operation, which we will call conjugation, $\langle z \rangle \mapsto \bar{z}$ for $z \in \mathcal{C}$, where

$$\overline{a\mathbf{1} + b\mathbf{i}} = a\mathbf{1} - b\mathbf{i}.$$

We have

$$|a\mathbf{1} + b\mathbf{i}|^2 = (a\mathbf{1} + b\mathbf{i})(\overline{a\mathbf{1} + b\mathbf{i}}) = a^2 + b^2 = \det \begin{bmatrix} a & -b \\ b & a \end{bmatrix}.$$

Then, $1/z = \bar{z}/|z|^2$. The conjugate of z corresponds to the transpose of the underlying matrix:

$$\overline{a\mathbf{1} + b\mathbf{i}} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}^T = \begin{bmatrix} a & b \\ -b & a \end{bmatrix} = a\mathbf{1} - b\mathbf{i},$$

but note that this is *not* the transpose of $a\mathbf{1} + b\mathbf{i}$ in the vector space $\mathcal{C}$

So, the vector space $\mathcal{C}$ has the additional structure of a semiring where every element has an additive inverse and every element that is not zero (the additive identity) has a multiplicative inverse. As described at the end of Section 17.1, this makes $\mathcal{C}$ a *field*.

The transpose operation on the vector space $\mathcal{C}$ converts vectors to covectors. To find it, first note that $\langle z \rangle \mapsto \bar{z}$ is *linear*: $\overline{z + y} = \bar{z} + \bar{y}$ and $\overline{rz} = r\bar{z}$ as you can confirm. The dual basis of $\mathbf{1}$ and $\mathbf{i}$ is therefore:

$$\begin{aligned}\mathbf{1}^T(a\mathbf{1} + b\mathbf{i}) &= a \\ \mathbf{i}^T(a\mathbf{1} + b\mathbf{i}) &= b,\end{aligned}$$

which gives $\mathbf{1}^T\mathbf{1} = 1$, $\mathbf{i}^T\mathbf{i} = 1$, and $\mathbf{1}^T\mathbf{i} = 0 = \mathbf{i}^T\mathbf{1}$ as required. So if $z = a\mathbf{1} + b\mathbf{i}$, then

$$z^T = a\mathbf{1}^T + b\mathbf{i}^T = \langle c\mathbf{1} + d\mathbf{i} \rangle \mapsto ac + bd,$$

so

$$z^T z = |z|^2 = a^2 + b^2.$$

With $\mathbf{1}, \mathbf{i}$ as the standard basis and $\mathbf{1}^T, \mathbf{i}^T$ as the standard dual basis, any $z \in \mathcal{C}$ has a tuple representation, as does any covector z^T . By construction, both $\mathbf{1}$ and $\mathbf{1}^T$ have $\langle 1, 0 \rangle$ as their tuple representation, and both $\mathbf{i}$ and $\mathbf{i}^T$ have $\langle 0, 1 \rangle$. A value $a\mathbf{1} + b\mathbf{i} \in \mathcal{C}$ thus has tuple representation $\langle a, b \rangle$.

There is consequently a bijection between $\mathcal{C}$ and $\mathbb{R}^2$, mapping $a\mathbf{1} + b\mathbf{i}$ to $\langle a, b \rangle$ and vice versa. We can carry operations from $\mathcal{C}$ to $\mathbb{R}^2$. The addition and scaling operations on $\mathcal{C}$ induce the familiar vector-space operations on $\mathbb{R}^2$; $r(a\mathbf{1} + b\mathbf{i}) + s(c\mathbf{1} + d\mathbf{i}) = (ra + sc)\mathbf{1} + (rb + sd)\mathbf{i}$ implies that

$$r\langle a, b \rangle + s\langle c, d \rangle = \langle ra + sc, rb + sd \rangle.$$

The product and conjugate operations on $\mathcal{C}$ similarly imply

$$\langle a, b \rangle \langle c, d \rangle = \langle ac - bd, ad + bc \rangle \quad \text{and} \quad \overline{\langle a, b \rangle} = \langle a, -b \rangle,$$

and every $\langle a, b \rangle \neq \langle 0, 0 \rangle$ has a reciprocal. With this structure, $\mathbb{R}^2$ becomes a field that is *isomorphic* to $\mathcal{C}$.

We can define the *exponential* function $e^{\blacksquare} : \mathcal{C} \rightarrow \mathcal{C}$. This is based on the exponential matrix e^A of any square matrix A that is defined by

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

Applying this formula to the value $\theta \mathbf{i}$ for real θ and using that $\mathbf{i}^2 = -\mathbf{1}$, $\mathbf{i}^3 = -\mathbf{i}$, $\mathbf{i}^4 = \mathbf{1}$ we get

$$e^{\theta \mathbf{i}} = \begin{bmatrix} 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \cdots & -\theta + \frac{\theta^3}{3!} - \frac{\theta^5}{5!} + \frac{\theta^7}{7!} - \cdots \\ \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \cdots & 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \cdots \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

The tuple representation of the vector $e^{\theta \mathbf{i}}$ is thus $\langle \cos(\theta), \sin(\theta) \rangle$. We have $|e^{\theta \mathbf{i}}| = 1$, and in $\mathbb{R}^2$, $e^{\theta \mathbf{i}}$ traces the unit circle as θ goes from 0 to 2π .

In particular, $e^{\pi \mathbf{i}} = -\mathbf{1}$!

The underlying matrix for $e^{\theta \mathbf{i}} \in \mathcal{C}$ is a 2×2 rotation matrix, and $(a\mathbf{1} + b\mathbf{i})e^{\theta \mathbf{i}}$ rotates $\langle a, b \rangle$ counter-clockwise by an angle θ . You can see this in the action of the rotation matrix on the columns of $a\mathbf{1} + b\mathbf{i}$, each of which is similarly rotated in the product. It follows that any $z \in \mathcal{C}$ can be written in *polar form* as

$$z = re^{\theta \mathbf{i}}$$

for some $r > 0$ and $\theta \in [0, 2\pi)$ with $r = |z|$. Then, $re^{\theta \mathbf{i}} se^{\alpha \mathbf{i}} = rse^{(\theta+\alpha)\mathbf{i}}$.

The space $\mathcal{C}$, and its isomorphic version in $\mathbb{R}^2$, is known as the field of **complex numbers**.

Puzzle 153.

Find the matrix representation of the following linear functions $\mathbb{R}^2 \rightarrow \mathbb{R}^2$:

1. $L_1(a\mathbf{e}^1 + b\mathbf{e}^2) = (a + 2b)\mathbf{e}^1 + \frac{1}{2}b\mathbf{e}^2$.
2. $L_2(a\mathbf{e}^1 + b\mathbf{e}^2) = 4a\mathbf{e}^1 - 3b\mathbf{e}^2$.
3. $L_3(a\mathbf{e}^1 + b\mathbf{e}^2) = b\mathbf{e}^1 + a\mathbf{e}^2$.

$$4. L_4(ae^1 + be^2) = -be^1 + ae^2.$$

Example 17.39 The Singular Value Decomposition

Every linear function $\mathbb{R}^n \rightarrow \mathbb{R}^m$ can be represented by a diagonal matrix *if we choose the right bases* for $\mathbb{R}^n$ and $\mathbb{R}^m$. This idea is expressed by what is called the **singular value decomposition**. If $L: \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a linear function, then we can find *orthonormal bases* for $\mathbb{R}^m$ with vectors $\mathbf{u}^1, \dots, \mathbf{u}^m$ and for $\mathbb{R}^n$ with vectors with vectors $\mathbf{v}^1, \dots, \mathbf{v}^n$ and dual basis covectors $\mathbf{v}_1, \dots, \mathbf{v}_n$ where for $r = \min(m, n)$,

$$L = \sum_{k=1}^r \lambda_k \mathbf{u}^k \mathbf{v}_k, \quad (17.96)$$

with all $\lambda_k \geq 0$. In particular, if $x = \sum_{j=1}^n x_j \mathbf{v}^j$, then $L(x) = \sum_{k=1}^r \lambda_k x_k \mathbf{u}^k$. This is called the **singular value decomposition** of L . It says that with properly chosen orthonormal bases for L 's domain and codomain, L is a simple function that just stretches or shrinks the components of its input, i.e., L applied to $\langle x_1, \dots, x_n \rangle$ to $\langle \lambda_1 x_1, \dots, \lambda_r x_r, 0, \dots, 0 \rangle$ with $m - r$ 0's at the end.

Let A be the matrix representing L in the standard bases. Using equation (17.90) in Puzzle 150, we can express A in terms of the $\mathbf{u}$ and $\mathbf{v}$ bases. Let the $m \times m$ matrix U and $n \times n$ matrix V be defined by

$$U_i^k = \mathbf{e}^i \cdot \mathbf{u}^k \quad (17.97)$$

$$V_\ell^j = \mathbf{e}^\ell \cdot \mathbf{v}^j. \quad (17.98)$$

Then U 's columns are the components (in the standard basis) of the $\mathbf{u}$ vectors, and V 's columns are the components (in the standard basis) of the $\mathbf{v}$ vectors. These matrices satisfy $UU^\top = I_m = U^\top U$ and $VV^\top = I_n = V^\top V$; their transposes are their inverses.

Equations (17.90) and (17.96) then tell us that there is an $m \times n$ matrix D

with

$$D = \begin{cases} \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \lambda_r \end{bmatrix} & \text{if } m = n \\ \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \lambda_r \\ 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix} & \text{if } m > n \\ \begin{bmatrix} \lambda_1 & 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & 0 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & 0 & \ddots & \dots & 0 \\ 0 & 0 & 0 & \lambda_r & 0 & \dots & 0 \end{bmatrix} & \text{if } m < n, \end{cases}$$

so that

$$D = U^T A V \quad (17.99)$$

or equivalently,

$$A = U D V^T. \quad (17.100)$$

Thus, any matrix can be “factored” in this way, revealing a diagonal matrix in the right coordinate system.

To understand this more clearly, let’s consider a concrete example when $n = 2 = m$. Suppose that in the standard basis, our linear function is represented by the matrix

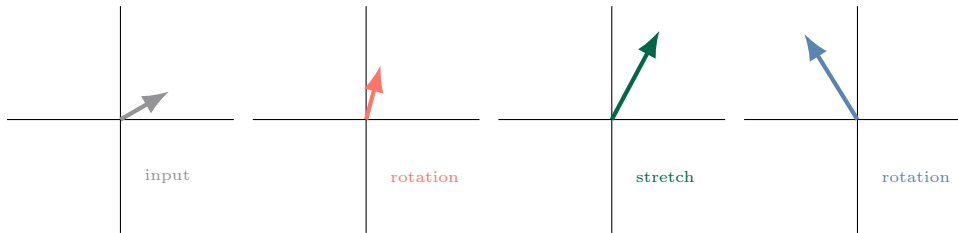
$$A = \begin{bmatrix} \frac{2}{\sqrt{3}} - 1 & -\frac{2}{\sqrt{3}} - 1 \\ \frac{1}{\sqrt{3}} + 2 & \frac{1}{\sqrt{3}} - 2 \end{bmatrix}$$

Computing the terms in (17.100) above (via a method not discussed here), we

have

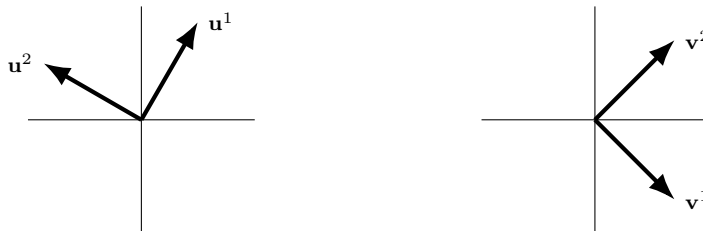
$$A = \begin{bmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \frac{4\sqrt{2}}{\sqrt{3}} & 0 \\ 0 & \frac{2\sqrt{2}}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

The first term U represents a counter-clockwise rotation by 60 degrees; the second term D represents a component-wise stretching; and the last term V^T represents a counter-clockwise rotation by 45 degrees. So the action of multiplying by A is to first rotate by 45 degrees, stretching the two components by $4\sqrt{2}/3$ and $2\sqrt{2}/3$, and then rotating the result by 60 degrees.



The vectors in the above picture go from input, through the **first rotation**, the **componentwise stretch/shrink**, and the **final rotation**, yielding the output vector.

The U and V matrices also tell us the input and output bases, with $\mathbf{u}^1 = \langle \frac{1}{2}, \frac{\sqrt{3}}{2} \rangle$, $\mathbf{u}^2 = \langle -\frac{\sqrt{3}}{2}, -\frac{1}{2} \rangle$ and $\mathbf{v}^1 = \langle \frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}} \rangle$, $\mathbf{v}^2 = \langle \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \rangle$. These are shown below with $\mathbf{u}$ on the left:



We also have $\lambda_1 = 4\sqrt{2}/\sqrt{3}$ and $\lambda_2 = 2\sqrt{2}/\sqrt{3}$ in (17.96).

Square matrices, like U and V , whose transpose is their inverse are called **orthogonal matrices**. All two-dimensional orthogonal matrices are rotations composed with an optional reflection across some line. Hence, *any* A is a rotation/reflection followed by a stretch/shrink and another rotation/reflection.

Puzzle 154. As we have seen, we can think of the set of $n \times n$ real matrices $\mathcal{M}_n$ as a vector space, with entry-wise addition and scaling as the operations. Define

two functions

$$\mathbf{dmat}: \mathbb{R}^n \rightarrow \mathcal{M}_n \quad (17.101)$$

$$\mathbf{diag}: \mathcal{M}_n \rightarrow \mathbb{R}^n \quad (17.102)$$

where $\mathbf{dmat}(v)$ is the diagonal matrix with $v_1, v_2, \dots, v_n$ for its diagonal entries and 0 for all other entries and $\mathbf{diag}(M)$ is the vector $\langle M_1^1, M_2^2, \dots, M_n^n \rangle$.

Convince yourself that (i) $\mathbf{dmat}$ and $\mathbf{diag}$ are *linear functions* and that (ii) $\mathbf{dmat}$ has $\mathbf{diag}$ as a post-inverse. Using the standard basis for $\mathbb{R}^n$ and a basis for $\mathcal{M}_n$ the matrices with a single 1 entry and the rest 0, find the matrix corresponding to $\mathbf{dmat}$.

The tricky thing about this is keeping straight what is what. We have two vector spaces, in one of which the “vectors” are matrices, and we are computing a matrix for a linear function between these vector spaces. How many entries/rows/columns must the matrix representing $\mathbf{dmat}$ have?

Example 17.40 Derivatives and the Chain Rule

Let $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$. Equation (14.1) in Section 14.2 defines the derivative of f at a point $x_0 \in \mathbb{R}^n$ as a *local linear approximation* to $(f(\blacksquare) - f(x_0))$ near x_0 . We are in a position now to appreciate more fully what that means.

Specifically, $f'(x_0)$ is defined so that

$$\frac{|f(x) - (f(x_0) + f'(x_0)(x - x_0))|}{|x - x_0|} \rightarrow 0$$

as $x \rightarrow x_0$. Notice that $x - x_0 \in \mathbb{R}^n$ and $f(x_0), f(x) \in \mathbb{R}^m$. Hence, the *linear function* $f'(x_0)$ maps $\mathbb{R}^n$ to $\mathbb{R}^m$. It is $f'(x_0)$, *not* f' , that is the linear function. For each x_0 , the derivative gives us a linear function $f'(x_0): \mathbb{R}^n \rightarrow \mathbb{R}^m$ that approximates f for points *near* x_0 .

We can represent this linear function as a matrix relative to the standard bases for $\mathbb{R}^n$ and $\mathbb{R}^m$. Write $f = \langle f_1, \dots, f_m \rangle$ in terms of its component functions and $x = \langle x_1, \dots, x_n \rangle$ in terms its components. Then, writing $f'(x_0)$ as a matrix we have

$$f'(x_0)_i^j = \frac{\partial f_i}{\partial x_j}(x_0), \quad (17.103)$$

where $\frac{\partial}{\partial x_j}$ is the *partial derivative*, meaning that we take derivatives varying x_j only, leaving the other components fixed. For example:

- When f is linear represented by matrix A in standard bases, then $f(x) = Ax$ and $f'(x_0)_i^j = A_i^j$; that is, $f'(x_0) = A$.
- If $f: \mathbb{R} \rightarrow \mathbb{R}^3$ with $f(t) = \langle \cos(2\pi t), \sin(2\pi t), \sin(\pi t/10) \rangle$, then the image of f traces a curve on the unit sphere 5 times as the curve moves from the equator to a pole (or vice versa).

$$f'(t_0) = \begin{bmatrix} -2\pi \sin(2\pi t_0) \\ 2\pi \cos(2\pi t_0) \\ \frac{2\pi}{20} \cos(\pi t_0/10) \end{bmatrix},$$

which is a 3×1 matrix (i.e., a 3-vector) as expected. Then $f'(t_0)(t - t_0)$ is a 3-vector approximating the three vector $f(x) - f(x_0)$.

- If $f: \mathbb{R}^2 \rightarrow \mathbb{R}$ with $f(x, y) = x^2 + y^2$, the squared distance of $\langle x, y \rangle$ from the origin, then

$$f'(x_0, y_0) = \begin{bmatrix} 2x_0 & 2y_0 \end{bmatrix},$$

which is a 1×3 matrix (i.e., a 2-covector). Then $f'(x_0, y_0) \begin{bmatrix} x-x_0 \\ y-y_0 \end{bmatrix} = 2x_0(x-x_0) + 2y_0(y-y_0)$ is a number approximating the number $f(x, y) - f(x_0, y_0)$.

In general, if $f: \mathbb{R}^n \rightarrow \mathbb{R}$, the derivative $f'(x_0)$ is an n -dimensional covector called the **gradient** of f , traditionally denoted $\nabla f(x_0)$. It's value at a vector v is the derivative of the function $\langle t \rangle \mapsto f(x_0 + tv)$ from $\mathbb{R} \rightarrow \mathbb{R}$, i.e., the derivative of f at x_0 in the *direction* v . The vector $\nabla f(x_0)^T$ gives the direction of *steepest ascent*.

Remember that we can write functions of n -tuples as functions of n arguments without changing the meaning.

In summary: the derivative f' maps a point $x_0 \in \mathbb{R}^n$ to a linear function $\mathbb{R}^n \rightarrow \mathbb{R}^m$ that approximates $(f(\blacksquare) - f(x_0))$ near x_0 .

From this perspective, the *chain rule* takes on new meaning as the *composition of those linear functions*. Specifically, if $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ and $g: \mathbb{R}^m \rightarrow \mathbb{R}^p$, then $g \circ f: \mathbb{R}^n \rightarrow \mathbb{R}^p$ and

$$(g \circ f)'(x_0) = g'(f(x_0))f'(x_0). \quad (17.104)$$

The two terms on the right-hand side of (17.104) are linear functions. Write $y_0 = f(x_0)$; then, $g'(f(x_0)) = g'(y_0)$ approximates $(g(\blacksquare) - g(y_0))$ near y_0 , and $f'(x_0)$ approximates $(f(\blacksquare) - f(x_0))$ near x_0 . Their juxtaposition represents the composition $g'(y_0) \circ f'(x_0)$. So to approximate $(g \circ f)$ near x_0 , we first

approximate f near x_0 and then g near $f(x_0)$.

The chain rule extends over any number of steps. Start with a chain of functions $f_1, f_2, \dots, f_K$ that we can compose, meaning that successive domains and codomains are compatible $\mathbb{R}^{p_k}$ for $k \in [1 \dots K + 1]$.

$$\begin{array}{ccccccc}
 y_1 & \xrightarrow{f_1} & y_2 & \xrightarrow{f_2} & \cdots & \xrightarrow{\quad} & y_{K-1} & \xrightarrow{f_{K-1}} & y_K & \xrightarrow{f_K} & y_{K+1} \\
 \\
 \mathbb{R}^{p_1} & \xrightarrow{f'_1(y_1)} & \mathbb{R}^{p_2} & \xrightarrow{f'_2(y_2)} & \cdots & \xrightarrow{\quad} & \mathbb{R}^{p_{K-1}} & \xrightarrow{f'_{K-1}(y_{K-1})} & \mathbb{R}^{p_K} & \xrightarrow{f'_K(y_K)} & \mathbb{R}^{p_{K+1}} \\
 & & & & & \searrow & & \nearrow & & & \\
 & & & & & & (f_1 \circ f_2 \circ \cdots \circ f_K)'(y_0) & & & &
 \end{array}$$

A point $y_1 \in \mathbb{R}^{p_1}$ maps successively to values in these sets, culminating in the value $y_{K+1} = (f_K \circ f_{K-1} \circ \cdots \circ f_1)(y_1) = (f_1 \circ f_2 \circ \cdots \circ f_K)(y_1)$. The derivative of this composite function is the composition of the derivatives for each step at the corresponding value, as seen in the diagram. That is:

$$(f_1 \circ f_2 \circ \cdots \circ f_K)'(y_0) = f'_1(y_1) \circ f'_2(y_2) \circ \cdots \circ f'_K(y_K) \quad (17.105)$$

or equivalently,

$$(f_K \circ f_{K-1} \circ \cdots \circ f_1)'(y_0) = f'_K(y_K) \circ f'_{K-1}(y_{K-1}) \circ \cdots \circ f'_2(y_2) \circ f'_1(y_1). \quad (17.106)$$

Loosely speaking, the chain rule tells us that the derivative of a composition is the composition of the derivatives.

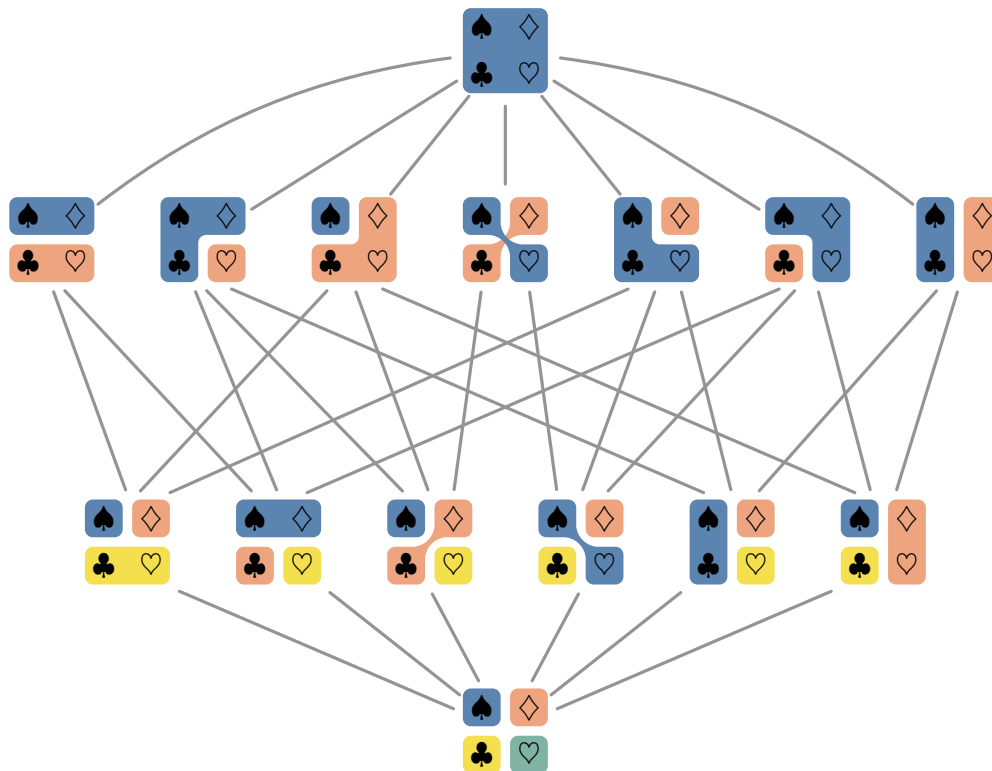


FIGURE 17.12. All partitions of the set $\{\spadesuit, \diamondsuit, \clubsuit, \heartsuit\}$ arranged in order from finest to coarsest. Parts are specified by connected regions with a shared (but arbitrary) color.

17.4 Orders

Figure 17.12 shows all partitions²⁴ of the set $\{\spadesuit, \diamondsuit, \clubsuit, \heartsuit\}$ arranged in a particular way. At the top is the partition with a single part containing all four elements; at the bottom is the partition with four parts, one per element. A line connecting a partition to a partition above it means that the parts of the second partition are unions of parts of the first. We say that the first (second) partition is *finer* (*coarser*) than the second (first) partition. This diagram gives structure to the set of partitions, a graded and hierarchical organization. Each row corresponds to partitions with the same number of parts; each chain of partitions from bottom to top connected by lines are arranged in order from finest to coarsest. This structure is induced by an **order relation** on the set. Given any two partitions p, q of $\{\spadesuit, \diamondsuit, \clubsuit, \heartsuit\}$, we say that partition $p \preceq q$ if p is finer than q .

Order relations, or orders for short, are a special case of the relations we considered in Chapter 16. The numeric relations $\leq, \geq, <, >$ are familiar and motivating examples. These divide into pairs in two ways. First, we can obtain $\geq$ from $\leq$ and $>$ from $<$ by using the opposite order of arguments; for instance, $x \leq y$ is equivalent to $y \geq x$. Second, we can obtain $<$ from $\leq$ and $>$ from $\geq$ by precluding equality; see say that $<$ and $>$ are *strict* in this sense.²⁵ These numeric relations have stronger properties than we always need for a useful order relation. For instance, every pair of numbers can be compared with $\leq$, in contrast to the relation we used on the partitions of $\{\spadesuit, \diamondsuit, \clubsuit, \heartsuit\}$ where some partitions are not directly comparable (like $\{\{\spadesuit, \diamondsuit\}, \{\heartsuit, \clubsuit\}\}$ and $\{\{\spadesuit, \clubsuit\}, \{\heartsuit, \diamondsuit\}\}$, neither is coarser than the other). Similarly, $x \leq y$ and $y \leq x$ implies that $x = y$ (anti-symmetry), but we will see reasonable orders that do not have this property.

In this Section, we will describe a small taxonomy of order relations along with many examples to illustrate the diverse ways in which orders give structure to sets. In these terms, $\leq$ and $\geq$ are *total orders*, and $<$ and $>$ are *strict total orders*. We will start, however, with a more general and broadly applicable order, called a *preorder*, and build from there. A set equipped with an order relation is called an *ordered set*, often qualified by the type of order. An order is a type of structure on a set, and as with other types of structures we have considered, it is useful to study functions that preserve that structure. We have seen these before in a narrower context; they are called *monotone functions*.

²⁴Partitions of a set are described in Examples 14.1 and 16.3.

²⁵ $<$ and $>$ are irreflexive and asymmetric; $\leq$ and $\geq$ are reflexive and anti-symmetric. See page 682.

PREORDERS. Starting with a relation like $\leq$, let's strip it down to its essential requirements for being considered an order. We have $x \leq x$ for every x , so we will require reflexivity. If $x \leq y$ and $y \leq z$, then any notion of consistent ordering requires $x \leq z$ as well. That is, we require transitivity. It turns out that those are the minimal requirements, and we call a homogeneous relation on a set that is reflexive and transitive a *preorder*. The name *preorder* seems a bit odd, as we consider this an order relation; it comes from the fact that we get more stringent order relations like $\leq$ by adding requirements on a preorder. We will use $\preceq$ as a generic symbol for preorders, but in practice, we can denote a preorder in any convenient way.

A homogeneous binary relation $\preceq$ on a set $\mathcal{A}$ that is *reflexive* and *transitive* is called a **preorder**. That is,

1. $a \preceq a$ for all $a \in \mathcal{A}$, and
2. $a \preceq b$ and $b \preceq c$ implies $a \preceq c$.

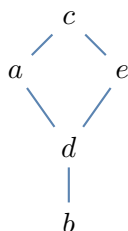
If either $a \preceq b$ or $b \preceq a$, then $a, b \in \mathcal{A}$ are said to be **comparable**.

A key aspect of this definition is that in a preordered set, two elements *need not be comparable*, as illustrated with the partitions described earlier. As an extreme example of this, we can define a preorder on any set $\mathcal{A}$ by $\{\langle a, a \rangle \mid a \in \mathcal{A}\}$. This relation is called the *discrete preorder* on $\mathcal{A}$. It is reflexive by definition and *trivially* transitive as elements are comparable only with themselves. Dually, the *codiscrete preorder* is $\mathcal{A} \times \mathcal{A}$, relating every pair $a, b \in \mathcal{A}$. It is trivially reflexive and transitive.

Of slightly more interest, we can define a preorder on the set of Booleans, $\mathbb{B} = \{\perp, \top\}$, by $\{\langle \perp, \perp \rangle, \langle \perp, \top \rangle, \langle \top, \top \rangle\}$, i.e., the reflexive relation with $\perp \preceq \top$. Transitivity follows because the set has only two elements.

For real numbers (or integers or natural numbers), $\leq$ and $\geq$ are both preorders. They satisfy stronger properties as well, but they are both reflexive and transitive.

We can build a preorder $\bullet \rightarrow$ on any finite set $\mathcal{A}$ in three steps: (i) declare $a \bullet \rightarrow a$ for all $a \in \mathcal{A}$, (ii) define $a \bullet \rightarrow b$ for any pairs of non-equal elements that we like, and (iii) set $a \bullet \rightarrow c$ for any pairs of elements required to make the transitivity property true. Step (i) ensures reflexivity. The last step, called the *transitive closure* or the relation through step (ii), ensures transitivity. For example, apply step (ii) to the set $\{a, b, c, d, e\}$ with an edge in the following graph



represents $x \bullet \circ y$ with x the *lower* node in the graph. This yields a preorder where $x \bullet \circ z$ whenever there is a *path* in the graph from x a node at or above x 's level in the picture, including of course the empty path from x that stays at x . The same process works with the graph in Figure 17.12, producing the order on partitions we saw earlier.

The next example shows that a preorder need not reflect a comparison of “size”.

Example 17.41 Integer Divisibility. Recall that $k \backslash n$ (read “ k divides n ”, note the lean in bar) means that $n = mk$ for some integer m . This gives a preorder on the integers.

We have that $n \backslash n$ as $n = 1n$. If $k \backslash m$ and $m \backslash n$, then $m = pk$ and $n = qm$ for some integers p and q . Thus, $n = (pq)k$ and so $k \backslash n$. The relation $\backslash$ is thus reflexive and transitive.

Notice that $n \backslash 0$ for every non-zero n , because $0 = 0n$, but zero divides no other integer. Conversely, $1 \backslash n$ for every integer n , but only 1 and -1 divide 1.

If we restrict our attention to the natural numbers $[0..)$, then the preorder $\backslash$ has a “least” element 1, because $1 \backslash n$ for all $n \in [0..)$, and a “greatest” element 0, because $n \backslash 0$ for all $n \in [0..)$.

Given one preorder, we actually get another one for free! Specifically, we can form the **opposite preorder** that simply reverses the order: $a \preccurlyeq_{op} b$ if and only if $b \preccurlyeq a$. (When the symbol for a preorder has an obvious direction, we often denote the opposite preorder mirroring it, like $\succcurlyeq$ for $\preccurlyeq_{op}$ or $\circ \bullet$ for $\bullet \circ_{op}$.) This mimics the relationship between $\leq$ and $\geq$. Reflexivity follows immediately because $a \preccurlyeq a$ and $a \succcurlyeq a$ mean the same thing. Transitivity follows because $a \succcurlyeq b$ and $b \succcurlyeq c$ if and only if $c \preccurlyeq b$ and $b \preccurlyeq a$, giving $c \preccurlyeq a$ and thus $a \succcurlyeq c$. For instance, for our partition preorder above, $p \succcurlyeq q$ when p is *coarser* than q , which is equivalent to reflecting Figure 17.12 vertically.

Example 17.42. We have a collection of people, and a list with the full names of the people in the collection written in alphabetical order. If p, q are two people, define $p \preceq q$ if p 's name appears on the list no later than q 's name.

Since everyone's name appears no later than their own name, $p \preceq p$. If $p \preceq q$ and $q \preceq r$, then p 's name appears no later than q 's and which appears no later than r 's, so $p \preceq r$. It follows that this $\preceq$ is a preorder on the set of people in our collection.

If there two distinct people $p \neq p'$ in the collection with the *same name*, we have $p \preceq p'$ and $p' \preceq p$ as their common name first occurs at the same point in the list. This shows that a preorder need not be anti-symmetric; we can have $p \preceq p'$ and $p' \preceq p$ but $p \neq p'$.

Also, the opposite preorder here corresponds to sorting the list of names in reverse alphabetical order.

As the previous example shows, we can have $a \preceq b$ and $b \preceq a$ while $a \neq b$. In practice, even when such an a and b are not equal, they tend to have a property in common that makes them “similar” or “congruent” in some way. For instance, in the previous example, such a pair are two distinct people with the *same name*. Given a preorder $\preceq$ on a set $\mathcal{A}$, we can define a relation $\cong$ on $\mathcal{A}$ by

$$a \cong b \quad \text{if and only if} \quad a \preceq b \wedge b \preceq a. \quad (17.107)$$

Because $\preceq$ and its opposite are preorders, $\cong$ is reflexive and transitive. It is symmetric by definition. In Chapter 16, we called a reflexive, transitive, and symmetric relation an *equivalence relation*.

Example 17.43 Subsets

If $\mathcal{A}$ is a set, its power set $2^{\mathcal{A}}$ is the set of all its subsets. $\langle 2^{\mathcal{A}}, \subseteq \rangle$ is a preordered set:

1. $\mathcal{S} \subseteq \mathcal{S}$
2. $\mathcal{S} \subseteq \mathcal{T}$ and $\mathcal{T} \subseteq \mathcal{U}$ implies $\mathcal{S} \subseteq \mathcal{U}$.

Here, the empty set $\{\}$ is the “least” element in this preorder because $\{\} \subseteq \mathcal{S}$ for all $\mathcal{S}$, and $\mathcal{A}$ is the “greatest” element in this preorder because $\mathcal{S} \subseteq \mathcal{A}$ for all $\mathcal{S}$.

Now let $\mathcal{A} = \mathbb{N}$, the set of natural numbers. Define a preorder $\sqsubseteq$ on $2^{\mathbb{N}}$ where $\mathcal{S} \sqsubseteq \mathcal{T}$ when $\mathcal{S} = \mathcal{S}' \cup \mathcal{F}$ where $\mathcal{S}' \subseteq \mathcal{T}$ and $\mathcal{F}$ is *finite*. Then

1. $\mathcal{S} \sqsubseteq \mathcal{S}$ with $\mathcal{F} = \{\}$, and

2. $\mathcal{S} \subseteq \mathcal{T}$ and $\mathcal{T} \subseteq \mathcal{U}$ implies that $\mathcal{S}$ is a subset of $\mathcal{U}$ except for at most finitely many elements that might belong to the part of $\mathcal{T}$ not in $\mathcal{U}$.

Hence, $\subseteq$ is a preorder.

Using the equivalence relation described above, we see that $\mathcal{S} \cong \mathcal{T}$ when $\mathcal{S}$ and $\mathcal{T}$ differ only by a finite set of elements. That is, $\mathcal{S} = \mathcal{U} \cup \mathcal{F}$ and $\mathcal{T} = \mathcal{U} \cup \mathcal{F}'$ where $\mathcal{F}$ and $\mathcal{F}'$ are finite. In particular, any two finite subsets are equivalent in this way.

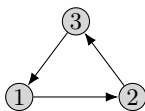
Example 17.44 Reachability in Directed Graphs

Let $G = \langle \mathcal{N}, \mathcal{E}, \varphi \rangle$ be a directed graph, as described in Example 10.18. A *path* in a directed graph is a sequence of distinct nodes $n_1, \dots, n_k$ and edges $e_1, \dots, e_{k-1}$ such that $\varphi(e_i) = \langle n_i, n_{i+1} \rangle$. Node n_1 is the start and node n_k is the end; if $k = 1$, the path starts and ends at n_1 with no edges. Loosely speaking, a path from node n to node n' is a way to move from n to n' by following directed edges in the graph and visiting each node at most once.

Define a preorder $\rightarrow$ on $\mathcal{N}$ with $n \rightarrow n'$ when there is a path from node n to node n' .

This is reflexive because we can always find a trivial path that starts and ends at node n . If there is a path from n to n' and a path from n' to n'' , we can find a path from n to n'' by joining the paths together. (If the paths overlap, we can cut out all parts of the second path between overlapping nodes.) Hence, $\rightarrow$ is transitive.

It is possible for $n \neq n'$ with $n \rightarrow n'$ and $n' \rightarrow n$. How? This means there is a path from n to n' and (because this is a digraph) a different path from n' to n . In other words, it is possible to go in a *cycle* from n to n' and back. Consider the simple graph below. We have $1 \rightarrow 3$ and $3 \rightarrow 1$ because the two nodes are connected in a *cycle*.



If we define $\cong$ analogously to (17.107) for this preorder, then equivalence classes contain all pairs of nodes that share a cycle. In an *acyclic* digraph, it follows that $\rightarrow$ is anti-symmetric.

The following examples further illustrate the diverse contexts in which preorders

arise.

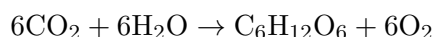
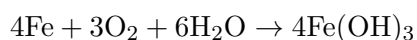
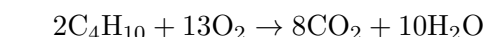
Example 17.45 Parts and Wholes. Given a collection of nouns, we can define a relation $\sqsubset$ that defines a “part-of” relationship. A finger is part of a hand, a leaf is part of a tree. The word for the part is a *meronym* and the word for the whole is a *holonym*, and we define $\text{meronym} \sqsubset \text{holonym}$. For example, a **finger** $\sqsubset$ **hand** and **leaf** $\sqsubset$ **tree**.

This relation is reflexive because everything is a part of itself, the whole part. By logic, it is also transitive. So $\sqsubset$ is a preorder. Not every pair of words is comparable; for instance, **finger** and **tree**. And some words can relate to other words in more than one sense through different meanings.

Example 17.46 Chemical Reactions

We start with a collection substances – atoms and molecules – like O_2 , NaCl , NaSO_4 . Then, we allow combinations of these substances by allowing positive integer multiples of the basic substances to be combined, e.g., $6\text{CO}_2 + 6\text{H}_2\text{O}$ or $2\text{C}_4\text{H}_{10} + 13\text{O}_2$. The result is a set of *chemicals*, $\mathcal{C}$.

Define a preorder $\rightarrow$ on $\mathcal{C}$ where $a \rightarrow b$ means that chemical a can produce chemical b , under some conditions and possibly with the addition of heat or light or such. For instance,



describe the possibility of butane combustion, iron rusting, and photosynthesis. Here, a reversible reaction means that two chemicals can be equivalent in this preorder but not equal.

Example 17.47 Task Prerequisites. You have a set of tasks on your TODO list. For two tasks t_1, t_2 on the list, we define $t_1 \preceq t_2$ if t_1 needs to be completed in order to complete t_2 ; that is, t_1 is a prerequisite of t_2 .

This is reflexive because we need to complete a task in order to complete it. (Tautologically.) It is transitive because if we need to complete t_1 in order to complete t_2 and to complete t_2 in order to complete t_3 , then t_1 is also a

prerequisite of t_3 .

Example 17.48 Truth and Consequences. In a collection of logical statements, we say that a statement s *entails* a statement t , if the truth of t follows as a consequence of the truth of s . Write this relation as $s \models t$.

Surprise, surprise: this is a preorder on our collection of statements. Every statement entails itself, and transitivity follows as well: if s entails t and t entails u , we may deduce the truth of u from the truth of s .

It is possible for $s \models t$ and $t \models s$ but $s \neq t$ with two equivalent but distinct statements. For example: “the integer n is even” and “one plus the integer n is odd”.

Example 17.49 Graph Minors Let G and H be undirected graphs. (See Example 10.19.) We say that H is a *minor* of G , $H \preceq G$, if H can be obtained from G by

1. deleting nodes,
2. deleting edges, and
3. contracting edges,

where the latter means removing an edge between two nodes and merging the two incident nodes into one.

Every graph is a minor of itself by simply not deleting or contracting anything. Transitivity follows by doing one set of deletions/contractions followed by the other. So the minor relation $\preceq$ is a preorder.

Puzzle 155. Is the graph minor relation in the previous example antisymmetric? Construct two distinct undirected graphs where one is a minor of the other.

Example 17.50 Asymptotic Order

In the analysis of algorithm, we often represent the computational complexity of an algorithm by a function $f: \mathbb{N} \rightarrow \mathbb{R}$, where $f(n)$ gives the computational effort (e.g., time, memory) required by the algorithm when the input is of “size” n .

While we would like to compute this function exactly for various algorithms in order to compare them, in practice that is often difficult, and we settle for comparing algorithms by their *asymptotic* computational complexity, i.e., the behavior of $f(n)$ as n approaches ∞ .

Define a preorder on $\mathbb{N} \rightarrow \mathbb{R}$ by saying that f is *eventually bounded by* g , $f \leq_{ev} g$, if there is an $N \in \mathbb{N}$ such that $f(n) \leq g(n)$ whenever $n \geq N$.

A related preorder is a bit less stringent: define $f \leq_O g$ if there is an $M > 0$ and an $N \in [0..)$ such that $f(n) \leq Mg(n)$ for all $n \geq N$. When $|f| \leq_O |g|$, we say that f is of *asymptotic order* g . It is traditional to write $f = O(g)$ and say that f is “big Oh” of g .

Example 17.51 Computational Oracles

In the study of computation, we say that problem A reduces to problem B when we can write a solution for A in terms of a solution for B . Then we can solve an instance of problem A by first translating it to an instance of problem B , applying our solution to B , and then translating that solution to a solution to A . We can write $A \leq B$ to indicate that A reduces to B . This is a preorder.

There are various types of reductions. For instance, we often want to insist that the translations of instances and solutions can be done in a polynomial number of steps in the size of the input. In this case, we specify the preorder as $\leq_P$. Or we might consider a *Turing reduction* where we solve an instance of problem A on a machine that offers the solution to instances of problem B as a subroutine or “oracle” that we can call. In this case, we might write the preorder as $\leq_T$.

For whatever type, saying that A reduces to B tells us that B is **at least as hard as** A . Notice that we can have $A \leq B$ and $B \leq A$, i.e., each problem reduces to the other, with $A \neq B$. The equivalence classes for $\cong$ in this case give us sets of problems of equivalent computational complexity.

Example 17.52 Subtypes in Programming

Many programming languages require that the programmer declare a data type for each variable. In some contexts, we can declare a type S that is a *subtype* of type T . This means that any value of type S can be passed to any computation that expects a value of type T or returned from any computation that is declared to return a value of type T . A value of type S “is a” value of type T .

The subtype relation is a preorder.

Example 17.53. We have seen several kinds of structures (e.g., graphs, monoids, semirings) on sets and functions between sets equipped with those structures that *preserve* the structure, homomorphisms. If we have two sets $\mathcal{A}$ and $\mathcal{B}$ equipped with the same type of structure, we can define a preorder $\mathcal{A} \vdash \mathcal{B}$ to mean that there is a homomorphism (of that type) from $\mathcal{A}$ to $\mathcal{B}$. The identity shows that this is reflexive, and composition shows that it is transitive.

Example 17.54. Define a preorder on FRPs in the following way. If X and Y are FRPs where there is a compatible statistic so that $Y = \psi(X)$, then write $X \multimap Y$. As in the previous example, the identity statistic and the composition of statistics shows that this is a reflexive and transitive relation, and thus a preorder.

Based on Part I

If $X \multimap Y$ and $Y \multimap X$, then we can convert values of X to values of Y and vice versa. In this sense, X and Y give equivalent information.

Puzzle 156. Consider a collection of machines or automata that take some input and produce some output, and each of which can be configured by one or more parameter settings. Machine M_2 to *simulate* machine M_1 if we can configure M_2 to produce the same output as M_1 for each input that M_1 accepts.

Can you use this idea to construct a preorder on machines?

Example 17.55 Lifting a Preorder

If $\langle \mathcal{X}, \preceq \rangle$ is a preordered set, then for any set $\mathcal{A}$, we can put a preorder on the set $\mathcal{A} \rightarrow \mathcal{X}$ of functions from $\mathcal{A}$ to $\mathcal{X}$. For simplicity, we'll use the same symbol for both relations, which perhaps surprisingly, will not be confusing.

If $f, g: \mathcal{A} \rightarrow \mathcal{X}$, define $f \preceq g$ when $f(a) \preceq g(a)$ for all $a \in \mathcal{A}$. Two functions are ordered by $\preceq$ if all their returned values are similarly ordered. This is exactly how we lift order relations to numeric functions, for instance, when we say $f \geq \text{const}_0$ (often shortened to $f \geq 0$).

You should confirm that this is a reflexive and transitive relation.

Example 17.56 The Product Preorder

If $\langle \mathcal{A}_1, \preceq_1 \rangle, \langle \mathcal{A}_2, \preceq_2 \rangle, \dots, \langle \mathcal{A}_n, \preceq_n \rangle$ are preordered sets, then we can define a compatible preorder on the Cartesian product $\mathcal{A}_1 \times \mathcal{A}_2 \times \dots \times \mathcal{A}_n$, the set of

tuples with components from these sets.

Define $\langle a_1, a_2, \dots, a_n \rangle \preceq \langle a'_1, a'_2, \dots, a'_n \rangle$ if and only if $a_1 \preceq_1 a'_1$, $a_2 \preceq_2 a'_2$, $\dots$, and $a_n \preceq_n a'_n$. This is reflexive and transitive because each of the component preorders is.

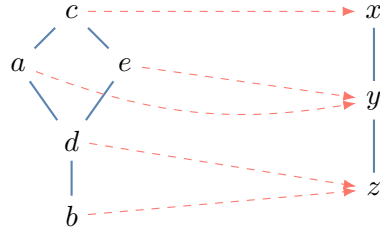
MONOTONE FUNCTIONS. We have seen several types of structure and with each of them structure-preserving functions (homomorphisms); orders are no different.

A **preorder homomorphism**, commonly called a *monotone function*, is a function $f: \langle \mathcal{A}, \preceq_{\mathcal{A}} \rangle \rightarrow \langle \mathcal{B}, \preceq_{\mathcal{B}} \rangle$ between preordered sets such that for $a, a' \in \mathcal{A}$

$$a \preceq_{\mathcal{A}} a' \text{ implies } f(a) \preceq_{\mathcal{B}} f(a'). \quad (17.108)$$

For example, the function from $\{a, b, c, d, e\}$ to $\{x, y, z\}$ (represented by dashed arrows) with the preorders determined by²⁶ the following graphs is monotone. That the dashed arrows do not cross reflects the monotonicity of the function.

²⁶See discussion of the example graph on page 780.



Example 17.57 Direct and Inverse Images

We saw in Example 17.43 that $\subseteq$ gives a preorder on $2^{\mathcal{A}}$ for a set $\mathcal{A}$. If $f: \mathcal{A} \rightarrow \mathcal{B}$, then we have functions $f^{\rightarrow}: 2^{\mathcal{A}} \rightarrow 2^{\mathcal{B}}$ and $f^{\leftarrow}: 2^{\mathcal{B}} \rightarrow 2^{\mathcal{A}}$. Both of the latter functions are monotone.

If $\mathcal{X} \subseteq \mathcal{Y} \subseteq \mathcal{A}$, then every element of $\mathcal{X}$ belongs to $\mathcal{Y}$, so $f(x) \in f^{\rightarrow}(\mathcal{Y})$ and hence $f^{\rightarrow}(\mathcal{X}) \subseteq f^{\rightarrow}(\mathcal{Y})$. If $\mathcal{S} \subseteq \mathcal{T} \subseteq \mathcal{B}$, then $f(x) \in \mathcal{S}$ implies $f(x) \in \mathcal{T}$. So, $f^{\leftarrow}(\mathcal{S}) \subseteq f^{\leftarrow}(\mathcal{T})$.

As we've seen, both $2^{\mathcal{A}}$ and $2^{\mathcal{B}}$ have $\{\}$ as their “least” element in the subset preorder, and $\mathcal{A}$ and $\mathcal{B}$ are the respective “greatest” element. And as monotonicity requires, both functions preserve the least element: $f^{\rightarrow}(\{\}) = \{\}$ and $f^{\leftarrow}(\{\}) = \{\}$. Only $f^{\leftarrow}$ preserves the greatest element: $f^{\leftarrow}(\mathcal{B}) = \mathcal{A}$.

Example 17.58 Partitions

Let $\text{Part}(\mathcal{A})$ be the set of partitions of a set $\mathcal{A}$. As we saw in Example 14.1 and earlier in this Section:

- every partition corresponds to a surjection $\mathcal{A} \twoheadrightarrow \mathcal{P}$ for some set $\mathcal{P}$ labeling the parts, and
- we can put a preorder $\preceq$ on $\text{Part}(\mathcal{A})$ where $p \preceq q$ when p is finer than q .

Given nonempty sets $\mathcal{A}$ and $\mathcal{B}$, let $f: \mathcal{A} \twoheadrightarrow \mathcal{B}$. This leads to a monotone function $f^*: \text{Part}(\mathcal{B}) \rightarrow \text{Part}(\mathcal{A})$.

To see this, start with a partition of $\mathcal{B}$ represented as a surjection $p: \mathcal{B} \twoheadrightarrow \mathcal{P}$ for some set $\mathcal{P}$. Then, $p \circ f: \mathcal{A} \twoheadrightarrow \mathcal{P}$ is a surjection and thus represents a partition of $\mathcal{A}$. This defines f^* .

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\ & \searrow f^*(p) & \downarrow p \\ & & \mathcal{P} \end{array}$$

If $p \preceq q$ are partitions of $\mathcal{B}$, we want to show that $f^*(p) \preceq f^*(q)$ among partitions of $\mathcal{A}$. Because $p \preceq q$, each part of p , $p^-(i)$, belongs to some part of q , $q^-(j)$. By the previous example, $f^-(p^-(i)) \subseteq f^-(q^-(j))$, and thus each part of $f^*(p)$ belongs to some part of $f^*(q)$. That is, $f^*(p) \preceq f^*(q)$.

Example 17.59 Upper and Lower Sets

If $\langle \mathcal{X}, \preceq \rangle$ is a preordered set, we can associate two sets with each $x \in \mathcal{X}$. The *upper set* at x is $\uparrow(x) = \{x' \in \mathcal{X} \mid x \leq x'\}$, the set of all elements at least as “big” (in the preorder) as x . The *lower set* at x is $\downarrow(x) = \{x' \in \mathcal{X} \mid x' \leq x\}$, the set of all elements no “bigger” (in the preorder) than x .

We can view $\uparrow$ and $\downarrow$ as *functions* on $\mathcal{X}$; specifically, they are monotone functions on preordered sets

$$\uparrow: \langle \mathcal{X}, \preceq_{op} \rangle \rightarrow \langle \mathcal{P}^{\mathcal{X}}, \subseteq \rangle \quad (17.109)$$

$$\downarrow: \langle \mathcal{X}, \preceq \rangle \rightarrow \langle \mathcal{P}^{\mathcal{X}}, \subseteq \rangle. \quad (17.110)$$

We use the *opposite* preorder in the first case because $x \preceq y$ implies that $\{x' \in \mathcal{X} \mid y \leq x'\} \subseteq \{x' \in \mathcal{X} \mid x \leq x'\}$. In contrast, $x \preceq y$ implies that $\{x' \in \mathcal{X} \mid x' \leq x\} \subseteq \{x' \in \mathcal{X} \mid x' \leq y\}$. But we can say more: if $\uparrow(y) \subseteq \uparrow(x)$, then $x \preceq y$, and if $\downarrow(x) \subseteq \downarrow(y)$, then $x \preceq y$. Taken together these indicate that $\uparrow(x)$ and $\downarrow(x)$ are in

a sense representative of x ; relationships among the upper sets (or alternatively among the lower sets) mimic the relationships among the elements, at least as far as the preorder is concerned.

UPPER AND LOWER BOUNDS. It would seem that a benefit of having an “order” is being able to find the “biggest” element in a collection. For two numbers x and y , we can compute $\max(x, y)$; put another way, if $x \leq y$, $\max(x, y) = y$. For a general set of numbers, things are more complicated: the maximum of a set of numbers might not exist within the set, for instance with $(0, 1)$ or $\mathbb{R}$. More care is needed with a general preorder.

So taking a less direct path, we start by thinking about *bounds*. If $\langle \mathcal{X}, \preceq \rangle$ is a preordered set and $\mathcal{A} \subseteq \mathcal{X}$, we say that

1. $u \in \mathcal{X}$ is an **upper bound** for $\mathcal{A}$ if $a \preceq u$ for all $a \in \mathcal{A}$, and
2. $\ell \in \mathcal{X}$ is a **lower bound** for $\mathcal{A}$ if $\ell \preceq a$ for all $a \in \mathcal{A}$.

We are interested in the greatest lower bound and the least upper bound of a set, if they exist.

A **meet** of $\mathcal{A}$ if it exists is a *greatest lower bound*, a value $\bar{\ell} \in \mathcal{X}$ such that for any lower bound ℓ for $\mathcal{A}$ we have $\ell \leq \bar{\ell}$. If a meet of $\mathcal{A}$ exists, we denote it $\bigwedge_{a \in \mathcal{A}} a$ or $\bigwedge \mathcal{A}$.

A **join** of $\mathcal{A}$ if it exists is a *least upper bound*, a value $\underline{u} \in \mathcal{X}$ such that for any upper bound u for $\mathcal{A}$ we have $\underline{u} \leq u$. If a join of $\mathcal{A}$ exists, we denote it $\bigvee_{a \in \mathcal{A}} a$ or $\bigvee \mathcal{A}$.

Over two elements, meet and join are written as infix operators: $a_1 \wedge a_2$ and $a_1 \vee a_2$ respectively; similarly, for any finite number of elements, $a_1 \wedge a_2 \wedge \cdots \wedge a_n$ and $a_1 \vee a_2 \vee \cdots \vee a_n$. If they exist, then

$$a_1 \leq a_1 \vee a_2 \tag{17.111}$$

$$a_2 \leq a_1 \vee a_2 \tag{17.112}$$

$$a_1 \wedge a_2 \leq a_1 \tag{17.113}$$

$$a_1 \wedge a_2 \leq a_2. \tag{17.114}$$

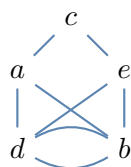
For real numbers x and y , $x \wedge y = \min(x, y)$ and $x \vee y = \max(x, y)$. For the Booleans $\langle \mathbb{B}, \leq \rangle$ with $\perp \leq \top$ every pair of elements has a meet and a join:

$\wedge$	$\perp$	$\top$
$\perp$	$\perp$	$\perp$
$\top$	$\perp$	$\top$

$\vee$	$\perp$	$\top$
$\perp$	$\perp$	$\top$
$\top$	$\top$	$\top$

And we see that meet $\wedge$ is just *logical and* and join $\vee$ is just *logical or*.

For a general preorder, the meet and join of two elements need not exist and it need not be unique. For example, if $\mathcal{X} = \{x_1, x_2, x_3\}$ and has the discrete preorder, then $x_1 \vee x_2$ does not exist as the set $\{x_1, x_2\}$ does not have an upper bound. And in the preorder given by



then both b and d are meets of $\{a, e\}$. Note though that we must have $b \leq d$ and $d \leq b$ (i.e., $b \cong d$) as any two meets are lower bounds of the set. When meets are non-unique they must be equivalent; hence in an anti-symmetric preorder – called a partial order below – meets and joins are unique if they exist, so we can say *the* meet or *the* join.

Meets (greatest lower bounds) and joins (least upper bounds) generalize²⁷ the idea of maximum and minimum to situations where the optimum may not be achieved.

²⁷These are also called the *infimum* and *supremum* and are variously denoted by $\inf$ or glb and $\sup$ or lub in different treatments.

Puzzle 157. Does the set

$$\left\{ \frac{1}{n} \mid n \in [1..) \right\}$$

have a meet and a join? Find them if they exist. Are these unique?

The set $\{-1\} \cup (4\_17) \cup [20\_21]$ has a unique meet and join. What are they?

PARTIAL ORDERS. We have seen that for a general preorder, we may have $a \preccurlyeq b$ and $b \preccurlyeq a$, which represents a notion of equivalence $a \cong b$. If a preorder is *antisymmetric*, then $a \cong b$ implies that $a = b$. The result of this stronger requirement is called a *partial order*, and a set equipped with a partial order is called a *partially ordered set* or *poset* for short. The word partial here reminds us that two elements of the set need not be comparable.

A preorder $\preccurlyeq$ on a set $\mathcal{A}$ that is antisymmetric is called a **partial order**. That is,

1. $a \preccurlyeq a$ for all $a \in \mathcal{A}$,
2. $a \preccurlyeq b$ and $b \preccurlyeq c$ implies $a \preccurlyeq c$, and
3. $a \preccurlyeq b$ and $b \preccurlyeq a$ implies $a = b$.

We call $\langle \mathcal{A}, \preccurlyeq \rangle$ a **partially ordered set** or **poset** for short.

The set of real numbers with $\leq$ or $\geq$ is a partially ordered set. The Booleans $\mathbb{B}$ with $\perp \preccurlyeq \top$ is a partially ordered set. Given any preordered set, we can define a partial order on the set of equivalence classes induced by $\cong$.

Puzzle 158. Consider the earlier examples of preorders. Which of these represent partial orders?

Preorders and partial orders capture the essence of what we need in an ordered set, but there are some variations that arise enough to be worth mentioning.

TOTAL ORDERS. If we add the additional requirement that every pair of elements must be comparable, then a partial order becomes a *total order*. The word total is intended to evoke the comparability condition.

A partial order $\preceq$ on a set $\mathcal{A}$ for which every pair of elements is comparable is called a **total order**. That is,

1. $a \preceq a$ for all $a \in \mathcal{A}$,
2. $a \preceq b$ and $b \preceq c$ implies $a \preceq c$, and
3. $a \preceq b$ and $b \preceq a$ implies $a = b$.
4. $a \preceq b$ or $b \preceq a$ for all $a, b \in \mathcal{A}$.

STRICT ORDERS. Like the relation between $\leq$ and $<$, given a preorder or partial order, we can construct a **strict preorder** or **strict partial order**. We simply exclude all the values that are equivalent or equal.

If $\preccurlyeq$ is a preorder or partial order on $\mathcal{A}$, then we can construct, respectively, a **strict preorder** or **strict partial order** $\prec$ on $\mathcal{A}$ by

$$a \prec b \quad \text{if and only if} \quad a \preccurlyeq b \quad \text{and} \quad \text{not } b \preccurlyeq a.$$

Application: Counting

18

Chapter

Contents

18.1 Arithmetic Redux	794
18.2 How to Count	799
18.3 A Framework for Common Counting Problems	805
18.4 Unique Marbles in Labeled Pouches	811
18.5 Unique Marbles in Labeled Tubes	814
18.6 Identical Marbles, Labeled Batches	816
18.7 Identical Marbles, Unlabeled Batches	819
18.8 Unique Marbles in Unlabeled Pouches	820
18.9 Unique Marbles in Unlabeled Tubes	822
18.10 Permutations and Cycles	824
18.11 The Inclusion-Exclusion Principle	825

Imagine that you are sitting at a table, and in front of you is a pile of items, like marbles or jelly beans, that you would like to count. You pick up one item from the pile, move it to another part of the table and say “one”; you pick up another, move it, and say “two”; then another, “three”; and so on until every item in the original pile has been moved. What you have done is assign a label to every item in the pile, ensuring that every item has a label and that no item is labeled twice. You do not care in which order you have labeled the items, only that there is a one-to-one correspondence between items and labels. And you have chosen labels so that when you have finished you know exactly how many labels you have used – and thus how many items were in the pile.

The familiar process of counting can be understood as *constructing a bijection* between two sets. To count the number of items in a finite set $\mathcal{A}$, we construct a bijection between $\mathcal{A}$ and another set whose cardinality we know, like $[1..n]$ for some n . A bijection $g: [1..n] \rightarrow \mathcal{A}$ can be viewed as a way of labeling the elements of $\mathcal{A}$ with labels from $[1..n]$. $g(k)$ is the element of $\mathcal{A}$ with label $k \in [1..n]$, and $g^{-1}(a)$ is the label attached to element $a \in \mathcal{A}$. That g is a bijection means that n labels are enough to give every element exactly one label without any unused labels remaining. So, $\mathcal{A}$ has cardinality n . This is the Counting Principle on page 663 in Chapter 14. In this section, we will explore this principle to learn how to count. We use functions here not only to describe the bijections under the Counting Principle but also to represent what we are counting. For instance, as we have seen, we can identify subsets of a set $\mathcal{A}$ with functions $\mathcal{A} \rightarrow 2$, and counting subsets is then equivalent to counting such functions. A basic goal of this section is to solve a variety of common counting problems so that we can re-use and combine the solutions to solve more complicated problems. A higher-level goal is to see the power of the ideas and functional thinking we have developed in this Interlude when applied to a concrete challenge.

18.1 Arithmetic Redux

We start by considering three fundamental counting problems that have their roots in the basic arithmetic of natural numbers. The three results we will derive in this Section are, for finite sets $\mathcal{A}$ and $\mathcal{B}$:

$$\#(\mathcal{A} \sqcup \mathcal{B}) = \#\mathcal{A} + \#\mathcal{B} \quad (18.1)$$

$$\#(\mathcal{A} \times \mathcal{B}) = \#\mathcal{A} \cdot \#\mathcal{B} \quad (18.2)$$

$$\#(\mathcal{A} \rightarrow \mathcal{B}) = (\#\mathcal{B})^{\#\mathcal{A}}. \quad (18.3)$$

The number of ways to make a single choice from either of two separate menus is the sum of the choices available on the two menus. The number of ways to make two independent choices, one from the first menu and one from the second, is the product of the choices available on the two menus. And the number of ways to make one choice from the second menu for each item on the first menu, is an exponential in terms of the number of choices on the two menus.

We start with the first result from which the idea of addition arguably derived. Try to remember how you thought about addition at its most basic, before it had become familiar enough for you to have memorized the simple digit sums and multi-digit addition algorithm. You are sitting at a table with two piles of marbles in front of you that you have already counted, and you now want a count of the combined collection. Say the first pile has 4 stones and the second 3; then you could count off the four stones in the first pile – “one”, “two”, “three”, “four” – and then continue the count with successive numbers as you label the stones in the second pile – “five”, “six”, “seven” – for a total of $7 = 4 + 3$.

Two collections of things that we keep separate is just a **disjoint union** of finite sets $\mathcal{A}$ and $\mathcal{B}$, as defined in equation (9.10). Counting the number of items in the disjoint union abstracts our counting of two piles of marbles above: first we count the elements of $\mathcal{A}$ and then continue with successive numbers as we count the elements of $\mathcal{B}$. The essence of addition is the relation between the count of the disjoint union and the counts of its constituent sets.

If $\mathcal{A}$ and $\mathcal{B}$ are finite sets, then

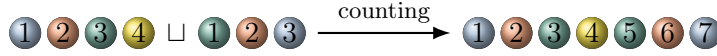
$$\#(\mathcal{A} \sqcup \mathcal{B}) = \#\mathcal{A} + \#\mathcal{B}. \quad (18.4)$$

If we think of $\mathcal{A}$ as one menu of choices and $\mathcal{B}$ as a menu of separate choices, then $\mathcal{A} \sqcup \mathcal{B}$ combines the two menus. Because it is a disjoint union, an item in $\mathcal{A}$ and an item in $\mathcal{B}$ are counted separately even if they are the same object. Equation (18.4) describes addition through the total number of choices in the combined menu.

Let’s see why equation (18.4) is true by constructing an explicit bijection. Let $m = \#\mathcal{A}$ and $n = \#\mathcal{B}$. This means that we have bijections $g: [1..m] \xrightarrow{\sim} \mathcal{A}$ and $h: [1..n] \xrightarrow{\sim} \mathcal{B}$. We want to find a bijection $f: [1..m+n] \xrightarrow{\sim} \mathcal{A} \sqcup \mathcal{B}$.

Think of g and h as ways of *labeling* elements of $\mathcal{A}$ and $\mathcal{B}$ respectively, where for instance, $g(1)$ is the element of $\mathcal{A}$ with label 1. From this, we want to label the elements of $\mathcal{A} \sqcup \mathcal{B}$, and we do this by tweaking the labels on elements of $\mathcal{B}$, adding m to each such label, ensuring that every element of the disjoint union has a distinct

label.



Specifically, the elements of $\mathcal{A}$ we label 1 through m and the elements of $\mathcal{B}$ we label $m + 1$ through $m + n$. This mimics the counting process describes earlier: counting items in $\mathcal{A}$ up to m and then continuing the count for the n items in $\mathcal{B}$.

More formally, as described in Example 13.6 diagram (13.12), the properties of the disjoint union¹ imply that for any functions ℓ_1, ℓ_2 in

$$\begin{array}{ccc}
 [1 \dots m+n] & \xleftarrow{\ell_2} & \mathcal{B} \\
 \uparrow \ell_1 & \nwarrow \ell_1 \sqcup \ell_2 & \downarrow \text{inj}_2 \\
 \mathcal{A} & \xrightarrow{\text{inj}_1} & \mathcal{A} \sqcup \mathcal{B}
 \end{array} \quad (18.5)$$

¹Also see Example 10.10. The inj_i “inject” a value into the disjoint union with label i .

there is a *unique* function $\ell_1 \sqcup \ell_2$ that makes the diagram commute.² Let $\ell_1 = g^{-1}$ and $\ell_2 = (m + h^{-1}(\bullet))$; these just use our bijections for $\mathcal{A}$ and $\mathcal{B}$ with the latter shifted to fit in $[1 \dots m+n]$. Then $\ell_1 \sqcup \ell_2$ is an injection because

- (i) if $a \neq a' \in \mathcal{A}$, $(\ell_1 \sqcup \ell_2)(\text{inj}_1(a)) = g^{-1}(a) \neq g^{-1}(a') = (\ell_1 \sqcup \ell_2)(\text{inj}_1(a'))$ because inj_1 and g^{-1} are injections;
- (ii) if $b \neq b' \in \mathcal{B}$, $(\ell_1 \sqcup \ell_2)(\text{inj}_2(b)) = m + h^{-1}(b) \neq m + h^{-1}(b') = (\ell_1 \sqcup \ell_2)(\text{inj}_2(b'))$ because inj_2 and h^{-1} are injections;
- (iii) if $a \in \mathcal{A}$ and $b \in \mathcal{B}$, $(\ell_1 \sqcup \ell_2)(\text{inj}_1(a)) \leq m$ and $(\ell_1 \sqcup \ell_2)(\text{inj}_2(b)) > m$ and thus are unequal.

²As seen earlier, $(\ell_1 \sqcup \ell_2)(\langle x, j \rangle) = \ell_j(x)$.

And $\ell_1 \sqcup \ell_2$ is a surjection as well because $(\ell_1 \sqcup \ell_2)(\text{inj}_1(a)) = g^{-1}(a)$ covers $[1 \dots n]$ for $a \in \mathcal{A}$ since g^{-1} is a surjection and $(\ell_1 \sqcup \ell_2)(\text{inj}_2(b)) = m + h^{-1}(b)$ covers $[m+1 \dots m+n]$ for $b \in \mathcal{B}$ since h^{-1} is a surjection. Hence, $\ell_1 \sqcup \ell_2$ is a bijection and we define $f^{-1} = \ell_1 \sqcup \ell_2$. We have thus established equation (18.4).

We can also derive an expression for f by reversing this process. We use the element of $\mathcal{A}$ with g -label k when $k \in [1 \dots m]$ and the element of $\mathcal{B}$ with h -label $k - m$ when $k \in [m+1 \dots m+n]$. Using indicators, f is defined by

$$f(k) = \text{inj}_1(g(k)) \{1 \leq k \leq m\} + \text{inj}_2(h(k-m)) \{k > m\}.$$

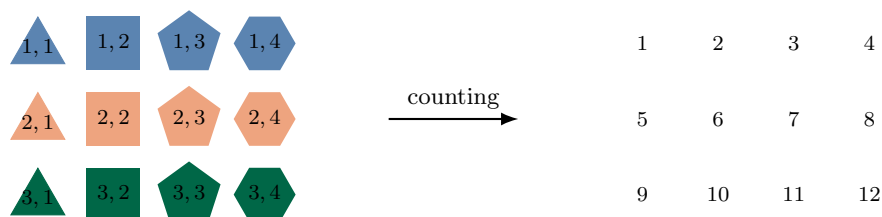
That f is a bijection follows because g and h are bijections.

Puzzle 159. Confirm for yourself that $f \circ f^{-1} = \text{id} = f^{-1} \circ f$.

Puzzle 160. The empty set acts as a loose kind of identity for disjoint union. In particular, if $\mathcal{A}$ is a finite set, then there is a bijection between $\mathcal{A}$ and $\mathcal{A} \sqcup \{\}$.

Why is this true? What is the bijection?

Next, we turn to multiplication. When we first learn how to multiply, we arrange items in rectangles. For instance, to compute 3×4 , we might put three items in each of four stacks and count the resulting collection:



Each such item has a “coordinate” here represented by a pairing of a color and shape or by a pairing of a row index and a column index, as shown. As we traverse this grid systematically (e.g., shapes or row indices varying fastest), we increment a count. More generally, we have two collections (e.g., colors and shapes), and we are counting all the ways to pair an element of the first collection with an element of the second. That is, we are counting the **Cartesian product** of two sets as defined in equation (9.8). The essence of multiplication is the relation between the number of items in this grid and the numbers of rows and columns.

If $\mathcal{A}$ and $\mathcal{B}$ are finite sets, then

$$\#(\mathcal{A} \times \mathcal{B}) = \#\mathcal{A} \cdot \#\mathcal{B}. \quad (18.6)$$

The cardinality of the product is the product of the cardinalities. If $\mathcal{A}$ and $\mathcal{B}$ are menus of choices, then we consider the total number of ways in which we can independently choose one item from each menu. If the items are arranged in a grid as above, we can think of the first choice as c

Equation (18.6) says that if $\mathcal{A}$ has cardinality m and $\mathcal{B}$ has cardinality n , then there is a bijection between $\mathcal{A} \times \mathcal{B}$ and $[1..mn]$. Again, we use bijections $g: [1..m] \rightarrow \mathcal{A}$ and $h: [1..n] \rightarrow \mathcal{B}$ that come from the cardinalities of the two sets.

As in Example 13.6 diagram (13.11), the properties of the Cartesian product

imply that for functions p_1, p_2 in

$$\begin{array}{ccc}
 \mathcal{A} \times \mathcal{B} & \xrightarrow{\text{proj}_2} & \mathcal{B} \\
 \text{proj}_1 \downarrow & \nwarrow p_1 \vee p_2 & \uparrow p_2 \\
 \mathcal{A} & \xleftarrow{p_1} & [1 \dots mn]
 \end{array} \tag{18.7}$$

there is a *unique* function $p_1 \vee p_2$ that makes the diagram commute.³

³As seen earlier,
 $(p_1 \vee p_2)(c) = \langle p_1(c), p_2(c) \rangle$.

Think of the indices in $[1 \dots mn]$ arranged in m rows and n columns (assuming $n \geq m$); for each index, we extract the row and column index and use the pair of objects in $\mathcal{A}$ and $\mathcal{B}$ with those labels. We define p_1 and p_2 to first extract from an index in $[1 \dots mn]$ the corresponding row and column “coordinates” and then get the g and h labels for these respectively. We use the fact that each $i \in [1 \dots mn]$ can be written uniquely as $1 + (k - 1) + (j - 1) \max(m, n)$ for $j \in [1 \dots m]$ and $k \in [1 \dots n]$. So, we define $p_1(i) = g(1 + (i - 1) \text{div} \max(m, n))$ and $p_2(i) = h(1 + (i - 1) \bmod \max(m, n))$, both of which are surjections. Using the uniqueness from the previous paragraph, we can then take $f = p_1 \vee p_2$. This yields

$$f(i) = \langle g(1 + (i - 1) \text{div} \max(m, n)), h(1 + (i - 1) \bmod \max(m, n)) \rangle,$$

or equivalently and more succinctly,

$$f(1 + (k - 1) + (j - 1) \max(m, n)) = \langle g(j), h(k) \rangle.$$

Because g and h are injections, changing i (and thus j or k) changes the result, so f is an injection. Because g and h are surjections, every element of $\mathcal{A} \times \mathcal{B}$ can be produced for some j and k pair, so f is a surjection. We have thus established equation (18.6).

The inverse function is given by

$$f^{-1}(\langle a, b \rangle) = 1 + (h^{-1}(b) - 1) + (g^{-1}(a) - 1) \max(m, n).$$

You can see directly from the previous equation that it inverts f .

Both (18.4) and (18.6) generalize to any finite combinations as well:

If $\mathcal{I}$ is a finite set and $\langle i \rangle \mapsto \mathcal{A}_i$ is a family of finite sets indexed by $\mathcal{I}$, then

$$\# \bigsqcup_{i \in \mathcal{I}} \mathcal{A}_i = \sum_{i \in \mathcal{I}} \# \mathcal{A}_i \quad (18.8)$$

$$\# \times_{i \in \mathcal{I}} \mathcal{A}_i = \prod_{i \in \mathcal{I}} \# \mathcal{A}_i. \quad (18.9)$$

These equations describe addition as aggregating over a collection of *alternative choices* and describe multiplication as aggregating a series of *independent choices*.

Finally, we consider the exponentiation of natural numbers with natural number powers. When we first learn this operation, we think about a power like n^m as multiplying n by itself m times, but we can see this another way as well: making a free choice of m elements from a set containing n elements, allowing repetitions. We can cast each such choice as a choice of a function between two sets.

Let $\mathcal{A}$ and $\mathcal{B}$ be finite sets with $m = \# \mathcal{A}$ and $n = \# \mathcal{B}$. Defining a function $\mathcal{A} \rightarrow \mathcal{B}$ means picking one of the n elements of $\mathcal{B}$ for each of the m elements of $\mathcal{A}$. That is, each function uniquely specifies a free choice of m elements of $\mathcal{B}$, each associated with an element of $\mathcal{A}$. Conversely, if we freely choose m elements of $\mathcal{B}$, we can associate them with elements of $\mathcal{A}$ in some fixed, pre-specified order, which gives us a function $\mathcal{A} \rightarrow \mathcal{B}$. This mapping is a bijection, so there are n^m such functions.

If $\mathcal{A}$ and $\mathcal{B}$ are finite sets, then

$$\#(\mathcal{A} \rightarrow \mathcal{B}) = (\# \mathcal{B})^{\# \mathcal{A}}. \quad (18.10)$$

Equation (18.10) explains the common notation $\mathcal{B}^{\mathcal{A}}$ for the set $\mathcal{A} \rightarrow \mathcal{B}$; it makes the equation more elegant $\#(\mathcal{B}^{\mathcal{A}}) = (\# \mathcal{B})^{\# \mathcal{A}}$. We will continue using our original notation because it is more directly clear, but be aware of this common alternative.

18.2 How to Count

The Counting Principle tells us that to count any collection, we need only construct a bijection with a collection whose cardinality we know. The results of the previous Section give us tools: if we can decompose a set into pieces – as a product or disjoint union of sets we can count – we know how to combine counts of the pieces. Here, we will see this in action on a few familiar counting problems. A key step in the method is finding two distinct descriptions of the set we want to count. These descriptions give us the sets between which to construct a bijection.

In how many ways, can we rearrange a list of n items? Given a set $\mathcal{A}$ with $\#\mathcal{A} = n$, we lay the elements of $\mathcal{A}$ out in order; any rearrangement of this list is a bijection $\mathcal{A} \succ \mathcal{A}$, i.e., a **permutation**.⁴ Our goal then is to count the set $\mathcal{A} \succ \mathcal{A}$ of permutations of a set with cardinality n .

⁴See also page 674.

For this purpose, we only need to count the permutations of the set $[1..n]$. If $\#\mathcal{A} = n$, there exists a bijection $\ell: [1..n] \succ \mathcal{A}$ where for any permutation p of $\mathcal{A}$ there is a unique permutation q of $[1..n]$ such that

$$\begin{array}{ccc} [1..n] & \xrightarrow{q} & [1..n] \\ \downarrow \ell & & \downarrow \ell \\ \mathcal{A} & \xrightarrow{p} & \mathcal{A} \end{array}$$

commutes, and vice versa. Rewriting this diagram in two equivalent ways (because ℓ is a bijection) will make this clearer.

$$\begin{array}{ccc} [1..n] & \xrightarrow{q} & [1..n] \\ \downarrow \ell & & \uparrow \ell^{-1} \\ \mathcal{A} & \xrightarrow{p} & \mathcal{A} \end{array} \qquad \begin{array}{ccc} [1..n] & \xrightarrow{q} & [1..n] \\ \uparrow \ell^{-1} & & \downarrow \ell \\ \mathcal{A} & \xrightarrow{p} & \mathcal{A} \end{array}$$

If we specify p , then there is a *unique* $q = \ell \circ p \circ \ell^{-1}$ that makes the diagram commute. Conversely, if we specify q , there is a *unique* $p = \ell^{-1} \circ q \circ \ell$ that makes it commute. Put another way: there is a bijection between the permutations of $\mathcal{A}$ and the permutations of $[1..n]$, so all have the same cardinality.

Let $\mathcal{P}_n$ be the set of permutations of $[1..n]$. We have a good, informal argument to count $\#\mathcal{P}_n$: a permutation has n choices for where to map 1, then $n - 1$ choices for where to map 2 (avoiding 1's destination), then $n - 2$ choices for where to map 3 (avoiding previous destinations), and so on until the position of n is forced to one destination. This gives $\#\mathcal{P}_n = n!$.

Such informal arguments are fine – we use them all the time – but to wield them effectively and *correctly*, it is helpful to study the formal argument that underlies it. The informal argument is expressed in terms of a sequence of choices from “menus” of different sizes, and as we saw in the previous Section, this suggests an underlying Cartesian product.

And indeed: for every $n \in [1..)$, there is a bijection f between $\mathcal{P}_n$ and the set $[1..n] \times \mathcal{P}_{n-1}$. Let $p \in \mathcal{P}_n$ and $m = p(n)$. Define $q \in \mathcal{P}_{n-1}$ to look like p except it “skips” m :

$$q(k) = p(k) \{p(k) < m\} + (p(k) - 1) \{p(k) > m\}.$$

Then, we set $f(p) = \langle m, q \rangle$. The function f is invertible with

$$f^{-1}(\langle m, q \rangle)(j) = q(j) \{j < n \wedge q(j) < m\} + (q(j)+1) \{j < n \wedge q(j) \geq m\} + m \{j = n\}$$

and hence f is a bijection.

It follows by equation (18.9) that $\#\mathcal{P}_n = n\#\mathcal{P}_{n-1}$ for each $n \in [1..)$. We know that $\#\mathcal{P}_1 = 1$ because there is only one function $\{1\} \rightarrow \{1\}$ and that $\#\mathcal{P}_0 = 1$ since there is one function from the empty set to any set (see Example 10.11). Hence, by iterating the product (or by formal induction), we have $\#\mathcal{P}_n = n!$ for all $n \in [0..)$.

How many subsets of an n -element set are there? In Example 10.13, we showed that there is a bijection between subsets of an n -element set and bit strings of length n . Moreover, we know that there is a bijection between the latter and the set $[0..2^n)$ mapping bits $\langle b_{n-1}, \dots, b_1, b_0 \rangle$ to $\sum_{k=0}^{n-1} b_k 2^k$. (This is just the binary representation of integers.) Hence, there are 2^n subsets of an n -element set?

How many subsets of an n -element set are there with cardinality k ? If $\mathcal{A}$ is a set with cardinality n , we define $\binom{\mathcal{A}}{k}$ to be the set containing all subsets of $\mathcal{A}$ with cardinality k . We define the **binomial coefficient** $\binom{n}{k}$ for $n, k \in \mathbb{N}$ to be the cardinality of $\binom{\mathcal{A}}{k}$.

We can find the binomial coefficients for small values of n and k . For instance, because the empty set is a subset of every set and is the unique set of size 0, we have $\binom{n}{0} = 1$ for every n . Similarly, $\binom{\mathcal{A}}{1}$ contains exactly the sets $\{a\}$ for $a \in \mathcal{A}$, so $\binom{n}{1} = n$. We would like to find $\binom{n}{k}$ in general. As above, it is sufficient to use $\mathcal{A} = [1..n]$.

We will give two equivalent but distinct descriptions of a set. On the one hand, we have the set of permutations $\mathcal{P}_n$, which we know to have cardinality $n!$. On the other, for any $k \in [0..n]$, we use $\binom{\mathcal{A}}{k} \times \mathcal{P}_k \times \mathcal{P}_{n-k}$. We establish equivalence by constructing a bijection between them. Informally, if we choose subset of size k , we can permute those k items in any way and list them first, followed by the remaining $n-k$ items listed in any order, yielding a permutation. Conversely, given a permutation of n items, choose the elements that map to $[1..k]$ as the subset of size k , and identify the permutation of those items and the permutation of the remaining items.

Formally, given $\mathcal{B}(\binom{\mathcal{A}}{k})$, $p \in \mathcal{P}_k$, and $q \in \mathcal{P}_{n-k}$, list the elements of $\mathcal{B}$ in increasing order $b_1 < b_2 < \dots < b_k$ and the elements of $\mathcal{A} - \mathcal{B}$ in increasing order $b'_1 < b'_2 < \dots < b'_{n-k}$. Define $f \in \mathcal{P}_n$ for $i \in [1..n]$ by

$$f(i) = b_{p(i)} \{i \leq k\} + b'_{q(i-k)} \{k < i \leq n\},$$

giving a permutation of $\mathcal{A}$. Conversely, given $f \in \mathcal{P}_n$, let $\mathcal{B} = f^{-1}([1 \dots k])$ with elements of $\mathcal{B}$ and $\mathcal{A} - \mathcal{B}$ listed in order as above. Then, $p(j) = b_j$ for $j \in [1 \dots k]$ and $q(j) = b_j - k$ for $j \in [1 \dots n - k]$ are in $\mathcal{P}_k$ and $\mathcal{P}_{n-k}$. This gives $\langle \mathcal{B}, p, q \rangle \in \binom{\mathcal{A}}{k} \times \mathcal{P}_k \times \mathcal{P}_{n-k}$, and the two maps are inverses of each other. It follows that $n! = \binom{n}{k} k! (n - k)!$, and so

$$\binom{n}{k} = \frac{n!}{k!(n - k)!} \quad (18.11)$$

is the number of subsets of size k from an n -element set.

This Counting Principle approach is versatile and can help us establish and understand equations with a combinatorial interpretation. For instance, equation (18.11) shows algebraically that $\binom{n}{k} = \binom{n}{n-k}$ because the right side is symmetric in k and $n - k$. Constructing an explicit bijection seems like more work to establish the same relation, but from that effort, we get a clearer understanding. Here are two examples.

Show that $\binom{n}{k} = \binom{n}{n-k}$.

To establish this identity, we need only construct a bijection between $\binom{[1..n]}{k}$ and $\binom{[1..n]}{n-k}$. But this is direct: map a subset to its complement, $\mathcal{B} \mapsto \mathcal{B}^c$. This takes a subset of size k to a subset of size $n - k$ and vice versa; the function is its own inverse.

Puzzle 161. Give a counting argument for the claim $\binom{n}{k} = 0$ for $k > n$.

Show that $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

Establishing this algebraically is possible but mechanical; better we construct a bijection between the set $\binom{[1..n]}{k}$, whose cardinality is the left side of the identity, and the set $\binom{[1..n]}{k} \sqcup \binom{[1..n]}{k-1}$, and whose cardinality is the right side by equation (18.8). This bijection not only proves the identity but *explains it*. Any subset of $[1 \dots n]$ of size k must either exclude n and thus be a subset of $[1 \dots n - 1]$ of size k or it is $\{n\}$ unioned with a subset of $[1 \dots n - 1]$ of size $k - 1$. This specifies the bijection, but for concreteness, we have formally:

$$f(\mathcal{B}) = \begin{cases} \text{inj}_1(\mathcal{B}) & \text{if } n \notin \mathcal{B} \\ \text{inj}_2(\mathcal{B} - \{n\}) & \text{otherwise,} \end{cases}$$

for $\mathcal{B} \in \binom{[1..n]}{k}$.

Show that $(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$ for $n \in \mathbb{N}$.

Assume (for the moment) that $x, y \in \mathbb{N}$, and let $\mathcal{A}$ and $\mathcal{B}$ be sets with respective cardinalities x and y . Then, the set $(\mathcal{A} \sqcup \mathcal{B})^n$ can be viewed as describing n independent choices from $\mathcal{A}$ or $\mathcal{B}$; think of n customers at an all-day diner making a choice from *either* the breakfast menu *or* the lunch menu. From equations (18.9) and (18.8), this set has cardinality $(x + y)^n$, which is the left-side of our identity.

To establish the identity, we consider separately the cases where k customers choose to order from the breakfast menu. For each k , we have to choose which k customers choose from $\mathcal{A}$, which choices from $\mathcal{A}$ those customers make, and which choices from $\mathcal{B}$ the remaining customers make. That is, we construct a bijection of $(\mathcal{A} \sqcup \mathcal{B})^n$ with the set $\bigsqcup_{k=0}^n \binom{[1..n]}{k} \times \mathcal{A}^k \times \mathcal{B}^{n-k}$. Suppose $\langle \mathcal{C}, a, b \rangle$ belongs to the latter set, let $c_1, \dots, c_k$ be the elements of $\mathcal{C}$ in increasing order and $c'_1, \dots, c'_{n-k}$ be the elements of $[1..n] - \mathcal{C}$ in increasing order. Then, we map $\langle \mathcal{C}, a, b \rangle$ to $\langle u_1, \dots, u_n \rangle$ in $(\mathcal{A} \sqcup \mathcal{B})^n$, where

$$u_i = a_j \{i = c_j\} + b_j \{i = c'_j\}.$$

You should convince yourself that this is a bijection.

We have thus established the identity, called the **binomial theorem**, for natural number x, y . However, this implies that the theorem holds for *any* numbers x, y . The reason is that $(x + y)^n$ is a polynomial of degree n in two variables, and a polynomial is determined by its values on the natural numbers.

In how many ways can we partition a set of n elements into disjoint subsets of sizes $c_1, c_2, \dots, c_k$ for $k \in [2..]$, where $c_1 + \dots + c_k = n$?

The idea is that we have k mutually exclusive categories into which we place each of the items. We denote this a quantity with a **multinomial coefficient**. We usually write this as

$$\binom{n}{c_1, \dots, c_{k-1}},$$

leaving the c_k term implicit, giving an immediate generalization of binomial coefficients for which $k = 2$. However, without any loss of clarity, we can also write it as $\binom{n}{c_1, \dots, c_{k-1}, c_k}$, with the final count, or $\binom{n}{c}$, with the tuple $c = \langle c_1, \dots, c_k \rangle$.

To find an expression for the multinomial coefficient, we again look at all permutations. We choose a set of c_1 items for category 1 and a way to arrange them in some order, then we choose a set of c_2 items from the remaining $n - c_1$ and a way to arrange them in some order, then we choose a set of c_3 items from the remaining $n - c_1 - c_2$ and a way to arrange them in some order, and so on until we have one

set of size $c_k = n - c_1 - \cdots - c_{k-1}$ and a way to arrange them in some order. Every permutation can be constructed like this by dividing its output into blocks of lengths $c_1, c_2, \dots, c_k$, and conversely, given such a collection of subsets and orderings we can concatenate the ordered subsets to get a permutation. We thus construct a bijection between the set $\mathcal{P}_n$ of permutations and the set

$$\binom{[1..n]}{c_1} \times \mathcal{P}_{c_1} \times \binom{[1..n-c_1]}{c_2} \times \mathcal{P}_{c_2} \times \binom{[1..n-c_1-\cdots-c_{k-1}]}{c_k} \times \mathcal{P}_{c_k}.$$

Each pair of terms corresponds to a choice of subsets and a choice of ordering of that subset. Hence, by the results above and equation (18.9)

$$n! = \binom{n}{c_1} c_1! \binom{n-c_1}{c_2} c_2! \binom{n-c_1-c_2}{c_3} c_3! \cdots \binom{n-c_1-c_2-\cdots-c_{k-1}}{c_k} c_k!,$$

which gives us

$$\binom{n}{c_1, \dots, c_{k-1}} = \frac{n!}{c_1! c_2! \cdots c_{k-1}! (n - c_1 - \cdots - c_{k-1})!}. \quad (18.12)$$

When $k = 2$, this reduces to our expression for the binomial coefficients.

Puzzle 162. Generalizing the binomial theorem, make an educated guess of an expression for $(x_1 + x_2 + \cdots + x_p)^n$ in terms of multinomial coefficients.

The Counting Principle method can be used to establish and understand many combinatorial identities. Try it with the next puzzle.

Puzzle 163. Use the method to establish that for $n > 0$

$$\sum_{k=0}^n \binom{n}{k} (-1)^k = 0.$$

It will help to rearrange the equation so that only positive terms are on each side. What do the terms on the left and right represent when you do that? Now, establish the identity by building a bijection.

Hint: Consider the mapping that takes a subset of $[1..n]$ as input and either adds n if it is absent or removes it if it is present.

While we are on the subject of binomial coefficients, it is worth noting that they can be defined more generally.

Let k be an integer and r be an arbitrary real (or even complex) number. Then the binomial coefficient $\binom{r}{k}$ is defined by

$$\binom{r}{k} = \frac{r^{\underline{k}}}{k!} \{k \geq 0\} \quad (18.13)$$

in terms of the **falling factorial power**

$$r^{\underline{k}} = r(r-1) \cdots (r-k+1). \quad (18.14)$$

For instance, $\binom{1/2}{2} = -\frac{1}{8}$ and $\binom{-1}{k} = (-1)^k$. And the binomial theorem generalizes too: for $|x| < 1$ and any r

$$(1+x)^r = \sum_{k=0}^{\infty} \binom{r}{k} x^k \quad (18.15)$$

18.3 A Framework for Common Counting Problems

Counting problems arise frequently in probability, including the problems we saw in the last Section and others like

- How many ways are there to make k independent choices from n items?
- How many sequences of length k can we form using n items without repetition? With repetitions allowed?
- How many multisets of size k can we form on a base set of n elements?⁵

But this just scratches the surface; there are many, many variations. While we can solve each problem individually, it would be helpful if we could organize the problems in a conceptually useful way. The good news is that we can: a wide range of problems can be cast into a common framework with which we can solve and relate them.

Here we describe a framework;⁶ we solve most of these problems in subsequent sections. The basic setup: we have a set of M marbles that we will to arrange into B batches. Our goal is to count the number of ways to **arrange the marbles in batches** subject to various conditions/constraints.

Let the objects in the set $\mathcal{M}$ represent the marbles with $M = \#\mathcal{M}$ and the objects in nonempty set $\mathcal{B}$ represent the batches with $B = \#\mathcal{B}$. The variations we consider include:

1. The marbles can be **uniquely colored** or **identically colored**.

In the former case, we can identify individual marbles, so permuting marbles gives a distinct arrangement; in the latter case, we cannot distinguish among the marbles, so permuting marbles does not change the arrangement.

⁵See Example 10.14 and page 571.

⁶These problems are selected from the “Twelvefold Way” by [Stanley1986] and related “Thirtyfold Way” by [Proctor2007], both of which inspired our framework.

2. The marbles in a batch can be stored in **tubes** or in **pouches**. The tubes open only on one side and have diameter just bigger than a marble.

In the former case, re-ordering uniquely colored marbles in a tube gives a distinct arrangement; in the latter case, the marbles' positions are ill-defined, so it does not. (This distinction is unimportant with identically colored marbles.)

3. The batches can be **labeled** with distinct identities or **unlabeled**.

In the former case, arrangements are distinguished by the contents of individual batches; in the latter case, arrangements with the same pattern of contents are treated as the same.

4. We can constrain the size of the batches, (i) putting **no constraint** on the batch sizes, (ii) requiring that all batches have size at most 1, which we call **binary** batches, (iii) requiring that all batches be **nonempty**, or (iv) requiring a specific **profile** of batch sizes, $\beta: [0 \dots M] \rightarrow [0 \dots B]$ with β_s batches of size s , where $\sum_{s=0}^M \beta_s = B$ and $\sum_{s=0}^M s\beta_s = M$.

The marbles here are of course a metaphor, and the elements of $\mathcal{M}$ can be any type of object. Saying the marbles are uniquely colored means that we distinguish among and label individuals in $\mathcal{M}$ and care about where an individual ends up in the arrangement. The colors represent any unique identifiers on the objects in $\mathcal{M}$, such as numbers, names, labels, or ... colors. Tubes are, in less metaphoric language, just *lists* of marbles, where we pay attention to the order/position of placement. Pouches are *sets* of marbles, where we do not.



FIGURE 18.1. Example arrangements with $M = 7$ and $B = 4$. Batch labels, when used, are I–IV

Figure 18.1 shows several example arrangements with $M = 7$ and $B = 4$. If batches are labeled (I–IV), marbles are unique, and batches stored in tubes, all these arrangements are distinct. If batches are labeled, marbles are unique, but batches stored in pouches, then arrangements 1 and 2 are the same as the same marbles are allocated to the same batches. With unlabeled batches, we “erase” the I–IV labels and distinguish arrangements of batches only by the pattern of contents. If batches are unlabeled, marbles are unique, and batches stored in tubes, then arrangements 1 and 3 are the same. These two arrangements have the same pattern: a 357, a 12, a 46, and an empty tube. If batches are unlabeled, marbles are unique, and batches stored in pouches, then arrangements 1, 2, and 3 are all the same as they all have the same pattern of batches with the same marbles, since the ordering of the marbles is ignored. Looking at these cases with identically colored marbles instead, the contents of a batch are determined by *how many* marbles it contains because the marbles are indistinguishable. There is no distinction between tubes and pouches in this case. If batches are labeled and marbles are identical, arrangements 1 and 2 are the same. If batches are unlabeled and marbles are identical, arrangements 1, 2, 3, and 4 are the same, as they all share the pattern of 3, 2, 2, and 0 marbles in a batch.

To get a feel for how this framework is applied, consider several concrete instances counting problems and where they fit in Table 18.1.

- A small movie theater has four rows of sixteen seats each. Twenty moviegoers have come for the show. In how many ways can the customers be seated?

Here, the marbles are customers, and the batches are seats. The marbles are *identically* colored as we do not distinguish seating arrangements if the same seats are occupied but by different individuals. Each batch can hold at most one marble, and so are *binary*. The batches *labeled* as we distinguish seats by their position in the theater. (Note that with binary batches, there is no distinction between tubes and pouches as batch storage.)

- Alice, Bob, Carlos, and Danielle each write a letter to a mutual friend. There is an undifferentiated pile of n empty envelopes on the table, and each takes an envelope for his or her letter. In how many ways, can the letters be placed into envelopes.

Here, the marbles are the letters, and the batches are envelopes. The marbles are *uniquely* colored as distinguish among the letters – and their allocations to envelopes – based on who wrote them. Each batch can hold at most one marble, so again the batches are *binary*. These batches *unlabeled*; we do not

care which envelope holds a particular letter only whether there is an envelope for that letter.

- Amanda has 100 books to place on a stack of seven shelves. She places the books starting from the left side of a shelf with no gaps. In how many ways can Amanda shelve the books?

Here the marbles are books, and the batches are shelves. Whether the marbles are uniquely or identically colored depends on context. Do we care about the position of particular books (due to e.g., author, title, size, color, format) or not? If so, marbles are unique; if not, identical. Let's say, for the purposes of discussion, that we do, so marbles are *uniquely* colored. The batches *labeled* because they have a distinct identity and we care (for visual impression and access) which batch a marble is placed in. (For instance, if there were only one book, we would distinguish between a shelving with the book near the floor or ceiling versus in the middle.) The batch storage is in *tubes* because we pay attention to the order of the books on a shelf. And we put no constraint on the batch size; shelves can be empty or full or in between.

- A northern toy distributor has many wrapped gifts, each addressed to a specific child. Workers in the shop place the gifts in sacks for delivery. In how many ways can the gifts be arranged into sacks?

Here the marbles are gifts, and the batches are sacks. The marbles are *uniquely* colored because we care about which children's gifts are in each sack. (For instance, the contents of a sack will determine the path of a delivery sleigh.) The batches *unlabeled* as we do not care which sack holds any particular gifts, and the batch storage is in *pouches* as we do not track the order or position of a gift within the sack. The batches are required to be *nonempty*; there is no need to deliver the contents of an empty sack.

- Five cards are drawn in sequence from a well-shuffled standard deck of 52 playing cards. How many possible sequences of cards are there?

Perhaps counter-intuitively, the marbles here are positions in the sequence of length five, and the batches are individual cards. We are allocating marbles to batches, which we can think of as assigning a sequence position to a card. The marbles are *uniquely* colored, and the batches are *labeled*. That is, we care about which marble is in which batch (the position assigned to a card) and about the identity of the batches (cards). Because cards can be assigned a

position or not, the batches are *binary*; as such, there is no distinction between tubes and pouches in this case.

- A town must transport 3,000 bags of sand from a depot to a river levee in danger of flooding using 75 trucks. In how many ways can the bags of sand be arranged for transport?

Here, the marbles are bags of sand, and the batches are trucks. The marbles are *identically* colored, and the batches are *unlabeled*, as we do not care about the identity of particular bags or trucks. The batch size is *unconstrained*.

- During one day, a business receives jobs from 10 different clients and queues them for processing. One client submits three jobs, another four jobs, and the rest one job. How many arrangements of the queue are there, where we do not distinguish arrangements that reorder the jobs only of one client?

Here, the batches are clients, and the marbles are positions in the queue. The marbles are *uniquely* colored, and the batches are *labeled*, as we care about the identity of the clients. The batches are stored in *pouches* as we do not distinguish among orderings of a client's jobs. The batch sizes have a specified *profile*, with one batch of size four, one of size three, and eight of size 1.

As these examples show, there can be some art to identifying the marbles and the batches. The context of what we are counting determines the Labeled/Unlabeled and Tubes/Pouches distinctions; whether we distinguish arrangements based on the identity of individual marbles or batches.

We organize the counting problem variations in Table 18.1, and we will fill in the entries as we proceed. Observe that if we know the count $n(M, B)$ in a cell of the table for any M and B , then we can find the count *allowing any number of batches*⁷ from the table by summing over B from 1 to M : $n(M) = \sum_{B=1}^M n(M, B)$. This count is often of independent interest. We will derive these counts by counting the number of functions, or equivalence classes of functions, of a particular type.

⁷Remember that there must be at least one batch.

It should not be a surprise at this point in the Interlude that *functions* play a key role in our thinking about these counting problems. We represent an arrangement marbles in batches by either a function or an equivalence classes of functions. We use two types of functions for this purpose. When batches are stored in pouches or marbles are identically colored, we use *allocation functions* $a: \mathcal{M} \rightarrow \mathcal{B}$. An allocation function a describes a way of creating batches directly, where marble $m \in \mathcal{M}$ is allocated to batch $a(m) \in \mathcal{B}$. A restriction to binary batches is equivalent to a being an injection; a restriction to nonempty batches is equivalent to a being a surjection.

Batch Identity	Marble Coloring	Batch Storage	Unconstrained Batches	Binary Batches	Nonempty Batches	Batch Profile β
Labeled	Unique	Tubes				
	Unique	Pouches				
	Identical	–				
Unlabeled	Unique	Tubes				
	Unique	Pouches				
	Identical	–				

TABLE 18.1. Our framework for common counting problems to be filled in below, see Table 18.7.

When batches of uniquely colored marbles are stored in tubes, an allocation function does not give enough information to describe the batches. Knowing which batch a marble belongs to is insufficient, we also need to know the marble's position in the tube. For these cases, we describe arrangements by *grouping functions* $g: \mathcal{M} \sqcup [1..B] \rightarrow [1..M+B]$. A grouping function g maps every marble and every batch identifier in $\mathcal{M} \sqcup [1..B]$ to a *distinct* slot/index in $[1..M+B]$, with a batch identifier in rightmost slot $M+B$. The marbles assigned slots directly to the left of a batch identifier belong to that batch, listed in order from its tube's open end. In this way, a grouping function fully specifies an arrangement of marbles. Figure 18.2 illustrates a grouping function g for $M = 7$ and $B = 4$, where the rectangles represent batch identifiers and the balls represent marbles. This assigns marbles 4

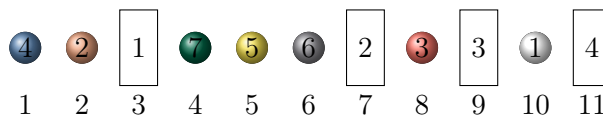


FIGURE 18.2. A representation of a grouping function with $M = 7$ and $B = 4$.

and 2 to batch 1, with 4 closer to the tube opening; marbles 7, 5, and 6 to batch 2 in order from the open end; marble 3 to batch 3; and marble 1 to batch 4.

In general, we put three constraints on any grouping function g : (i) g must be an injection, meaning that every marble or batch identifier maps to a distinct slot, (ii) $g(\text{inj}_2(B)) = M+B$, meaning that the last batch identifier maps to the last slot, and (iii) $g \circ \text{inj}_2$ is monotone increasing on $[1..B]$, meaning that batch identifiers are listed in order in the slots. Note that because $\#(\mathcal{M} \sqcup [1..B]) = M+B$, (i) implies that g must be a bijection. Every grouping function lists the

batch identifiers in increasing order, and we use an arbitrary but fixed bijection $\ell: \mathcal{B} \xrightarrow{\sim} [1..B]$ to associate batches to these identifiers. If we prefer, we can use $\tilde{g}: \mathcal{M} \sqcup \mathcal{B} \rightarrow [1..M+B]$ with $\tilde{g} = g \circ (\text{id} \sqcup \ell)$ that defines an analogous function on marbles and batches instead.

Grouping functions are more general than allocation functions in the sense that we can derive an allocation function from any grouping function but not vice versa. Given g , we can find a corresponding allocation function a by identifying the batch that most directly bounds any given marble under g . Define $g_{\mathcal{M}} = g \circ \text{inj}_1$ and $g_{\mathcal{B}} = g \circ \text{inj}_2 \circ \ell$ that, respectively, map marbles and batches to their slots in $[1..M+B]$. For a marble $m \in \mathcal{M}$, its batch $a(m)$ must satisfy (i) $g_{\mathcal{M}}(m) \leq g_{\mathcal{B}}(a(m))$, and (ii) $g_{\mathcal{B}}(a(m)) \leq g_{\mathcal{B}}(b)$ for any batch $b \in \mathcal{B}$ for which $g_{\mathcal{M}}(m) \leq g_{\mathcal{B}}(b)$. These conditions uniquely define a for every marble. For instance, in the above example we have $a(\text{1}) = 4$, $a(\text{2}) = 1$, $a(\text{3}) = 3$, $a(\text{4}) = 1$, $a(\text{5}) = 2$, $a(\text{6}) = 2$, and $a(\text{7}) = 2$.

With this framework in hand, we now turn to solving many problems in Table 18.1, and a few others as well.

18.4 Unique Marbles in Labeled Pouches

We start with the case where both individual marbles and individual batches are distinguishable, but where batches are stored without any regard to order or position. This is row 2 of Table 18.1, with uniquely colored marbles in labeled pouches. Each way of arranging marbles into batches in this row can be described by an allocation function $\mathcal{M} \rightarrow \mathcal{B}$.

Our goal then is to find the cardinality of the set $\mathcal{M} \rightarrow \mathcal{B}$ (unconstrained), the set $\mathcal{M} \rightarrow \mathcal{B}$ of injections (binary), the set $\mathcal{M} \twoheadrightarrow \mathcal{B}$ of surjections (nonempty), and the subset $\mathcal{M} \rightarrow \mathcal{B}$ with a specified profile.

We have already found the first of these, in the previous Section: by equation (18.10), the number of unconstrained allocation functions is B^M .

To construct an injection $\mathcal{M} \rightarrow \mathcal{B}$, we need to associate to each $m \in \mathcal{M}$ a *distinct* $b \in \mathcal{B}$. Put the marbles in any order $m_1, m_2, \dots, m_M$. We have B choices for the batch assigned to m_1 , then $B-1$ choices for the batch assigned to m_2 (necessarily excluding the batch assigned to m_1), $B-2$ choices for m_3 , and so on, until the choice for the batch assigned to m_M . The result is

$$B^{\underline{M}} = B(B-1)(B-2)\cdots(B-M+1), \quad (18.16)$$

which is called the M th **falling factorial power** of B .

(Following the Counting Principle, we could establish the count in the binary case by building an explicit bijection, but the extra complication is not more enlightening than the argument just given. If we were, one way to start this effort would be to apply equations (18.8) and (18.10) using

$$\#(\mathcal{M} \succrightarrow \mathcal{B}) = \# \bigsqcup_{b \in \mathcal{B}} (\mathcal{M} \succrightarrow \mathcal{B}_{-1}),$$

where $\mathcal{B}_{-1}$ is a subset of $\mathcal{B}$ that is missing one element.)

Observe that if $M > B$, we get $B^{\underline{M}} = 0$ injections between the larger $\mathcal{M}$ and the smaller $\mathcal{B}$. Put another way: if $M > B$, then every function $\mathcal{M} \rightarrow \mathcal{B}$ must map two marbles to the same bin. This is the famous **pigeonhole principle**.

If $M = B$, we get $B^{\underline{M}} = B!$. When we have the same number of marbles and batches, an injection from $\mathcal{M}$ to $\mathcal{B}$ is in fact a *bijection*. Taking $\mathcal{M} = \mathcal{B} = [1..B]$, the set of bijections $\mathcal{M} \succrightarrow \mathcal{B}$ equals the set of permutations of $[1..B]$. Hence, we have found that the **number of permutations of n items is $n!$** . More generally, The **number of sequences of k items selected from among $n \geq k$ items** is $n^{\underline{k}}$. These are a useful facts to have in our toolbox.

Next, we count the surjections $\mathcal{M} \twoheadrightarrow \mathcal{B}$, but on a moment's reflection, we can see that we have already done the work. Recall from Examples 14.1 and 16.3 every surjection $a: \mathcal{M} \twoheadrightarrow \mathcal{B}$ induces a partition of $\mathcal{M}$ with B parts $a^+(\{b\})$ for $b \in \mathcal{B}$. Similarly, from every partition of $\mathcal{M}$ into B parts, we get $B!$ different surjections; each surjection maps a part of $\mathcal{M}$ to an element of B and every permutation of $\mathcal{B}$ rearranges the mapping. Hence, we have a bijection between the set $\mathcal{M} \twoheadrightarrow \mathcal{B}$ and the set $\text{part}_B(\mathcal{M}) \times (\mathcal{B} \succrightarrow \mathcal{B})$, where the first term is the set of partitions of $\mathcal{M}$ with B parts and the second term is the set of permutations of $\mathcal{B}$. The cardinality $\#\text{part}_B(\mathcal{M})$ is denoted $\{^M_B\}$, so by equation (18.9) and the fact above about permutations, the number of surjections $\mathcal{M} \twoheadrightarrow \mathcal{B}$ is $\{^M_B\} B!$.

The numbers $\{^n_k\}$ are called the **Stirling subset numbers**, or more mundanely called the *Stirling numbers of the second kind*.⁸ The name and notation are supposed to evoke and reinforce each other, with the curly braces reminiscent of our set delimiters; the notation for the Stirling numbers is similar to that for Binomial coefficients, which were discussed in Section 18.2. The Stirling subset number $\{^n_k\}$ gives the count of the number of ways to partition a set of n items into k parts.

We take by convention that there is $\{^0_0\} = 1$ way to partition the empty set into zero parts. Then, $\{^n_0\} = 0$ for $n > 0$. On the other hand, $\{^0_1\} = 0$ as the empty set is, well, empty, but $\{^n_1\} = 1$ for $n > 0$, taking the whole set as the single

⁸Sadly, the latter soul-deadening name is far more common, but I suggest that you resist. It's neither memorable nor conceptual; only habit keeps it alive.

part. Similarly, $\left\{ \begin{smallmatrix} 0 \\ 2 \end{smallmatrix} \right\} = \left\{ \begin{smallmatrix} 1 \\ 2 \end{smallmatrix} \right\} = 0$ as there aren't enough elements for two parts, and $\left\{ \begin{smallmatrix} n \\ 2 \end{smallmatrix} \right\} = (2^{n-1} - 1)\{n > 0\}$. To see this, given a set with cardinality $n \geq 2$, fix a particular element x . In any partition of the set into two nonempty parts, one part contain x and some subset of the other $n - 1$ elements; the other part contains the remaining elements. That second part is a subset that can be chosen in $2^{n-1} - 1$ ways because it is drawn from $n - 1$ element but the empty set is excluded as the part must be nonempty.

In general, the Stirling subset numbers satisfy the following recurrence relation:

$$\left\{ \begin{smallmatrix} 0 \\ k \end{smallmatrix} \right\} = \{k = 0\} \quad (18.17)$$

$$\left\{ \begin{smallmatrix} n \\ 0 \end{smallmatrix} \right\} = \{n = 0\} \quad (18.18)$$

$$\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} = k \left\{ \begin{smallmatrix} n-1 \\ k \end{smallmatrix} \right\} + \left\{ \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right\}. \quad (18.19)$$

To see this for $n > 0$, let x be a fixed element of the set. Then any partition of the set into k blocks either has x alone in one part in $\left\{ \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right\}$ ways or puts x in one of the $\left\{ \begin{smallmatrix} n-1 \\ k \end{smallmatrix} \right\}$ parts, from partitioning the remaining elements, in k ways. This recurrence uniquely defines the subset numbers for all integers n and k .

As a side note, these Stirling numbers allow us to convert from falling factorial powers to ordinary powers:

$$x^n = \sum_k \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} x^k. \quad (18.20)$$

Puzzle 164. Make a table of Stirling subset numbers for small values of n and k , including negative values.

Finally, we consider the constraint that the allocation function satisfy a particular profile $\beta = \langle \beta_0, \beta_1, \dots, \beta_M \rangle$, where β_s is a specified number of batches of size s . How many allocation functions a have a specific profile β ? Here, we will derive the answer without going too deep into the details because the result is useful. We discussed the combinatorial interpretation of the multinomial coefficient $\binom{n}{c}$ earlier and will see it again in Section 18.6; it corresponds to the profile cell in row three of Table 18.1.

If β is a profile, let $\iota(\beta)$ be the $B - \beta_0$ tuple $\langle 1, \dots, 1, 2, \dots, 2, \dots, M, \dots, M \rangle$ with β_1 1's, then β_2 2's, and so on through β_M M 's. Notice that the sum β is $\sum_{s=0}^M \beta_s = B$ and the sum of $\iota(\beta)$ is $\sum_{s=0}^M s\beta_s = M$. The number of allocation functions a have a

specific profile β is then

$$\binom{B}{\beta} \binom{M}{\iota(\beta)}. \quad (18.21)$$

The first term is the number of ways to arrange elements of $\mathcal{B}$ into groups of sizes $\beta_0, \beta_1, \dots, \beta_M$.⁹ The second term is the number of ways to arrange elements of $\mathcal{M}$ into groups of sizes given by $\iota(\beta)$: β_1 groups of size 1, β_2 groups of size 2, and so on through β_M groups of size M . There is a bijection between the Cartesian product of these sets of groupings and the set of allocation functions matching the profile β .

⁹For this sentence, $\mathcal{B}$ is playing the role of marbles and the groups are the batches. (*Sound of head exploding*)

Table 18.2 summarizes our work in this Section. As special cases, we have found that

- The number of ways to allocate n items to m groups without restriction is m^n .
- The number of permutations of n items is $n!$.
- The number of sequences of k items selected from among $n \geq k$ items is $n^{\underline{k}}$.

Batch Identity	Marble Coloring	Batch Storage	Unconstrained Batches	Binary Batches	Nonempty Batches	Batch Profile β
Labeled	Unique	Pouches	B^M	$B^{\underline{M}}$	$\{^M_B\} B!$	$\binom{B}{\beta} \binom{M}{\iota(\beta)}$

TABLE 18.2. Row 2 of Table 18.1 filled in.

18.5 Unique Marbles in Labeled Tubes

We now consider the case where both marbles and batches have distinguished identities for our counts and in addition we pay attention to the order/position of marbles *within batches*. This is Row 1 of Table 18.1. For this purpose, we need to count grouping functions $g: \mathcal{M} \sqcup \mathcal{B} \rightarrow [1 \dots M + B]$ as described earlier.

For the unconstrained case, we can visualize a grouping function as depicted in Figure 18.2, with marbles laid out in order and separated by batch identifiers, where a batch identifier always maps to the last position $M + B$. The marbles most directly below a particular batch identifier are assigned to that batch, and the order they are listed gives their positions in the tube (with the opening on the left end, say).

To construct such a function, start all B batch identifiers lined up adjacent, and one marble at a time, we place the marble to the *left* of one of the line up objects (batch identifier or marble). When all marbles are used, this produces an bijective assignment of marbles and batch identifiers to $[1 \dots M + B]$ with one batch identifier on the right end, thus producing a valid grouping function. Consider the choices at each stage. We can place the marble in B positions, directly to the left of one of the

batch identifiers. We can place the second marble in $B + 1$ positions, directly to the left of a batch identifier *or* the first marble. And so on, we can place the k th marble in $B + k - 1$ positions, directly to the left of B batch identifiers or $k - 1$ marbles. By equation (18.9), the total number of choices is

$$B^{\overline{M}} = B(B + 1) \cdots (B + M - 1). \quad (18.22)$$

This is the M th **rising factorial power** of B . Each such choice gives a different grouping function, and every grouping function can be produced by one such series of choices. Equation (18.22) thus gives the total number of grouping functions and thus the entry in the Unconstrained cell.

The constraint of binary batches obviates the distinction between tubes and pouches as there is no ordering to consider, so we know that the Binary cell in Row 1 is the same as in Row 2, $B^{\overline{M}}$. We can see this directly in terms of grouping functions, just for fun. Consider the set of choices in the previous paragraph. Imposing the constraint that no batch can have size bigger than 1, once we have put a marble to the left of a batch identifier, we cannot choose that same batch identifier again. Thus we have B choices for the first marble, $B - 1$ for the second, and so on to $B - M + 1$.

The constraint to Nonempty batches means that we need to ensure that there is at least one marble *directly* to the left of each batch identifier. If $M < B$, this is impossible, so assume $M \geq B$. First, we assign B marbles with one in every batch, and then we add the remaining marbles freely. We know from the Binary case in Row 2 that the first step can be done in $M^{\overline{B}}$ ways. (Note the inversion of marbles and batches here. In the sub-problem of assigning each batch identifier to a marble, the batch identifiers play the role of “marbles” and the marbles play the role of binary “batches”.) We know from the calculation above in the Unconstrained case that the second step can be done in $B^{\overline{M-B}}$, using the allocating $M - B$ balls to B batches. Because a way of doing the first step is paired with a way of doing the second step,

the number of arrangements is thus the product $M^B B^{\overline{M-B}}$. Simplifying, we have

$$M^B = M(M-1) \cdots (M-B+1) = M(M-1)^{\underline{B-1}} \quad (18.23)$$

$$\begin{aligned} B^{\overline{M-B}} &= B(B+1) \cdots (B+(M-B)-1) = B(B+1) \cdots (M-1) = (M-1)^{\underline{B-1}} \\ &= \frac{(M-1)!}{(B-1)!} \end{aligned} \quad (18.24)$$

$$\begin{aligned} M^B B^{\overline{M-B}} &= M(M-1)^{\underline{B-1}} \frac{(M-1)!}{(B-1)!} \\ &= M! \frac{(M-1)^{\underline{B-1}}}{(B-1)!} \\ &= M! \binom{M-1}{B-1}. \end{aligned} \quad (18.25)$$

This gives $M! \binom{M-1}{B-1}$ suitable arrangements of marbles into batches.

Table 18.3 summarizes the work we’ve done in this Section. For the last column, we use the corresponding result for identically colored marbles, in the next Section. If there are n arrangements with batch profile β with identical marbles, each generates $M!$ arrangements in this case, or $nM!$ in total, because we consider all the orderings of the marbles in each identically colored arrangement.

Batch Identity	Marble Coloring	Batch Storage	Unconstrained Batches	Binary Batches	Nonempty Batches	Batch Profile β
Labeled	Unique	Tubes	$B^{\overline{M}}$	$B^{\underline{M}}$	$\binom{M-1}{B-1} M!$	$\binom{B}{\beta} M!$

TABLE 18.3. Row 1 of Table 18.1 filled in.

18.6 Identical Marbles, Labeled Batches

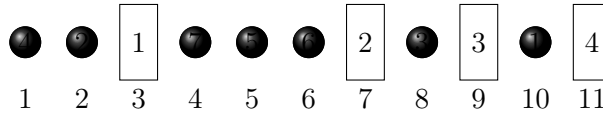
When the marbles are identical, we do not distinguish which marbles are in which batch or as a consequence, the order in which they are stored. Because we cannot distinguish among marbles, an arrangement is determined by *the number of marbles* in each batch. We can describe any such arrangement by a function $\mathcal{B} \rightarrow [0..M]$, that is a **multiset** with base set $\mathcal{B}$ with cardinality M .¹⁰ We denote the number of multisets of size M on a base set of cardinality B by the “multi-choose” coefficients $\left(\!\!\binom{B}{M}\!\!\right)$.

The unconstrained batches in Row 3 are in bijection with the multisets of size M , $\left(\!\!\binom{B}{M}\!\!\right)$. The binary batches are just multisets for which every element of the base

¹⁰See Example 10.14 and page 571.

set appears either 0 or 1 times, making them sets. So that cell counts the number of subsets of size M , $\binom{B}{M}$. A nonempty batch first allocates one marble to each batch then groups the remaining marbles among batches in any ways, giving $\left(\binom{B}{M-B}\right)$ possibilities. And for a specific profile β , the number of arrangements is the number of multisets that map β_0 batches to 0, β_1 batches to 1, and so on. But this is the number of ways to partition B into subsets of sizes $\beta_0, \beta_1, \dots, \beta_M$, which is the multinomial coefficient $\binom{B}{\beta}$.

This gives us answers for each cell of Row 3 without having found the multi-choose coefficients. Fortunately, we've already done the critical work in Row 1. Look again at the grouping function depicted in Figure 18.2. If we erase/obscure the identities of the balls, as follows



we get a representation of a multiset. The value for batch b is the number of marbles directly to the left of the rectangle for batch b . And observe that any grouping function that rearranges the order of the marbles in Figure 18.2 gives the *same multiset*! That is, each multiset is produced by $M!$ grouping functions. We thus obtain the results in Row 3 of Table 18.1 by dividing the entries in Row 1 by $M!$.

Batch Identity	Marble Coloring	Batch Storage	Unconstrained Batches	Binary Batches	Nonempty Batches	Batch Profile β
Labeled	Identical	—	$\left(\binom{B}{M}\right)$	$\binom{B}{M}$	$\binom{M-1}{B-1}$	$\binom{B}{\beta}$

TABLE 18.4. Row 3 of Table 18.1 filled in.

In particular, we have found that $\left(\binom{B}{M}\right) = B^{\overline{M}}/M!$ giving a nice pairing:

$$\binom{B}{M} = \frac{B^{\overline{M}}}{M!} \quad (18.26)$$

$$\left(\binom{B}{M}\right) = \frac{B^{\overline{M}}}{M!}. \quad (18.27)$$

Because $B^{\overline{M}} = B(B+1)\cdots(B+M-1) = (B+M-1)^{\overline{M}}$, we also have that

$$\left(\binom{B}{M}\right) = \binom{B+M-1}{M} \quad (18.28)$$

and thus

$$\left(\binom{B}{M-B}\right) = \binom{M-1}{M-B} = \binom{M-1}{B-1}. \quad (18.29)$$

This reconciles all the expressions we have derived for Row 3.

While this solves our problems, it's worth diving a little more deeply to understand the connection between Rows 1, 2, and 3. Suppose we have two grouping functions $g, g': \mathcal{M} \sqcup [1..B] \rightarrow [1..M+B]$. We say that $g \sim g'$ if there is a permutation $p: \mathcal{M} \rightarrow \mathcal{M}$ that equates them, i.e., that makes the following diagram commute

$$\begin{array}{ccc} \mathcal{M} \sqcup [1..B] & & \\ p \sqcup \text{id} \downarrow & \searrow g' & \\ \mathcal{M} \sqcup [1..B] & \xrightarrow{g} & [1..M+B] \end{array}$$

The function $p \sqcup \text{id}$ passes through as is any input from the $[1..B]$ part but permutes any input from the $\mathcal{M}$ part with p

The relation $\sim$ is reflexive ($g \sim g$); just take id as the permutation. It is also symmetric; just use the inverse permutation:

$$\begin{array}{ccc} \mathcal{M} \sqcup [1..B] & & \\ p^{-1} \sqcup \text{id} \uparrow & \searrow g' & \\ \mathcal{M} \sqcup [1..B] & \xrightarrow{g} & [1..M+B] \end{array}$$

It is transitive ($g' \sim g$ and $g \sim g''$ implies $g' \sim g''$) by simply composing the permutations and pasting the two diagrams together:

$$\begin{array}{ccc} \mathcal{M} \sqcup [1..B] & & \\ \downarrow p \sqcup \text{id} & \searrow g' & \\ (q \circ p) \sqcup \text{id} \downarrow \mathcal{M} \sqcup [1..B] & \xrightarrow{g} & [1..M+B] \\ \downarrow q \sqcup \text{id} & \searrow g'' & \\ \mathcal{M} \sqcup [1..B] & & \end{array}$$

It follows that $\sim$ is an equivalence relation (see page 684). The set of equivalence classes is then in one-to-one correspondence with the number of arrangements of identical marbles in labeled batches, and we want the cardinality of that set. Because each equivalence class contains $M!$ functions, we get the results in the table above.

It is worth noting that we can find the results for Row 3 from the results for Row 2 with a similar relation, using allocation functions. If $a, a': \mathcal{M} \rightarrow \mathcal{B}$ are allocation

functions (for uniquely colored marbles in labeled pouches), then a and a' represent the same allocation with *identically colored marbles* if there is a relabeling of the marbles that makes the two allocations equal. That is, there is a permutation p that makes the following diagram commute:

$$\begin{array}{ccc} \mathcal{M} & & \\ \downarrow p & \searrow a' & \\ \mathcal{M} & \xrightarrow{a} & \mathcal{B} \end{array}$$

This relation is also an equivalence relation with equivalence classes containing allocation functions for the problems in Row 2. Unlike the reasoning above, not all the equivalence classes have the same size as pouches already elide any distinction on order, but we can use equivalence classes of allocation functions to find Row 3 from Row 2 as well.

Puzzle 165. Give an argument to show that the above relation between allocation functions is an equivalence relation.

18.7 Identical Marbles, Unlabeled Batches

Row 6 adapts the results of the previous Section by erasing the batch labels, so we no longer distinguish a specific identity for batches. All the matters for distinguishing arrangements is the *pattern of counts* of marbles in batches. “Pattern of counts” might seem a bit vague, so we need to specify what this means.

If n is a natural number, a *partition* with k parts is a k -tuple $\lambda = \langle \lambda_1, \dots, \lambda_k \rangle$ with $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_k > 0$ and $\lambda_1 + \dots + \lambda_k = n$. For $n \in \mathbb{N}$ and $k \in [1..n]$, define $\mathbf{p}(n, k)$ to be the number of partitions of the natural number n into exactly k parts.¹¹ Then, $\mathbf{p}(n) = \sum_{k=1}^n \mathbf{p}(n, k)$ is the total number of partitions of n .

A “pattern of counts” with identical marbles and unlabeled batches is just a partition of the natural number M . For instance, if $M = 7$ and $B = 4$, then

$$\begin{array}{cccc} & & 7 & \\ & & & \\ & 6, 1 & 5, 2 & 4, 3 \\ 5, 1, 1 & 4, 2, 1 & 3, 3, 1 & 3, 2, 2 \\ & 4, 1, 1, 1 & 3, 2, 1, 1 & \end{array}$$

are all the possible patterns we can see in distinct arrangements. There are $\mathbf{p}(7, 1) + \mathbf{p}(7, 2) + \mathbf{p}(7, 3) + \mathbf{p}(7, 4) = 10$ such arrangements. If instead $B = 7$, we get patterns

¹¹Note that a partition of a natural number and a partition of a set are related but distinct notions.

7
 6, 1 5, 2 4, 3
 5, 1, 1 4, 2, 1 3, 3, 1 3, 2, 2
 4, 1, 1, 1 3, 2, 1, 1
 3, 1, 1, 1, 1 2, 2, 1, 1, 1
 2, 1, 1, 1, 1, 1
 1, 1, 1, 1, 1, 1, 1

giving 14 possible arrangements. Again, observe that we do not distinguish among the marbles in the batches nor among the batches of the same size.

The number of partitions of a natural number has no nice closed form, but it can be computed via a recurrence relation or accurately approximated. For instance, [HardyRamanujan1918] showed that

$$\mathfrak{p}(n) \asymp \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2}{3}n}},$$

which means that the ratio of the two sides approaches 1 as n grows larger.

These “partition numbers” determine all the entries in Row 6. The number of arrangements with B unconstrained batches is just the number of partitions with at most B parts. (If there are fewer than B parts, the remaining batches are empty.) This gives $\sum_{k=1}^B \mathfrak{p}(M, k)$ arrangements. There is an arrangement with B binary batches only if the partition of M with all 1’s fits in B parts. Thus, the indicator $\{B \geq M\}$ counts the number of such batches. An arrangement with B nonempty batches must have exactly B parts, so we have $\mathfrak{p}(M, B)$ such arrangements. And finally, for any profile β , we know that $\sum_{s=1}^M s\beta_s = M$; ordering the $s\beta_s$ give single partition of M – and thus one arrangement.

Batch Identity	Marble Coloring	Batch Storage	Unconstrained Batches	Binary Batches	Nonempty Batches	Batch Profile β
Unlabeled	Identical	–	$\sum_{k=1}^B \mathfrak{p}(M, k)$	$\{B \geq M\}$	$\mathfrak{p}(M, B)$	1

TABLE 18.5. Row 6 of Table 18.1 filled in.

18.8 Unique Marbles in Unlabeled Pouches

If the marbles are uniquely colored but the batches are both unordered (pouches) and unlabeled, then our idea of “pattern” changes from the previous Section. Rather than

simply paying attention to the *number* of marbles in each batch, we now care about the *subset* of marbles in each batch. Thus, we move from partitions of a natural number to partitions of a set as the organizing idea in Row 5. The structure of Row 5 mimics that of Row 6 in this sense.

Again, we can think of each such arrangement as an equivalence class of functions. Let $a, a': \mathcal{M} \rightarrow \mathcal{B}$ be allocation functions (for uniquely colored marbles in labeled pouches). We declare a and a' equivalent – representing the same arrangement with *unlabeled pouches* – if there is a relabeling of the pouches that makes the two allocations equal. That is, there is a permutation p that makes the following diagram commute:

$$\begin{array}{ccc} \mathcal{M} & \xrightarrow{a} & \mathcal{B} \\ & \searrow a' & \downarrow p \\ & & \mathcal{B} \end{array}$$

meaning that $a' = p \circ a$. Define $a \sim a'$ in this case; this relation is an equivalence relation. The equivalence class of an allocation function a consists of all functions where the set (of sets) $\{a^{-1}(\{b\}) \mid b \in \mathcal{B}\}$ are the same. Each such set is a partition of $\mathcal{M}$,¹² and different partitions of M are in different equivalence classes!

¹²See Examples 14.1 and 16.3.

Thus, the count of arrangements with unconstrained batches is the number of partitions into at most B parts: $\sum_{k=1}^B \{M_k\}$. The number arrangements with nonempty batches is the number of partitions with exactly B parts: $\{M_B\}$. An arrangement with binary batches only exists if $B \geq M$ by the *pigeonhole principle* described above, so the count is given by the indicator $\{B \geq M\}$. And for a specific batch profile β , $\binom{M}{\iota(\beta)}$ is the number of ways to divide $\mathcal{M}$ into subsets with sizes $1, 1, \dots, 1, 2, 2, \dots, 2, \dots, M, \dots, M$ (with respective counts β_s). But because the batches are unlabeled, we do not distinguish the $\beta_1!$ orderings of the size 1 sets, $\beta_2!$ orderings of the size 2 sets, and so on. The resulting count is $\frac{1}{\beta_1! \dots \beta_M!} \binom{M}{\iota(\beta)}$.

Batch Identity	Marble Coloring	Batch Storage	Unconstrained Batches	Binary Batches	Nonempty Batches	Batch Profile β
Unlabeled	Unique	Pouches	$\sum_{k=1}^B \{M_k\}$	$\{B \geq M\}$	$\{M_B\}$	$\frac{1}{\beta_1! \dots \beta_M!} \binom{M}{\iota(\beta)}$

TABLE 18.6. Row 5 of Table 18.1 filled in.

18.9 Unique Marbles in Unlabeled Tubes

Finally, we consider uniquely colored marbles stored in order in *unlabeled* tubes, Row 4. We can derive all of these solutions quickly from Row 1 by simply erasing the labels to create equivalence classes of grouping functions. This follows the pattern of the previous Sections, so we will keep it quick and informal.

First, there is an arrangement with binary batches only if there are at least as many batches as marbles. If there are, there is exactly one arrangement because we do not distinguish batch identities and a batch with only marble is identified with that marble. Hence, the binary cell in Row 4 is $\{B \geq M\}$.

Second, any arrangement with nonempty batches is produced by exactly $B!$ batches with *labeled* tubes by simply erasing the batch labels (which can occur in any “order”). Hence, we simply divide the corresponding cell in Row 1 by $B!$: $\frac{M!}{B!} \binom{M-1}{B-1}$. The same reasoning applies for a specific batch profile: $\frac{M!}{\beta_1! \dots \beta_M!}$.

Finally, the number of arrangements with unconstrained batches is just the number of arrangements with k nonempty batches for each $k \in [1..B]$. Using the previous result, we get $\sum_{k=1}^B \frac{M!}{k!} \binom{M-1}{k-1}$.

Notice the similarity in structure of corresponding cells in Rows 4-6. In each case, there is an organizing quantity (e.g., partitions of a natural number or partitions of a set) that gets used in the same way in particular columns.

The result for this Section and for all our calculations in the framework are given in Table [18.7](#).

Batch Identity	Marble Coloring	Batch Storage	Unconstrained Batches	Binary Batches	Nonempty Batches	Batch Profile β
Labeled	Unique	Tubes	$B^{\overline{M}}$	$B^{\underline{M}}$	$\binom{M-1}{B-1} M!$	$\binom{B}{\beta} M!$
	Unique	Pouches	B^M	$B^{\underline{M}}$	$\left\{ \begin{matrix} M \\ B \end{matrix} \right\} B!$	$\binom{B}{\beta} \binom{M}{\iota(\beta)}$
	Identical	–	$\left(\left(\begin{matrix} B \\ M \end{matrix} \right) \right)$	$\binom{B}{M}$	$\binom{M-1}{B-1}$	$\binom{B}{\beta}$
Unlabeled	Unique	Tubes	$\sum_{k=1}^B \frac{M!}{k!} \binom{M-1}{k-1}$	$\{B \geq M\}$	$\frac{M!}{B!} \binom{M-1}{B-1}$	$\frac{M!}{\beta_1! \cdots \beta_M!}$
	Unique	Pouches	$\sum_{k=1}^B \left\{ \begin{matrix} M \\ k \end{matrix} \right\}$	$\{B \geq M\}$	$\left\{ \begin{matrix} M \\ B \end{matrix} \right\}$	$\frac{1}{\beta_1! \cdots \beta_M!} \binom{M}{\iota(\beta)}$
	Identical	–	$\sum_{k=1}^B \mathfrak{p}(M, k)$	$\{B \geq M\}$	$\mathfrak{p}(M, B)$	1

Common Cases

- Number of ways to make M independent choices of B items: B^M .
- Number of distinct sequences of length M from B items: $B^{\underline{M}}$.
- Number of subsets of size M from a set of B items: $\binom{B}{M}$.
- Number of multisets of size M from a base set of size B : $\left(\left(\begin{matrix} B \\ M \end{matrix} \right) \right)$.
- Number of partitions of a set of size M into B parts: $\left\{ \begin{matrix} M \\ B \end{matrix} \right\}$.

TABLE 18.7. Solutions to the common counting problems in the framework of Section 18.3.

18.10 Permutations and Cycles

How many ways are there to sit n people around k circular tables so that no table is empty? Here, we treat the people as uniquely colored marbles and the tables as unlabeled batches. However, unlike the problems in Table 18.7, the batches are *cyclically ordered*, not in marbles in tubes but rather marbles threaded onto a loop of string. So,  and  represent the same cycle because the order is the same when we allow for *wrapping around from end to beginning*, but both differ from .

We can view this problem in terms of permutations. On page 674, we saw that every permutation of n items can be written as the composition of disjoint cycles. If we think of the people at each table as a cycle, the seating arrangement in our problem corresponds to a collection of k cycles that make up a permutation.

The number of permutations of n items that are composed of k cycles is called the **Stirling cycle number**,¹³ denoted by $\left[\begin{smallmatrix} n \\ k \end{smallmatrix} \right]$. We have $\left[\begin{smallmatrix} n \\ 0 \end{smallmatrix} \right] = \{n = 0\}$ with the 0 case by convention. By direct calculation, we have for $n \geq 1$

¹³This is more commonly called the Stirling number of the *first* kind. See the earlier comment on soul-deadening names related to the other Stirling numbers.

$$\left[\begin{smallmatrix} n \\ n \end{smallmatrix} \right] = 1 \quad (18.30)$$

$$\left[\begin{smallmatrix} n \\ n-1 \end{smallmatrix} \right] = \binom{n}{2} \quad (18.31)$$

$$\left[\begin{smallmatrix} n \\ 1 \end{smallmatrix} \right] = (n-1)!. \quad (18.32)$$

The identity permutation has n cycles leaving each element unchanged. With $n-1$ cycles, there are two values that appear in the same cycle, and the order does not matter. The number of such permutations is the number of ways to choose that subset of two values. And finally, if we have one cycle, we can start it with the first item n , and the rest of the cycle is a choice of an ordering in the cycle of the remaining $n-1$ items.

Because every permutation can be decomposed into cycles, we have that

$$\sum_{k=1}^n \left[\begin{smallmatrix} n \\ k \end{smallmatrix} \right] = n!. \quad (18.33)$$

The Stirling cycle numbers satisfy a recurrence relation for $n \geq k \geq 1$:

$$\left[\begin{smallmatrix} n \\ k \end{smallmatrix} \right] = \left[\begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right] + (n-1) \left[\begin{smallmatrix} n-1 \\ k \end{smallmatrix} \right]. \quad (18.34)$$

We can establish this by the Counting Principle. Informally, consider a permutation of n items with k cycles. Either n belongs to its own cycle or not. If so, we can specify the permutation by choosing a permutation of $n - 1$ items with $k - 1$ cycles in $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$ ways. If not, we can specify the permutation by choosing a permutation of $n - 1$ items with k cycles in $\begin{bmatrix} n-1 \\ k \end{bmatrix}$ ways and place n to the right of any of the $n - 1$ items, in $n - 1$ ways. If $\mathcal{P}_{n,k}$ is the set of permutations of n items with k cycles, then we have just described a bijection between $\mathcal{P}_{n,k}$ and $\mathcal{P}_{n-1,k-1} \sqcup ([1 \dots n-1] \times \mathcal{P}_{n-1,k})$.

18.11 The Inclusion-Exclusion Principle

Let $\mathcal{S}$ be a set with cardinality n and suppose $f: 2^{\mathcal{S}} \rightarrow \mathbb{R}$. Define

$$g(\mathcal{A}) = \sum_{\mathcal{B} \supseteq \mathcal{A}} f(\mathcal{B}), \quad \mathcal{A} \subseteq \mathcal{S}. \quad (18.35)$$

If we are given the function g , we can recover f :

$$f(\mathcal{A}) = \sum_{\mathcal{B} \supseteq \mathcal{A}} (-1)^{\#(\mathcal{B}-\mathcal{A})} g(\mathcal{B}), \quad \mathcal{A} \subseteq \mathcal{S}. \quad (18.36)$$

This invertible relationship between f and g is called the **principle of inclusion-exclusion**.

A common use case for this principle is where $\mathcal{S}$ is a set of properties that objects in a set $\mathcal{X}$ might have and the function $f(\mathcal{A})$ counts the number of objects in $\mathcal{X}$ that have *exactly* the properties in $\mathcal{S}$. Then $g(\mathcal{A})$ counts the number of objects in $\mathcal{X}$ that have *at least* the properties in $\mathcal{A}$.

Example 18.1. How many permutations of n items leave none of the items fixed? Call this number D_n .

Let the properties in $\mathcal{S} = [1 \dots n]$ be the indices of a fixed point; that is, permutation p has property $i \in \mathcal{S}$ if $p(i) = i$. D_n is the number of permutations that have *none* of these properties.

Using f and g from the above discussion, we want $f(\{\})$ and equation (18.36) gives

$$f(\{\}) = \sum_{\mathcal{B}} (-1)^{\#\mathcal{B}} g(\mathcal{B}).$$

$g(\mathcal{B})$ is the number of permutations which fix at least the values in $\mathcal{B}$. If $\#\mathcal{B} = k$, then $g(\mathcal{B}) = (n - k)!$ because we fix the k points and allow an arbitrary

permutation of the other $n - k$ points, which can include additional fixed points.

Hence,

$$\begin{aligned}
 D_n &= \sum_{\mathcal{B}} (-1)^{\#\mathcal{B}} g(\mathcal{B}) \\
 &= \sum_{k=0}^n \binom{n}{k} (-1)^k (n-k)! \\
 &= \sum_{k=0}^n \binom{n}{n-k} (-1)^{n-k} k! \\
 &= \sum_{k=0}^n \binom{n}{k} (-1)^{n-k} k! \\
 &= \sum_{k=0}^n (-1)^{n-k} n^{\underline{k}}.
 \end{aligned}$$

Part II

Distributions

In Part I, we developed a powerful conceptual and computational toolbox, including a grammar for building, describing, and analyzing finite random systems. The basic ingredients are

- Random quantities that we can measure/observe as a random system evolves, represented by **(conditional) FRPs**;
- Our *knowledge* about a random quantity, embodying our predictions about its value, represented by the **(conditional) Kind** of the corresponding FRP;
- Three operations (transformation with statistics, joins, constraining with observations) on FRPs and Kinds that produce new FRPs and Kinds.
- One operation (expectation) on FRPs and Kinds that extracts our predictions of an FRP's value.

This is coupled with the physical metaphor of FRPs that lets us reason about relationships, information flow through the system, and the act of observation.

This last point is critical. The values of random quantities are typically *unknown to us*, at least at the point when we have to make decisions or take action. We might observe some partial information about various quantities as the system evolves, but in the end, we use *predicted* answers to our questions to guide our actions.

So, most of the FRPs we work with our *fresh*, and our operations on them are expressed without knowing their value.

1. We express questions, change representations, extract information, and summarize outcomes by transforming with **statistics**.
2. We update our knowledge with new information, using partial observations of some FRPs, by constraining with **observations**.
3. We combine parts of the system into a larger system with **joins**, capturing interactions and dependence among the parts.
4. We compute our predictions of an FRP's value with **expectation**.

Each of the first three operations (the 3 in the Big 3+1) produce a new FRP from the original, transformed or constrained or combined with a new source of randomness. So for example, if we have two independent, scalar FRPs X and Y , we observe that $Y - X$ has the value 4, and we want to predict their sum, we can express this as

$$\mathbb{E}(\text{Sum}(X \star Y) \mid Y - X = 4),$$

or in `frplib` `E(Sum(X * Y) | Diff == 4)`. This expresses how the system is built (an independent join), what partial information we have (that the difference is 4), the

question we want to answer (what is the sum?), and gives us the predicted answer to that question.

Our operations give us a grammar for describing the system and our analysis. But without the values of the FRPs how do we proceed? That is the role of the FRPs' Kinds, which describe the nature of the FRPs. Whereas each FRP (eventually) has a single value, its Kind represents *all* its possible values simultaneously, with weights giving the likelihood on each value. Every operation on FRPs has a coherent parallel operation on Kinds operating on all those values and their weights at once. Transforming an FRP with a statistic produces a new FRP whose value is the result of the statistic applied to the original FRP's value. Transforming a Kind, we apply the statistic to every value and combine branches in the tree accordingly. Constructing a join wires the output of a FRP to trigger activate other FRPs (packaged in a conditional FRP). The join of Kind reflects this by adjoining trees in a way that matches the information flow in the wiring diagram. Similarly, constraining with an observation filters out the branches of a Kind that are inconsistent with the constraint, and the expectation takes a average of all the values using the weights.

Figure 18.3 illustrates the basic schema. Our assumptions about the system comprise our model. The model specifies the Kinds of the component FRPs, and the components are combined (via join and transformation) to produce the Data FRP that describes all relevant measurements/observations we might make. From the data, we can derive (via transformation and constraint) feature FRPs whose values represent answers to our questions. And finally, using all the available information, we predict the answers to our questions to guide our decisions and actions.

With these tools and ideas in hand, we can describe and analyze any finite system and even handle some infinities. This is significant for many problems sufficient. Nonetheless, for reasons we will explore, it is worthwhile to generalize to systems that are *infinite* in various senses. The good news is that the same operations, the same grammar, and the same core concepts carry over as is. What we will need are some new ways to express the structure of Kinds and their infinite generalizations, some new interfaces on the same information.

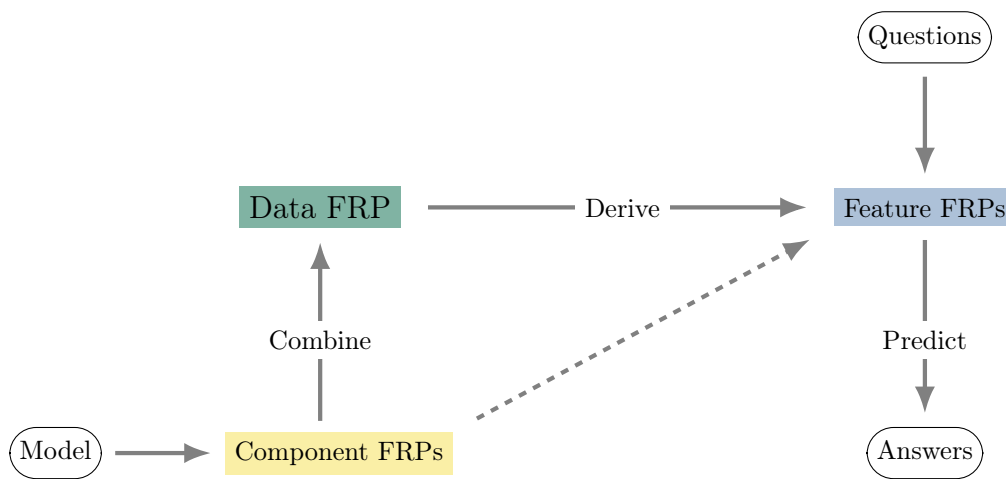


FIGURE 18.3. The basic schema for our description and analysis of a random system, reproducing Figure 2.8 in Part I. Our model specifies the component FRPs that are combined to build the data for the system (via join and transformation). From our motivating questions, we derive feature FRPs by transformation and constraint. We then predict the answers to those questions by taking expectations. The dashed arrow indicates that some feature FRPs may be defined in terms of selected components and thus may be activated before the Data FRP is.

Kinds, Kernels, and Distributions

19

Chapter

Contents

19.1 The Kernel	833
19.2 The Distribution Operator	837
19.3 Converting Among Kind, Kernel, and Distribution	842
19.4 Conditional Kinds and Conditional FRPs	845
19.5 Transforming with Statistics	851
19.6 Building with Joins	854
19.7 Constraining with Observations	860
19.8 Predicting with Expectations	863

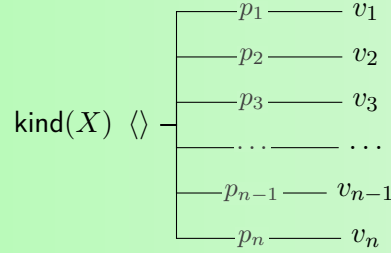
Key Take Aways

The Kind of an FRP embodies all of our knowledge and predictions about the FRP's value. We can express the same information with two objects derived from the Kind: the **kernel** and the **distribution operator** (or distribution for short).

If X is an FRP with Kind $\text{kind}(X)$, we denote the Kind's Kernel by K_X and its Distribution by D_X . The Kernel is a function that takes values and returns the canonical weights of those values. The Distribution is an operator that takes a question about X – in the form of a compatible statistic – and returns the predicted answer to that question. Specifically, $K_X(v)$ is the canonical weight on v in $\text{kind}(X)$, or 0 if v is not a possible value of X . And $D_X(\psi) = \mathbb{E}(\psi(X))$ for each compatible statistic ψ .

We name quantities associated with various FRPs and Kinds with a sans-serif letter to specify what type of quantity it is (e.g., K for Kernel, D for Distribution) with a subscript that indicates the (conditional) FRP or Kind it is associated with. We will see use this convention extensively throughout.

The relationships among the Kind, Kernel, and Distribution are easily expressed. Suppose that $\text{kind}(X)$ is



$$D_X(\psi) = \sum_{i=1}^n p_i \psi(v_i)$$

$$K_X(v) = \sum_{i=1}^n p_i \{v = v_i\}$$

$$D_X(\psi) = \sum_v K_X(v) \psi(v)$$

$$K_X(v) = D_X(\{\text{■} = v\})$$

The Kernel and Distribution can be computed for any (conditional) Kind or (conditional) FRP, including those created from various combinations of the Big 3. We use a consistent **naming convention** where we label the Kernel and Distribution to indicate what Kinds they are the Kernel and Distribution for.

For conditional Kinds and FRPs, we compute the Kernel and Distribution of the target for each input value. These give us conditional Kernels (e.g., $K_{Y|X}(v | u)$) and conditional Distributions (e.g., $D_{Y|X}(\psi | u)$). For each possible value u of the input FRP X , we get a Kernel function or a Distribution. The input is placed after the “given” bar as an auxilliary argument.

We can express all of the Big 3+1 in terms of Kernels and Distributions:

1. Transforming with Statistics

$$K_{\varphi(X)}(v) = \sum_u K_X(u) \{\varphi(u) = v\} \quad (19.1)$$

$$D_{\varphi(X)}(\psi) = D_X(\psi \circ \varphi) \quad (19.2)$$

2. Constraining with Observations

$$K_{X|\xi}(v) = \frac{K_X(v) \xi(v)}{\sum_u K_X(u) \xi(u)} \quad (19.3)$$

$$D_{X|\xi}(\psi) = \frac{D_X(\psi\xi)}{D_X(\xi)} \quad (19.4)$$

Building with Joins

$$K_{XY}(u, v) = K_X(u) K_{Y|X}(v | u) \quad (19.5)$$

$$= K_Y(v) K_{X|Y}(u | v) \quad (19.6)$$

$$D_{XY}(\psi) = D_X(\langle u \rangle \mapsto D_{Y|X}(\psi(u, \blacksquare) | u)) \cdot = D_Y(\langle v \rangle \mapsto D_{X|Y}(\psi(\blacksquare, v) | v)) \cdot \quad (19.7)$$

Predicting with Expectations

$$D_X(\psi) = \mathbb{E}(\psi(X)) = \sum_v K_X(v) \psi(v) \quad (19.8)$$

In this Chapter, we take a closer look at Kinds and the information they contain. A Kind is a tree with the possible values of an FRP at the leaves and positive weights on the branches. In canonical form, the tree has width 1 and the weights sum to 1. The Kind expresses our knowledge about the value of an FRP; the weights tell us how likely each value is to be observed; and from the Kind we can predict the answer to any reasonable question about the value of the FRP, i.e., can compute the expectation of *any transform* of the FRP.

In `frplib` and Chapter 6, we briefly considered two objects derived from a Kind that give us access to all the information the Kind contains: its *kernel* and its *distribution operator* (distribution for short). We think of these as alternate interfaces to the knowledge embodied by the Kind. With either the Kernel or the Distribution, we can reconstruct the Kind, and we can express all of the Big 3+1 operations. Here, we take a deeper look at these objects and how to use them.

19.1 The Kernel

In `frplib`, with a Kind `k` and a value `v`, we can find the weight associated with that value, if any, with `k.kernel(v)`. This returns the *canonical weight* on the `v` branch in the Kind, or 0 if there is no such branch. When the tree is in canonical form, this

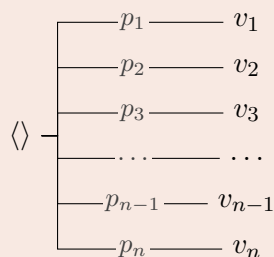
function `k.kernel` packages all the information in tree. Indeed, the function and canonical tree are *isomorphic* – each can be constructed from the other.¹

In general, the **Kernel** associated with a Kind is a function that takes a potential value and returns the canonical weight of that value in the Kind, or zero if the value is not possible.

If X is an FRP, we denote the Kernel associated with $\text{kind}(X)$ by K_X . The sans-serif K will always denote a Kernel, and the subscript is a *label* that tells us what this Kernel is associated with. In practice, we typically use the FRPs for the labels because it is specific and will be more convenient in practice, though for an arbitrary Kind denoted by a , we could just as well write K_a for its Kernel. This is a special case of our general labeling convention that we will discuss below in detail.

¹If k and j are *equivalent* Kinds, `k.kernel` and `j.kernel` are the *same function*, so the function only determines the equivalence class of Kinds.

Definition 46. If X is an FRP whose Kind is



with values $v_1, \dots, v_n$ and canonical weights $p_1, \dots, p_n$, then its **kernel** is the function K_X given by

$$K_X(v) = \sum_{i=1}^n p_i \{v = v_i\}. \quad (19.9)$$

We thus have $K_X(v_i) = p_i$ and $K_X(v) = 0$ if $v \notin \{v_1, \dots, v_n\}$.

If useful, we can also label a Kernel by its Kind. For instance, if the Kind above were named a , we would write the Kernel K_a . The point of the label is to tell us what the Kernel belongs to.

We interchangeably call K_X the Kernel of X and the Kernel of $\text{kind}(X)$, as we prefer.

Example 19.1. Let X be an FRP defined by

```
pgd> X = frp( weighted_by(1, 2, ..., 12, weight_by=lambda j: abs(j - 4)) )
pgd> kind(X)
,---- 3/42 ---- 1
|---- 2/42 ---- 2
```

	----	1/42	----	3
	----	1/42	----	5
	----	2/42	----	6
<>	-+----	3/42	----	7
	----	4/42	----	8
	----	5/42	----	9
	----	6/42	----	10
	----	7/42	----	11
	`----	8/42	----	12

Then,

$$K_X(v) = \frac{\sum_{j=1}^{12} |j-4| \{v=j\}}{\sum_{j=1}^{12} |j-4|} = \frac{1}{42} \sum_{j=1}^{12} |j-4| \{v=j\}.$$

The denominator in the first equation is just the sum of the original weights, converting the weights into canonical form. This factor does not affect the results when v is not a possible value, as 0 is returned.

Example 19.2. An FRP Y has kernel

$$K_Y(v) = \frac{1}{6} \{v = \langle -1, -1 \rangle\} + \frac{1}{12} \{v = \langle -1, 1 \rangle\} + \frac{1}{4} \{v = \langle 0, 0 \rangle\} + \frac{1}{9} \{v = \langle 1, -1 \rangle\} + \frac{7}{18} \{v = \langle 1, 1 \rangle\}.$$

What is its Kind?

We can read off both the possible values and their canonical weights, yielding

$$\langle \rangle \begin{cases} \frac{1}{6} & \langle -1, -1 \rangle \\ \frac{1}{12} & \langle -1, 1 \rangle \\ \frac{1}{4} & \langle 0, 0 \rangle \\ \frac{1}{9} & \langle 1, -1 \rangle \\ \frac{7}{18} & \langle 1, 1 \rangle \end{cases}$$

Example 19.3. A Kind K is given by

$$\langle \rangle \begin{cases} 10 & \langle -8, -8, -8 \rangle \\ 20 & \langle 0, 0, 0 \rangle \\ 40 & \langle 8, 8, 8 \rangle \end{cases}$$

What is its Kernel?

Remember that the Kernel always gives the *canonical* weights. So, we renormalize the weights and get

$$\mathsf{K}_K(v) = \frac{1}{7} \{v = \langle -8, -8, -8 \rangle\} + \frac{2}{7} \{v = \langle 0, 0, 0 \rangle\} + \frac{4}{7} \{v = \langle 8, 8, 8 \rangle\}$$

Using the same letter for the base function and the label looks odd at first, but a label is just a label. And observe how the base K is typeset differently.

The pattern language of our Big 3+1 operations let us generate new FRPs and Kinds from old ones. So for instance, if Y is an FRP, ψ is a compatible statistic, and ξ is a compatible condition, then we can form the FRPs $\psi(Y)$ and $Y \mid \xi$, which have Kinds $\psi(\text{kind}(Y))$ and $\text{kind}(Y) \mid \xi$, respectively. We can find the Kernels of these Kinds just as easily, and importantly, our *labels* reflect the source, e.g., $\mathsf{K}_{\psi(X)}$ and $\mathsf{K}_{Y \mid \xi}$.

Example 19.4. Let the FRP P represent a randomly chosen point uniformly from the grid $[0..4] \times [0..4]$. Define FRPs $X = \text{proj}_1(P)$ and $Y = \text{proj}_2(P)$ to represent the horizontal and vertical coordinates of the point.

The Kernel of P is

$$\mathsf{K}_P(x, y) = \frac{1}{25} [0..4](x) [0..4](y).$$

The indicators ensure that $\mathsf{K}_P(x, y)$ only return a non-zero value when $\langle x, y \rangle \in [0..4] \times [0..4]$, and all such values have weight $1/25$, which is what it means for the Kind to be uniform. We could write this as a sum as well:

$$\mathsf{K}_P(x, y) = \sum_{i=0}^4 \sum_{j=0}^4 \frac{1}{25} \{\langle x, y \rangle = \langle i, j \rangle\}.$$

This is in fact an independent join, and we will see an even easier way to derive this in a little while.

Let $\psi = \text{Sum}$ and define $\xi(x, y) = \{x + y = 4\}$. We can compute the Kinds of $\psi(P)$ and $P \mid \xi$, either using `frplib` or manually using our understanding of these operations from Part I. In fact, these operations give us the Kernels of these Kinds in terms of K_P , another important point that we will return to later.

See Section 9.3 in Interlude F for more on Cartesian products.

For now, our goal is to see how we write and label these Kernels:

$$K_{\text{Sum}(P)}(s) = \sum_{j=0}^8 \frac{\min(j+1, 9-j)}{25} s = j \quad (19.10)$$

$$K_{P|\xi}(x, y) = \sum_{i=0}^4 \frac{1}{5} \langle x, y \rangle = \langle i, 4-i \rangle. \quad (19.11)$$

Notice that $\xi(P)$ is exactly the event $\{\text{Sum}(P) = 4\} = \{X + Y = 4\}$. As we have seen, we like to avoid making up names for things unnecessarily, so we can express this condition in terms of the event, writing $P \mid \text{Sum}(P) = 4$ or $P \mid X + Y = 4$, as we prefer. So, the second Kernel above can be written as

$$K_{P|\xi}(x, y) = K_{P|\text{Sum}(P)=4}(x, y) = K_{P|X+Y=4}(x, y). \quad (19.12)$$

Recall that the braces around the event are optional, and typically dropped, after the given bar.

This is useful and good, but we can take it one step further. With complicated events, such labels can get awkward to read and write, especially when consider multiple related events that differ only in some details. So instead, we exploit the connection we saw in Chapter 5 between observations and conditional Kinds and FRPs. We thus write

$$K_{P|\text{Sum}(P)}(x, y \mid 4) = K_{P|X+Y}(x, y \mid 4), \quad (19.13)$$

where the *argument* to the Kernel after the given bar is the *given value* of the FRP after the given bar in the label. We will connect this to conditional Kinds and FRPs below, a third important point that we defer briefly in this example.

19.2 The Distribution Operator

We saw in Section 6.3 that two FRPs X and Y have the same Kind if and only if they have the same possible values and $\mathbb{E}(\psi(X)) = \mathbb{E}(\psi(Y))$ for *every compatible statistic* ψ . Because of this, we can view the Kind as embodying our knowledge of the FRP's value because it determines and is determined by *our predicted answers to every valid question about that value*.

This suggests that we can express the information in the Kind by packaging all these predicted answers (expectations). That is what the Distribution operator does, as we have seen in `frplib`.

```
pgd> X = frp( uniform(1, 2, ..., 6) )
```

```

pgd> D_X = D_(X)
pgd> D_X(Id)
7/2
pgd> D_X(__ ** 2)
91/6
pgd> D_X(__ - 3.5)
0
pgd> D_X(__ == 2)
1/6
pgd> D_X(__ <= 3)
1/2

```

In each case, we give D_X a *statistic*, and it returns the expectation of the transform of X by that statistic.

We think of the D_X as a sort of “oracle.” We bring the oracle a question about the value of X , as embodied in a compatible statistic ψ , and the oracle gives us a *predicted* answer $\mathbb{E}(\psi(X))$. Note for instance that one such question might be “Does X have the value 2?” for which we would use the condition $\psi(x) = \{x = 2\}$ or $(\_ == 2)$ in `frplib`. In this way, we can recover the weights for every value and thus the Kind itself.

For example, $D_X(\text{id}) = \mathbb{E}(\text{id}(X)) = \mathbb{E}(X)$, yielding the expectation of X . For any value v , $D_X(\_\text{■} = v) = \mathbb{E}(\{X = v\}) = K_X(v)$, the probability of the event that X has value v . Thus we can easily recover the Kernel of X from D_X by passing in indicators. A few further examples:

- $D_X(\text{const}_c) = \mathbb{E}(\text{const}_c(X)) = \mathbb{E}(c) = c$, in particular $D_X(\text{const}_1) = 1$.
- $D_X(a \leq \_\text{■} \leq b) = \mathbb{E}(\{a \leq X \leq b\})$, the probability of the event that X ’s value lies in the interval $[a\_b]$.
- $D_X(\text{Sum}) = \mathbb{E}(\text{Sum}(X)) = \mathbb{E}(X_1) + \mathbb{E}(X_2) + \cdots + \mathbb{E}(X_d)$, the expected sum of X ’s components, where X has dimension d .
- $D_X(\langle u \rangle \mapsto |u|) = \mathbb{E}(|X|)$, the expectation of the transformed FRP whose value is that absolute value of X ’s value.²
- $D_X(\langle v \rangle \mapsto |v - \mathbb{E}(X)|^2) = \mathbb{E}((X - \mathbb{E}(X))^2) = \text{Var}(X)$, the *variance* of X , a measure of uncertainty in X defined in Example 6.4 that we will see again.
- $D_X(-\lg K_X) = \mathbb{H}(X)$, the *entropy* of X , a measure of uncertainty in X defined in Example 6.4 that we will see again.

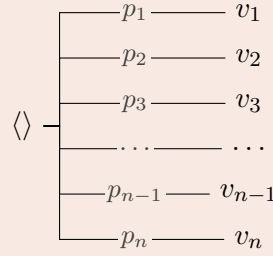
²Note that we pass the longer form of anonymous function here; we could also have used $D_X(|\_\text{■}|)$.

In all these cases, we hand the oracle a question (“Is X between a and b ?” or “How far (in squared units) is X from our prediction $\mathbb{E}(X)$?”), and the oracle hands us back a predict answer (the probability that X lies between a and b or the expected distance in squared units between X ’s value and $\mathbb{E}(X)$).

Definition 47. If X is an FRP, its **Distribution operator** D_X gives the **predicted answer to any valid question about X ’s value**. For any compatible statistic ψ , we define

$$D_X(\psi) = \mathbb{E}(\psi(X)). \quad (19.14)$$

If $\text{kind}(X)$ has canonical form



then by equation (6.31)

$$D_X(\psi) = \sum_{i=1}^n p_i \psi(v_i). \quad (19.15)$$

D_X only depends on $\text{kind}(X)$, not on X ’s specific value. We can just as well refer to the Distribution operator of a Kind k with $D_k(\psi) = \mathbb{E}(\psi(k))$.

Note the strong similarity between equations (19.15) and (19.9). This is not a coincidence. We can also think of D_X as a recipe for computing expectations with X . Equation (19.15) tells us exactly how to compute $\mathbb{E}(\psi(X))$

Example 19.5. Let C be a constant FRP with value v_0 . It’s Kind is

$$\langle \rangle \text{---} 1 \text{---} \langle v_0 \rangle$$

Then, for any statistic that accepts v_0 as a valid input, $\psi(C)$ is also a constant FRP with value $\psi(v_0)$. So, $\mathbb{E}(\psi(C)) = \psi(v_0)$, also by the Constancy property. It follows that

$$D_C(\psi) = \psi(v_0). \quad (19.16)$$

For any question, our predicted answer is the answer to that question at value v_0 .

Example 19.6. Let V be an event, an FRP with possible values 0 and 1 only. We can write $\text{kind}(V)$ as

$$\langle \rangle \begin{cases} \text{---} 1-p \text{---} \langle 0 \rangle \\ \text{---} p \text{---} \langle 1 \rangle \end{cases}$$

for some value $p \in [0\_1]$. Using equation (19.15), we see that

$$D_V(\psi) = (1-p)\psi(0) + p\psi(1) \quad (19.17)$$

for any statistic ψ that accepts 0 and 1 as inputs. For such a ψ , $\psi(V)$ has two possible values, $\psi(0)$ and $\psi(1)$, and its expectation is a weighted average of those two values, using the probabilities weighting $\text{kind}(V)$ as weights.

Example 19.7. If FRP S has Kind

$$\langle \rangle \begin{cases} \text{---} (1-p)^2 \text{---} \langle -1, -1 \rangle \\ \text{---} (1-p)p \text{---} \langle -1, 1 \rangle \\ \text{---} p(1-p) \text{---} \langle 1, -1 \rangle \\ \text{---} p^2 \text{---} \langle 1, 1 \rangle \end{cases}$$

for some $p \in [0\_1]$. What is its Distribution operator D_S ?

A statistic ψ that is compatible with S must include the set $\{-1, 1\}^2$ in its domain. Equations (6.31) and (19.15) tells us how to find the expectation of $\psi(S)$ from this information:

$$D_S(\psi) = (1-p)^2 \psi(-1, -1) + (1-p)p \psi(-1, 1) + p(1-p) \psi(1, -1) + p^2 \psi(1, 1).$$

For instance,

$$\begin{aligned} D_S(\text{Sum}) &= (1-p)^2 \cdot (-2) + (1-p)p \cdot 0 + p(1-p) \cdot 0 + p^2 2 \\ &= 2p^2 - 2(1-p)^2 = 2(2p-1) \end{aligned}$$

$$\begin{aligned} D_S(\text{Max}) &= (1-p)^2 \cdot (-1) + (1-p)p \cdot 1 + p(1-p) \cdot 1 + p^2 1 \\ &= 2p - p^2 - 2(1-p)^2 = -2 + 6p - 3p^2 \end{aligned}$$

$$\begin{aligned} D_S(|\blacksquare|^2) &= (1-p)^2 \cdot 2 + (1-p)p \cdot 2 + p(1-p) \cdot 2 + p^2 2 \\ &= 2. \end{aligned}$$

And if ψ is of the form $\psi(u, v) = \zeta(u) \xi(v)$ for statistics ζ and ξ , we write $\psi = \zeta \otimes \xi$, where $\otimes$ is the *tensor product* operator, and we call ψ the tensor product of ζ and ξ . Then

See Interlude F Section 15.5.

$$\begin{aligned} D_S(\zeta \otimes \xi) &= (1-p)^2 \zeta(-1) \xi(-1) + (1-p)p \zeta(-1) \xi(1) + p(1-p) \zeta(1) \xi(-1) + p^2 \zeta(1) \xi(1) \\ &= [(1-p)\zeta(-1) + p\zeta(1)] [(1-p)\xi(-1) + p\xi(1)]. \end{aligned}$$

This looks like the product of two expectations, and indeed it is. If $U = \text{proj}_1(S)$ and $V = \text{proj}_2(S)$, then $D_S(\zeta \otimes \xi) = D_U(\zeta) D_V(\xi)$. As we will see, this comes from the fact that S is an independent join: $S = U \star V$.

A point to keep in mind is that the distribution operator is a *function*, a function that takes other functions as input, which may seem odd at first but is not all that different from what we are used to. When we define generic functions, e.g., $f(x) = x^2$, we specify their output in terms of their input. The input parameters are *local variables*. If we write $f(z) = z^2$ or $f(t) = t^2$ or even $f(\text{foo}) = \text{foo}^2$, it is the same function.³ In the same way, if we write $D_X(\psi)$ or $D_X(\varphi)$ or even $D_X(\text{foo})$ it is the same function. The parameter is an arbitrary symbol – a local variable – that we use to specify the input statistic. In practice, we tend to use ψ by default, or φ , ζ , ξ as needed, but there is nothing magical about it.

³This is discussed extensively in Interlude F.

Example 19.8. An FRP R has Distribution operator

$$D_R(\psi) = \frac{1}{2} \psi(10) + \frac{1}{4} \psi(100) + \frac{1}{8} \psi(1000) + \frac{1}{8} \psi(10000).$$

Find its Kind and Kernel.

From the equation defining D_R , we can read off both the canonical weights

and corresponding values of $\text{kind}(R)$. Hence,

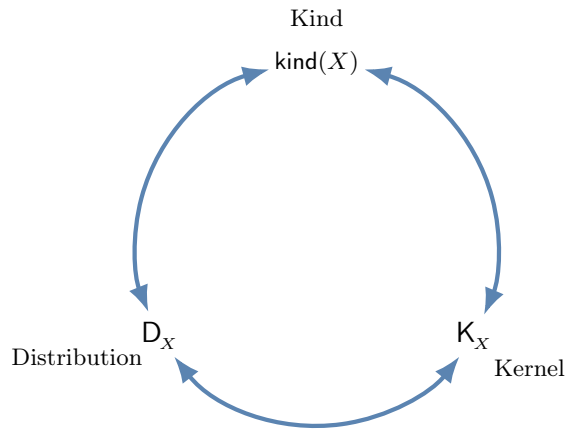
$$\langle \rangle \begin{cases} \frac{1}{2} & \langle 10 \rangle \\ \frac{1}{4} & \langle 100 \rangle \\ \frac{1}{8} & \langle 1000 \rangle \\ \frac{1}{8} & \langle 10000 \rangle \end{cases}$$

and

$$\mathbf{K}_R(v) = \frac{1}{2} \{v = 10\} + \frac{1}{4} \{v = 100\} + \frac{1}{8} \{v = 1000\} + \frac{1}{8} \{v = 10000\}.$$

19.3 Converting Among Kind, Kernel, and Distribution

The Kind, Kernel, and Distribution of an FRP are different packages for the same information, different interfaces to the same operations. And as such, we can convert among them. From the Kind, we can derive the Kernel and Distribution operator; from the Kernel, we can compute the Distribution operator and the Kind; and from the Distribution operator, we can compute the Kind and the Kernel.



From a Kind

$$\text{kind}(X) \langle \rangle \begin{cases} p_1 & v_1 \\ p_2 & v_2 \\ p_3 & v_3 \\ \dots & \dots \\ p_{n-1} & v_{n-1} \\ p_n & v_n \end{cases}$$

we can read off the weights and values to derive the Kernel and Distribution operator (first row of equations) and can derive the Kernel from the Distribution operator and vice versa (second row of equations).

$$K_X(v) = \sum_{i=1}^n p_i \{v = v_i\} \quad D_X(\psi) = \sum_{i=1}^n p_i \psi(v_i) \quad (19.18)$$

$$K_X(v) = D_X(\{\blacksquare = v\}) \quad D_X(\psi) = \sum_v K_X(v) \psi(v). \quad (19.19)$$

So we see that the kernel $K_X(v) = \mathbb{E}(\{X = v\})$, the probability that X has the value v , and $D_X(\psi)$ is a weighted average of ψ 's outputs with the Kernel's values as the weights.

Example 19.9 The Cumulative Distribution Function

Let X be a scalar FRP. For each real t , we can define a condition $\langle x \rangle \mapsto \{x \leq t\}$ that tests whether its input is less than or equal to the number t .

Define a function $F_X: \mathbb{R} \rightarrow [0\_1]$ by

$$F_X(t) = D_X(\{\blacksquare \leq t\}) = \mathbb{E}(\{X \leq t\}). \quad (19.20)$$

So for each t , $F_X(t)$ is the *probability of the event that X is less than or equal to t* . If you were given F_X , could you find the Kind and Kernel and Distribution operator for X ? The answer turns out to be yes! Let's see why.

Consider first what F_X looks like when X 's possible values are $v_1 < v_2 < \dots < v_m$. If $v_{j-1} < t < v_j$ for $j \in [2..n]$, then

$$\{X \leq t\} = \{X \leq v_{j-1}\} + \{v_{j-1} < X < v_j\} = \{X \leq v_{j-1}\}.$$

The first equality of events – binary-valued FRPs – follows just by logic. For $\{X \leq t\}$ to occur, exactly one of the two mutually exclusive events must occur. The second equality follows because the v_i 's are the only possible values of X , so $\{v_{j-1} < X < v_j\}$ *cannot occur* and thus must have value 0. Hence, $F_X(t) = F_X(v_{j-1})$.

But at $t = v_j$,

$$\{X \leq v_j\} = \{X < v_j\} + \{X = v_j\} = \{X \leq v_{j-1}\} + \{X = v_j\}$$

by the above logic, with the events on the right mutually exclusive, so taking expectations

$$F_X(v_j) = F_X(v_{j-1}) + K_X(v_j).$$

For values of t *between* the v_j 's, $F_X(t)$ is constant, but at each v_j , the function jumps by $K_X(v_j)$. So, F_X is a *piecewise constant* function with jumps at the possible values v_j of size $K_X(v_j)$.

See Example 12.1.

We now have all the ingredients we need to find the values and weights of $\text{kind}(X)$. Given F_X , we search its inputs looking for jumps, recording the values at which those jumps occur and the sizes of the jumps. When we have found jumps whose sizes sum to 1, we know we have them all. The Kind has the values at the leaves and the sizes of the jumps as corresponding weights. We can express the Kernel using this same logic, though with a bit of a mathematical trick:

$$K_X(v) = F_X(v) - F_X(v-), \quad (19.21)$$

where $F_X(v-)$ denotes the biggest value of $F_X(t)$ for all t *strictly less than* v . At a point v in between possible values, F_X is locally constant, so we will get 0 for the Kernel. But at a possible value v , we get a jump of precisely size $K_X(v)$. From the Kernel, we can reconstruct the Distribution operator as well using equation (19.15).

The function F_X gives us yet another way to package the information in the Kind. It was derived from D_X but we can in turn reconstruct D_X , K_X , and the Kind from it.

$$F_X(v) = D_X(\blacksquare \leq v) \quad D_X(\psi) = \sum_v (F_X(v) - F_X(v-)) \psi(v) \quad (19.22)$$

$$K_X(v) = F_X(v) - F_X(v-) \quad F_X(v) = \sum_t K_X(t) \{t \leq v\}, \quad (19.23)$$

where the bottom right equation is just equation (19.15).

We call F_X the **cumulative distribution function**, or CDF for short, because it *accumulates* probability as its argument grows.

19.4 Conditional Kinds and Conditional FRPs

Recall that a conditional FRP is a function that takes a value and returns an FRP; it represents a random quantity that is contingent on some other quantity. Similarly, a conditional Kind is a function that takes a value and returns a Kind. Both the Kinds returned by these functions (that include the input in their values) and the associated *target* Kinds (that exclude the input from their value) have Kernels and Distribution operators. By packaging these Kernels and Distributions into a function of the input value, we get functions that take a value and return a Kernel or Distribution. Naturally, we call these **conditional Kernels** and **conditional Distributions**. The wrinkle is that for the Kernel and Distribution representations, it is convenient to focus on the target Kinds, leaving the input value as part of our book keeping

(It is worth recalling that we can view all Kinds and FRPs as conditional, with ordinary Kinds and FRPs having codimension 0. The Aside below illustrates this.)

Suppose Z is a conditional FRP with conditional Kind z . (That is, $z(x) = \text{kind}(Z(x))$ for each input value x .)

As described in Chapter 4, $z(x)$ includes the input x at the beginning of every value. When describing this Kind by Kernels or Distributions, it is more convenient to use the *target Kinds* that exclude the input value from the tuple because we will include it elsewhere in our notation. Following `frplib`, we will write `z.target(x)` for the target Kind at input value x . this is just the Kind `z//constant(x)`.

The Kernel of the conditional Kind z is thus the function $K_z(\blacksquare \mid x)$ that takes a value of the *target* dimension and returns the canonical weight of that value in the *target* Kind `z.target(x)`, or 0 if it is not a possible value. Notice that we include the input value after the “given” bar, leaving the Kernel as a function of the later stage’s value (with input excluded). For each input value, we write this function as either $K_z(\blacksquare \mid x)$ or $K_z(\blacksquare \mid x)$.

This represented in `frplib` as well, where `ck.kernel(y, x)` corresponds to $K_{ck}(y \mid x)$ for a *target* value y and input value x :

```
pgd> ck = conditional_kind({0: either(1, 2), 1: uniform(500, 900, 1200)})
pgd> ck.kernel(1, 0)      # This means K_ck(1 | 0)
0.5
pgd> ck.kernel(900, 1)    # This means K_ck(900 | 1)
0.3333333333333333
```

Similarly, the Distribution of a conditional Kind z gives the distribution operator for the target Kind `z.target(x)` for each input value x . Again, we give the input value

after the “given” bar $|$ as book keeping.

$$D_Z(\psi \mid x) = \mathbb{E}(\psi(z \parallel \text{constant}(x))).$$

We can write this as D_z as well, if we like.

All Kinds are Conditional. An ordinary Kind can be viewed as a function from Value_0 , which has one possible value $\langle \rangle$ to the set of Kind trees, and vice versa. This relates to the comment earlier and to Example 14.3 in Interlude F that a function from the unit type Value_0 to some other type b is equivalent to an object of type b . Each function with Value_0 We associate the function to the value it returns and the value to the function that returns that value.

We can see this isomorphism directly in `frplib`. The following function takes a Kind `k` and returns a conditional Kind of codimension 0 whose target equals `k`:

```
def k_to_ck(k : Kind) -> ConditionalKind:
  @conditional_kind(codim=0, dim=dim(k))
  def ck():
    return k

  return ck
```

Conversely, the following function takes a conditional Kind `ck` of codimension 0 and returns its single target Kind:

```
def ck_to_k(ck : ConditionalKind) -> Kind:
  return ck()
```

These two functions are inverses of each other, e.g., `Kind.equal(k, ck_to_k(k_to_ck(k)))` returns `True` and similarly in the other direction. So we treat Kind_d as equivalent to the type $\text{ConditionalKind}_{0,d}$. The same logic shows that FRP_d as equivalent to the type $\text{ConditionalFRP}_{0,d}$.

When writing these expressions mathematically, it is often more convenient to work with the component FRPs and their Kinds *without naming the conditional FRP/Kind*. There is a good way to do this by using Definition 28 and the discussion on page 315 in Chapter 4.

If X is an FRP of dimension m and Z is a conditional FRP of type $m \rightarrow (m + n)$,

then we can define an n -dimensional FRP Y by either⁴

$$Y = Z // X \quad (19.24)$$

$$= \overline{\text{proj}}_{1..m}(X \triangleright Z). \quad (19.25)$$

⁴The second equation here just gives definition of the conditioning operator. See Section 15.2 in Interlude F for more on projections.

Conversely, if I have two FRPs X and Y of dimension m and n that are related by a join, it implies that there exists a conditional FRP that relates them as in the previous equations. Rather than giving this conditional FRP a name, we can denote it in a way that makes this relationship salient. We write $Y \mid X$ for this conditional FRP, and we call it the **conditional FRP of Y given X** :

$$X = \text{proj}_{1..m}(X \triangleright (Y \mid X)) \quad (19.26)$$

$$Y = \overline{\text{proj}}_{1..m}(X \triangleright (Y \mid X)) \quad (19.27)$$

$$= (Y \mid X) // X. \quad (19.28)$$

The conditional FRP $(Y \mid X)$ has conditional Kind $\text{kind}(Y \mid X)$, and $\text{kind}(Y) = \text{kind}(Y \mid X) // \text{kind}(X)$.

We write the Kernel and Distribution of $\text{kind}(Y \mid X)$ as

$$\mathbf{K}_{Y \mid X}(v \mid u) = \mathbb{E}(\{Y = v\} \mid X = u) \quad (19.29)$$

$$\mathbf{D}_{Y \mid X}(\psi \mid u) = \mathbb{E}(\psi(Y) \mid X = u). \quad (19.30)$$

Notice how the label maps to the arguments of both functions. In the first case, v is a possible value, and the Kernel returns the canonical weight on the target Kind for that value. In the second case, ψ is a statistic compatible with Y , and the Distribution returns the expectation of the transformed Y for that target.

So, $\mathbf{K}_{Y \mid X}(v \mid u)$ is the weight on the v -branch of the Kind returned by u in the conditional Kind. It is the **probability that Y 's value is v given that X 's value is u** . This is the expectation of the *event* $\{Y = v\} \mid X = u$ given the observation that X has value u . Because for each possible value u of X , $\mathbf{K}_{Y \mid X}(\blacksquare \mid u)$ is a valid Kernel, the canonical weights that must all sum to one:

$$\sum_v \mathbf{K}_{Y \mid X}(v \mid u) = 1. \quad (19.31)$$

Similarly, $\mathbf{D}_{Y \mid X}(\psi \mid u)$ is our **best prediction of $\psi(Y)$ given the knowledge that X equals u** . Because for each possible value u of X , $\mathbf{D}_{Y \mid X}(\blacksquare \mid u)$ is a valid

Distribution, the Constancy Rule holds:

$$D_{Y|X}(\text{const}_1 \mid u) = 1. \quad (19.32)$$

Moreover, we can convert between conditional Kernel and conditional Distribution in exactly the same way as before, for each possible value of X :

$$K_{Y|X}(v \mid u) = D_{Y|X}(\{\blacksquare = v\} \mid u) \quad (19.33)$$

$$D_{Y|X}(\psi \mid u) = \sum_v K_{Y|X}(v \mid u) \psi(v). \quad (19.34)$$

Compare these equations with (19.18) and (19.19). The equations are the same, only here we have the extra book keeping of tracking the input value.

The conditional Kernel and conditional Distribution give us a *family* of Kernels and Distributions, one for each possible value of X . For each valid u , For each valid input value u , $K_{Y|X}(\blacksquare \mid u)$ is a Kernel and $D_{Y|X}(\blacksquare \mid u)$ is a Distribution. Importantly however, these are only well-defined when u is a *possible value of X* . If u is not a possible value, the result is undefined.

Definition 48. Let X and Y be FRPs of dimension m and n that are related by a join. Then there is a conditional FRP $Y \mid X$ with conditional Kind $\text{kind}(Y \mid X)$ such that $Y = (Y \mid X) \Join X$.

The **conditional Kernel of Y given X** and **conditional Distribution of Y given X** are functions that take a possible value of X and return the Kernel and Distribution operator of the corresponding target. They are defined by

$$K_{Y|X}(y \mid x) = \mathbb{E}(\{Y = y\} \mid X = x) \quad (19.35)$$

$$D_{Y|X}(\psi \mid x) = \mathbb{E}(\psi(Y) \mid X = x). \quad (19.36)$$

Here y is a value of dimension n , and ψ is a statistic of codimension n that is compatible with Y .

The join FRP $X \triangleright (Y \mid X)$ and its Kind is the basic object from which derive the conditional Kernel and Distribution. When we wish to specify this join without a new name, we write it as $\langle X, Y \rangle$.

Definition 49. If X and Y are FRPs of dimension m and n that are the components of an $m+n$ -dimensional join FRP, we can write this join as $\langle X, Y \rangle$ without naming it. We have

$$\langle X, Y \rangle = X \triangleright (Y \mid X) \quad (19.37)$$

$$= Y \triangleright (X \mid Y) \quad (19.38)$$

$$X = \text{proj}_{1..m}(\langle X, Y \rangle) \quad (19.39)$$

$$Y = \overline{\text{proj}}_{1..m}(\langle X, Y \rangle). \quad (19.40)$$

And as for any FRP or Kind, we can derive the Kernel and Distribution operator of $\langle X, Y \rangle$. We can write these as $K_{\langle X, Y \rangle}$ and $D_{\langle X, Y \rangle}$, but in practice, we elide the delimiters and write them as K_{XY} and D_{XY} . The argument of K_{XY} is an $m+n$ -dimensional value, but because we are often operating over possible values of X and Y , we usually write the argument in terms of these values. Specifically, we typically write this in the form $K_{XY}(u, v)$ for an m -tuple u and an n -tuple v . D_{XY} takes a statistic of codimension $m+n$ that is compatible with the full join. Sometimes we do pass statistics of a form that separates the two pieces, as we will see.

Note that when we write something like $\mathbb{E}(\{Y = v\} \mid X = u)$ we are implicitly using the join here. This is made explicit in `frplib`. For instance, if XY is the join FRP $\langle X, Y \rangle$ we can compute $\mathbb{E}(\{Y = v\} \mid X = u)$ in two equivalent ways

$$\begin{aligned} & \mathbb{E}(\langle XY \mid (\text{Proj}[:, \text{d}+1]) == u \rangle \wedge (\text{Proj}[(\text{m}+1):] == v)). \\ & \mathbb{E}(\langle \text{Proj}[(\text{m}+1):] == v \rangle @ XY \mid (\text{Proj}[:, (\text{m}+1)] == u)). \end{aligned}$$

This shows how we get $\mathbb{E}(\{Y = v\} \mid X = u)$ by first applying a constraint to the join, then applying the condition $\langle y \rangle \mapsto \text{proj}_{(m+1)..(m+n)}(y) = v$, then taking expectations.⁵

Example 19.10. It is helpful to see an explicit connection between computational and mathematical versions of these ideas. Suppose we have

```
x = uniform(1, 2, 3)
y_given_x = conditional_kind({1: constant(1),
                             2: either(1, 3),
                             3: uniform(0, 3, 6)})
```

⁵The `@` operator exists to make this case more natural, with “at” indicating transformation by a statistic while maintaining a reference to the original join.

in `frplib`. These Kinds have the Kernels

$$\begin{aligned} K_X(u) &= \frac{1}{3} \{1, 2, 3\}(u) \\ K_{Y|X}(v \mid u) &= \{v = 1 \wedge u = 1\} + \frac{1}{2} \{v \in \{1, 3\} \wedge u = 2\} + \frac{1}{3} \{v \in \{0, 3, 6\} \wedge u = 3\}. \end{aligned}$$

Here we use both the long form of indicators, like $\{1, 2, 3\}$ which returns 1 when its argument is in $\{1, 2, 3\}$, and the Iverson brace expressions, like $\{u = 2\}$ which equals $\{\blacksquare = 2\}(u)$; see Chapter 12 in Interlude F. The indicators select appropriate values of u and v . In this case, we could conveniently write $K_{Y|X}$ as a table:

		v			
		0	1	3	6
u	1	0	1	0	0
	2	0	$\frac{1}{2}$	$\frac{1}{2}$	0
	3	$\frac{1}{3}$	0	$\frac{1}{3}$	$\frac{1}{3}$

Each row of the table corresponds to a Kernel for Y given the corresponding value of X .

We get the corresponding Distribution operators as well:

$$\begin{aligned} D_X(\psi) &= \sum_{i \in \{1, 2, 3\}} \frac{1}{3} \psi(i) \\ D_{Y|X}(v \mid u) &= \{u = 1\} \psi(1) + \frac{1}{2} \{u = 2\} (\psi(1) + \psi(3)) + \frac{1}{3} \{u = 3\} \sum_{i \in \{0, 3, 6\}} \psi(i) \\ &= \frac{1}{3} \{u = 3\} \psi(0) + (\{u = 1\} + \frac{1}{2} \{u = 2\}) \psi(1) \\ &\quad + (\frac{1}{2} \{u = 2\} + \frac{1}{3} \{u = 3\}) \psi(3) + \frac{1}{3} \{u = 3\} \psi(6) \\ &= \sum_v K_{Y|X}(v \mid u) \psi(v). \end{aligned}$$

We now consider how to express the Big 3+1 in terms of the Kernel and Distribution associated with a Kind or FRP. The important takeaway to keep in mind is that while the mechanics of how we do the computations are superficially different among Kind, Kernel, and Distribution the structure of those computations is exactly the same. See the discussion of this for each operation in the sections to follow.

19.5 Transforming with Statistics

Recall what it means to transform an FRP with a statistic. We connect the original FRP to an empty FRP with an adapter that computes the statistic. When the original FRP is activated, its value is passed as input to the adapter, which computes the value of the statistic and activates the transformed FRP.

So, if X is an FRP and φ a compatible statistic, then $\varphi(X)$ is a new, related FRP. When X is fresh, so is $\varphi(X)$, but when X is activated and produces value x , $\varphi(X)$ is activated and produces value $\varphi(x)$. The relation between X and $\varphi(X)$ lets us compute the $\text{kind}(\varphi(X))$ from $\text{kind}(X)$. Because $\text{kind}(X)$ tells us all possible values of X and their weights, we can determine all possible values of $\varphi(X)$ and their weights. Indeed, as we have seen

$$\varphi(\text{kind}(X)) = \text{kind}(\varphi(X))$$

with the transform of the Kind equal to the Kind of the transform. To transform a Kind, we **apply the statistic to the value at each leaf and then combine branches that map to the same value, adding their weights**. In this section, we will see how this process of transforming Kinds manifests in terms of their Kernels and Distributions.

THE KERNEL. Any possible value u of $\varphi(X)$ must equal $\varphi(v)$ for at least one possible value v of X . By the process of transforming Kinds, we know that the weight on u in the Kind of $\varphi(X)$ is just the sum of the weights of all such v 's. So the process of mapping values and combining branches that produce the same value yields

$$K_{\varphi(X)}(u) = \sum_v K_X(v) \{\varphi(v) = u\}. \quad (19.41)$$

Look closely at this sum. The only terms that are non-zero are those for which $\varphi(v) = u$. For those terms, the indicator equals 1, so we get the weight on the corresponding branch of $\text{kind}(X)$. Thus, we add up the weights over all values of X that lead to the value u of $\varphi(X)$, accounting for all the ways that u might be produced. The resulting weight is exactly what we get for the branch of u in the transformed Kind.

THE DISTRIBUTION. The Distribution of a transformed FRP is even easier to write. Let $Y = \varphi(X)$ be the transformed FRP. For an arbitrary statistic ψ compatible

with Y , we have

$$D_Y(\psi) = \mathbb{E}(\psi(Y)) = \mathbb{E}(\psi(\varphi(X))) = \mathbb{E}((\psi \circ \varphi)(X)) = D_X(\psi \circ \varphi),$$

so we can express D_Y directly in terms of D_X . (Remember that the composition $\psi \circ \varphi$ is ψ *after* φ ; see Chapter 13 in Interlude F.) The key to this is the third equality: applying ψ to $\varphi(X)$ is the same as applying φ then ψ to X , or equivalently applying ψ after φ to X . Hence:

$$D_{\varphi(X)}(\psi) = D_X(\psi \circ \varphi). \quad (19.42)$$

The Distribution thus transforms by composition of the statistics.

Example 19.11. Let $\text{kind}(X)$ be the Kind

$$\langle \rangle \begin{cases} \frac{2}{12} & \langle -4 \rangle \\ \frac{1}{12} & \langle -1 \rangle \\ \frac{4}{12} & \langle 3 \rangle \\ \frac{2}{12} & \langle 6 \rangle \\ \frac{2}{12} & \langle 8 \rangle \\ \frac{1}{12} & \langle 11 \rangle \end{cases}$$

and consider the transform of X by the statistic $\varphi(x) = (x-1)^2$. The transformed Kind $\text{kind}(\varphi(X))$ is

$$\langle \rangle \begin{cases} \frac{5}{12} & \langle 4 \rangle \\ \frac{4}{12} & \langle 25 \rangle \\ \frac{2}{12} & \langle 49 \rangle \\ \frac{1}{12} & \langle 100 \rangle \end{cases}$$

Applying equation (19.41), we get the Kernel

$$\begin{aligned} K_{\varphi(X)}(u) &= \sum_v K_X(v) \{\varphi(v) = u\} \\ &= \frac{2}{12} \{(-4-1)^2 = u\} + \frac{1}{12} \{(-1-1)^2 = u\} + \frac{4}{12} \{(3-1)^2 = u\} \\ &\quad + \frac{2}{12} \{(6-1)^2 = u\} + \frac{2}{12} \{(8-1)^2 = u\} + \frac{1}{12} \{(11-1)^2 = u\} \\ &= \left(\frac{1}{12} + \frac{4}{12}\right) \{4 = u\} + \left(\frac{2}{12} + \frac{2}{12}\right) \{25 = u\} + \frac{2}{12} \{49 = u\} + \frac{1}{12} \{100 = u\} \end{aligned}$$

$$= \frac{5}{12} \{4 = u\} + \frac{4}{12} \{25 = u\} + \frac{2}{12} \{49 = u\} + \frac{1}{12} \{100 = u\}.$$

Notice how the branches are *automatically* combined in the sum.

Applying equation (19.42), we get the Distribution

$$\begin{aligned} D_{\varphi(X)}(\psi) &= D_K(\psi \circ \varphi) \\ &= \frac{2}{12} \psi((-4-1)^2) + \frac{1}{12} \psi((-1-1)^2) + \frac{4}{12} \psi((3-1)^2) \\ &\quad + \frac{2}{12} \psi((6-1)^2) + \frac{2}{12} \psi((8-1)^2) + \frac{1}{12} \psi((11-1)^2) \\ &= \frac{5}{12} \psi(4) + \frac{4}{12} \psi(25) + \frac{2}{12} \psi(49) + \frac{1}{12} \psi(100). \end{aligned}$$

Example 19.12 Projections

Projection statistics are commonly used transformations and provide a good way to understand equations (19.41) and (19.42) because the input values that map to the same output are easy to recognize and express.

Suppose we have a 3-dimensional FRP W whose Kernel and Distribution we know. Our goal is to find the Kernel and Distribution of the projected FRPs $S = \text{proj}_2(W)$ and $T = \text{proj}_{1,3}(W)$.

We'll start with the Distributions as they are easy consequences of (19.42):

$$\begin{aligned} D_S(\psi) &= D_W(\psi \circ \text{proj}_2) \\ D_T(\xi) &= D_W(\xi \circ \text{proj}_{1,3}). \end{aligned}$$

We simply extract the desired components of W and then apply the statistic and expectation. Keep in mind that ψ and ξ must be compatible with S and T , respectively, not with W .

For the Kernels, we have for S :

$$\begin{aligned} K_S(s) &= \sum_{a,b,c} K_W(a,b,c) \{\text{proj}_2(\langle a,b,c \rangle) = s\} \\ &= \sum_{a,b,c} K_W(a,b,c) \{b = s\} \\ &= \sum_{a,c} K_W(a,s,c). \end{aligned}$$

The indicator combines the weights over all branches $\langle a, s, c \rangle$ of W 's Kind in which the second component is fixed at s . The sum here is over all values of W 's components; we could write it as a triple sum $\sum_a \sum_b \sum_c$, which you can think of as a triple loop. Similarly for T :

$$\begin{aligned} K_T(t_1, t_2) &= \sum_{a,b,c} K_W(a, b, c) \{ \text{proj}_{1,3}(\langle a, b, c \rangle) = \langle t_1, t_2 \rangle \} \\ &= \sum_{a,b,c} K_W(a, b, c) \{ t_1 = a \wedge t_3 = c \} \\ &= \sum_b K_W(t_1, b, t_2). \end{aligned}$$

Again, the statistic maps all values $\langle t_1, b, t_2 \rangle$ to the same value of T and we combine the weights of those branches to get the weight of the $\langle t_1, t_2 \rangle$ branch of $\text{kind}(T)$.

19.6 Building with Joins

To begin, consider independent joins. The Kind of independent join $X \star Y$ is formed by gluing a copy of $\text{kind}(Y)$ to each leaf node of $\text{kind}(X)$ and then including the values of both in the tuples. Assuming all the Kinds are in canonical form, we get an unfolded tree where the weights emanating from each branch point sum to 1. We know from ALGORITHM COMPACT in Chapter 3 that the weight on a value v in the canonical Kind can be obtained by *multiplying all the weights on the path from root to leaf* in the unfolded tree.

For value $\langle x, y \rangle$: from the root we take the x branch of $\text{kind}(X)$ which has weight $K_X(x)$ and from that node we take the y branch of $\text{kind}(Y)$ which has weight $K_Y(y)$. We obtain a final weight of $K_X(x) K_Y(y)$. This reflects the independence of X and Y . The value produced by Y has the same weight whatever X 's value is (and vice versa).

Consider Example 4.2 on page 241 of Chapter 4. The unfolded tree for the independent join is shown in Figure 4.10. The canonical weight in the independent join for value $\langle 2, 8 \rangle$ is $\frac{2}{21} \cdot \frac{2}{21}$; the weight for $\langle 4, 12 \rangle$ is $\frac{4}{21} \cdot \frac{6}{21}$; and the weight for $\langle 1, 10 \rangle$ is $\frac{1}{21} \cdot \frac{4}{21}$. In each case, we choose the weight for the first component then the weight for the second component in the subtree and multiply them.

In general,

$$K_{X \star Y}(u :: v) = K_X(u) \cdot K_Y(v). \quad (19.43)$$

Here, $u :: v$ expresses the value of the join as the *concatenation*⁶ of a tuple that is

⁶See Interlude F Section 15.4 for more on the concatenation operator ($::$).

a value for X and a tuple that is a value for Y . Specifically, supposing that X has dimension m and Y has dimension n . Then, the tuple $u :: v$ has dimension $m + n$ with the first m components comprising the tuple u and the last n components comprising the tuple v .

This generalizes to the independent join of any number of FRPs.

$$K_{X_1 * X_2 * \dots * X_n}(u_1 :: u_2 :: \dots :: u_n) = K_{X_1}(u_1) K_{X_2}(u_2) \dots K_{X_n}(u_n). \quad (19.44)$$

Think of this in terms of the unfolded Kind of the join. This is a tree with n levels. At the first branch, we choose a value u_1 of X_1 ; at the second branch, in the u_1 subtree, we choose a value u_2 of X_2 ; and so on until we choose a value u_n of X_n from the subtree at the bottom level. The final canonical weight on the value $u_1 :: u_2 :: \dots :: u_n$ is the product of the canonical weights on the path in the tree from root to leaf. So, the Kernel for an independent join separates into the product of the Kernels over all the components. This reflects the independence: the weight of a value at each stage does not depend on the values of other stages.

The idea is the same for a general join, except the weight at any stage *now depends on where you are in the tree*, on the values at previous stages. Consider the general join $\langle X, Y \rangle = X \triangleright (Y \mid X)$. We obtain its Kind by gluing the target Kind from $\text{kind}(Y \mid X = u)$ to the leaf node for u in $\text{kind}(X)$, for each possible value u of X . Again, the weights of the final value are obtained by *multiplying the canonical weights on the path from root to leaf*. To get to value $u :: v$, we take the u branch for X stage and then take the v branch in the v -subtree for the Y stage. So, the join Kernel K_{XY} for is defined by

$$K_{XY}(u, v) = K_X(u) K_{Y \mid X}(v \mid u). \quad (19.45)$$

The independent join Kernel is just a special case of these, where the conditional Kind gives the same target for every input. With a join, we can decompose it in the opposite order, unfolding the tree with the Y -stage first. As such, the following is also true:

$$K_{XY}(u, v) = K_Y(v) K_{X \mid Y}(u \mid v). \quad (19.46)$$

We can use either form as needed.

Example 19.13. We choose roll a balanced six-sided die and roll it. We then flip a balanced coin the number of times equal to the die roll. Assume all the

flips are independent.

What is the Kernel of the sequence of flips?

```
d = uniform(1, 2, ..., 6)
c_given_d = conditional_kind({
    j: binary() ** j for j in irange(1, 6)
})
dc = d >> c_given_d
c = Proj[1](dc)
```

In practice, we would pad all these Kinds out to a common dimension, but this is not strictly needed for our purposes here.

To find the Kernels, we start with the Kind and conditional Kind in the join. Define FRPs: D is the die roll and C is the sequence of flips as binary values with 1 representing heads.

$$\begin{aligned} K_D(j) &= \sum_{i=1}^6 \frac{1}{6} \{i = j\} = \frac{1}{6} \{j \in [1..6]\} \\ K_{C|D}(u | j) &= \prod_{i=1}^j K_{C_i}(u_i) \{\dim(u) = j\} = 2^{-j} \{\dim(u) = j\} \\ K_{DC}(j, u) &= K_D(j) K_{C|D}(u | j) \\ &= \frac{2^{-j}}{6} \{j \in [1..6]\} \{\dim(u) = j\}. \end{aligned}$$

Applying proj_2 , we have

$$K_C(u) = \sum_{j=1}^6 K_{DC}(j, u) = 2^{-\dim(u)} \{\dim(u) \in [1..6]\}.$$

Example 19.14. A broken warning light can be on or off. It starts either on or off, uniformly at random, and at each tick of the clock, it changes state randomly as follows.

```
start = either(0, 1) # 1 is on, 0 is off
@conditional_kind
def tick(history):
    current = history[-1] # most recent state
    if current == 0: # off
```

```

    return weighted_as({0: 3/4, 1: 1/4})
return weighted_as({0: 1/4, 1: 3/4})

```

So light is more likely to stay in whichever state it is in than it is to switch.

Let L_0 be the FRP representing the initial state and let L_n be the FRP representing the state of the light after n ticks of the clock.

This system has the *Markov property* discussed in Chapter 7: once you know the current state of the system, the future evolution of the system is independent of the previous states. Knowing L_0 and L_1 gives us the *same predictions* about L_2 (and $L_3, L_4, \dots$) as knowing just L_1 . This means that for all values of ℓ_0 :

$$\begin{aligned}
 K_{L_2|L_1L_0}(\ell_2 \mid \ell_1, \ell_0) &= K_{L_2|L_1}(\ell_2 \mid \ell_1) \\
 D_{L_2|L_1L_0}(\psi \mid \ell_1, \ell_0) &= D_{L_2|L_1}(\psi \mid \ell_1).
 \end{aligned}$$

Thus to analyze the evolution of the system, we only need to pay attention to the Kind of the initial state L_0 and the conditional Kind of the one-step transitions from L_{n-1} to L_n , for $n \in [1..)$.

The Kernel of the initial state is

$$K_{L_0}(s_0) = \frac{1}{2} \{s_0 = 0\} + \frac{1}{2} \{s_0 = 1\} = \frac{1}{2} 0, 1(s_0),$$

and for every n , the conditional Kernel is

$$K_{L_n|L_{n-1}}(s_n \mid s_{n-1}) = \frac{3}{4} \{s_n = s_{n-1}\} + \frac{1}{4} \{s_n = 1 - s_{n-1}\}.$$

We can describe the history through the n th tick by taking repeated joins

```
start >> tick >> tick >> tick >> ... # n ticks
```

Using the Markov property, we can write this as

$$\text{kind}(L_0) \triangleright \text{kind}(L_1 \mid L_0) \triangleright \text{kind}(L_2 \mid L_1) \triangleright \dots \triangleright \text{kind}(L_n \mid L_{n-1}),$$

which is an $(n+1)$ -dimensional Kind (for the initial state and n transitions).

The Kernel of this join tells us the probability of any sequence of states

through the n th tick. Applying equation (19.45) yields

$$\begin{aligned} K_{L_0 L_1 \dots L_n}(s_0, s_1, \dots, s_n) &= K_{L_0}(s_0) K_{L_1|L_0}(s_1 | s_0) K_{L_1|L_0}(s_1 | s_0) \cdots K_{L_n|L_{n-1}}(s_n | s_{n-1}) \\ &= \frac{1}{2} \{0, 1\}(s_0) \left(\frac{3}{4} \{s_1 = s_0\} + \frac{1}{4} \{s_1 = 1 - s_0\} \right) \\ &\quad \cdot \left(\frac{3}{4} \{s_2 = s_1\} + \frac{1}{4} \{s_2 = 1 - s_1\} \right) \cdots \left(\frac{3}{4} \{s_n = s_{n-1}\} + \frac{1}{4} \{s_n = 1 - s_{n-1}\} \right). \end{aligned}$$

If we want the Kernel of L_n , we apply the projection statistic proj_{n+1} :

$$\begin{aligned} K_{L_n}(s) &= \sum_{s_0, \dots, s_n} K_{L_0 L_1 \dots L_n}(s_0, s_1, \dots, s_n) \{ \text{proj}_{n+1}(s_0, \dots, s_n) = s \} \\ &= \sum_{s_0, \dots, s_n} K_{L_0 L_1 \dots L_n}(s_0, s_1, \dots, s_n) \{ s_n = s \} \\ &= \sum_{s_0, \dots, s_{n-1}} K_{L_0}(s_0) K_{L_1|L_0}(s_1 | s_0) K_{L_1|L_0}(s_1 | s_0) \cdots K_{L_n|L_{n-1}}(s | s_{n-1}). \end{aligned}$$

At first glance, this expression looks nasty, but if we look closely, we can see the meaning here. The sum is over all $(n+1)$ -tuples containing 0s and 1s. Each such tuple represents one path of evolution for the system from the initial state to the state after the n th tick of the clock. For each such path, we evaluate the Kernel, which via (19.45) breaks the probability into a product of the probability of starting in the specified initial state and the probabilities of making *each* step through the n th tick.

The good news is that using the technique in Section 17.2 of Interlude F, we can rewrite this sum in a far simpler form using *matrices*:

$$\begin{bmatrix} K_{L_n}(0) \\ K_{L_n}(1) \end{bmatrix} = \begin{bmatrix} K_{L_0}(0) & K_{L_0}(1) \end{bmatrix} \begin{bmatrix} K_{L_n|L_{n-1}}(0 | 0) & K_{L_n|L_{n-1}}(1 | 0) \\ K_{L_n|L_{n-1}}(0 | 1) & K_{L_n|L_{n-1}}(1 | 1) \end{bmatrix}^n = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \frac{3}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}^n.$$

For each n , we can solve this in various ways, the easiest is to use an eigenvalue decomposition of the symmetric matrix on the right. We will explore this transformation at length later.

Joins can also be expressed in terms of the Distribution operator and has a particularly nice form for independent joins. As we will see in later sections, the conditioning operator is also elegantly expressed in terms of Distribution operators.

For independent joins, equations (19.43) and (19.44) show that the Distribution operator has a simple form for particular statistics. As described in Section 15.5 in Interlude F, given statistics ψ and φ of codimensions m and n , the statistic $\psi \otimes \varphi$ of

codimension $m + n$, called the tensor product, is defined by

$$(\psi \otimes \varphi)(u :: v) = \psi(u) \varphi(v),$$

where u and v are tuples of dimension m and n respectively. For such statistics, equation (19.43) gives us for X and Y of dimensions m and n that

$$D_{X*Y}(\psi \otimes \varphi) = \sum_u \sum_v K_{X*Y}(u :: v) (\psi \otimes \varphi)(u :: v) \quad (19.47)$$

$$= \sum_u \sum_v K_X(u) K_Y(v) \psi(u) \varphi(v) \quad (19.48)$$

$$= \sum_u K_X(u) \psi(u) \left(\sum_v K_Y(v) \varphi(v) \right) \quad (19.49)$$

$$= \left(\sum_u K_X(u) \psi(u) \right) \left(\sum_v K_Y(v) \varphi(v) \right) \quad (19.50)$$

$$= D_X(\psi) D_Y(\varphi). \quad (19.51)$$

This generalizes to any number of terms in the independent join by equation (19.44). Letting $X_1, \dots, X_r$ have compatible statistics $\psi_1, \dots, \psi_r$ with corresponding codimensions, we have

$$D_{X_1 * \dots * X_r}(\psi_1 \otimes \dots \otimes \psi_r) = D_{X_1}(\psi_1) \dots D_{X_r}(\psi_r). \quad (19.52)$$

In particular, taking $\psi_i = \text{id}$, when $X_1, X_2, \dots, X_r$ are independent

$$\mathbb{E}(X_1 X_2 \dots X_r) = \mathbb{E}(X_1) \mathbb{E}(X_2) \dots \mathbb{E}(X_r).$$

For general joins, the relationship is more complicated, giving an iterated application of the conditional Distributions. Let ψ be a statistic that is compatible with the join $\langle X, Y \rangle = X \triangleright (Y \mid X)$. For each possible value x of X , write $(\psi(x, \blacksquare))$ for the statistic that fixes X 's value at x . Then:

$$D_{XY}(\psi) = D_X \left(\langle u \rangle \mapsto D_{Y \mid X}(\psi(u, \blacksquare) \mid u) \right). \quad (19.53)$$

Take this from the inside out. The statistic ψ is compatible with the join FRP $\langle X, Y \rangle$. We start by asking a question about $\psi(\langle X, Y \rangle)$ given the constraint that X is known to be u , for a possible value of X . This gives us $D_{Y \mid X}(\psi(u, \blacksquare) \mid u) = \mathbb{E}(\psi(Y \mid X) \mid X = u)$, and putting it together for every value of X , we get a function (in fact, a

statistic) that returns that prediction for each x . Passing this statistic to D_X averages all those answers. We get the join with nested applications of the Distribution operators for $Y | X$ and X , respectively. Again, this works if we decompose in the other order as well:

$$D_{XY}(\psi) = D_Y \left(\langle v \rangle \mapsto D_{Y|X}(\psi(\blacksquare, v) | v) \right). \quad (19.54)$$

We will return to this result in Chapter 24, where we will see it in an easier form with the Mighty Conditioning Identity, one of our most powerful techniques.

19.7 Constraining with Observations

When we constrain a Kind with an observation, we simply **erase all the branches that are inconsistent with the constraint**. We can view the action of erasing a branch as being equivalent to *setting the branch's weight to 0*.

A constraint is expressed in terms of a *condition* on one or more FRPs, a binary-valued statistic that we interpret as a Boolean. For example, the constrained FRP $X | X > 5$ is $X | \xi$ for the condition $\xi(u) = \{u > 5\}$. In practice, we *inline* such constraints when convenient rather than naming a new condition, writing $X | X > 5$ rather than $X | \xi$. We also allow arbitrary Boolean expressions in the constraints that can involve more than one FRP

In `frplib`, we had to constrain an FRP that includes all the used quantities as its components, but mathematically, we can be more succinct. For instance, to write $X | Y < 0$ in `frplib`, we would do

```
Proj[1] @ XY | (Proj[2] < 0)
```

where XY is the join $\langle X, Y \rangle$. This enabled `frplib` to apply the constraint correctly. With equations, we can use the more succinct notation, though the join is implicitly there.

Erasing the branches inconsistent with the constraint ξ changes the Kernel by simply excluding values for which ξ is 0. We do that by multiplying

$$K_X(v) \xi(v).$$

This will be non-zero only when the constraint is satisfied because $\xi(v) = 1$ when it is satisfied and $\xi(v) = 0$ otherwise.

That's all there is to it, except that with Kernels we keep them in canonical form. So, we need to renormalize so that the weights of the constrained Kernel sum to 1.

The sum of the weights is just

$$\sum_v \mathsf{K}_X(v) \xi(v),$$

where we sum over all values of X . Putting this in the denominator gives

$$\sum_v \frac{\mathsf{K}_X(v) \xi(v)}{\sum_u \mathsf{K}_X(u) \xi(u)} = \frac{\sum_v \mathsf{K}_X(v) \xi(v)}{\sum_u \mathsf{K}_X(u) \xi(u)} = 1.$$

Hence, the Kernel of the constrained FRP is

$$\mathsf{K}_{X|\xi}(v) = \frac{\mathsf{K}_X(v) \xi(v)}{\sum_u \mathsf{K}_X(u) \xi(u)}. \quad (19.55)$$

The same principle applies with the Distribution. We have

$$\begin{aligned} \mathsf{D}_{X|\xi}(\psi) &= \mathbb{E}(\psi(X) \mid \xi) \\ &= \frac{\sum_v \psi(v) \mathsf{K}_X(v) \xi(v)}{\sum_v \mathsf{K}_X(v) \xi(v)} \\ &= \frac{\mathsf{D}_X(\psi \cdot \xi)}{\mathsf{D}_X(\xi)}. \end{aligned}$$

Here, $(\psi \cdot \xi)(v) = \psi(v) \cdot \xi(v)$ and the denominator is the normalizing constant

$$\mathsf{D}_{X|\xi}(\psi) = \frac{\mathsf{D}_X(\psi \cdot \xi)}{\mathsf{D}_X(\xi)}. \quad (19.56)$$

The denominator here is just

$$\mathsf{D}_X(\xi) = \sum_v \mathsf{K}_X(v) \xi(v),$$

as before.

Example 19.15. We have two buckets. In bucket 1, there are 5 blue balls and 10 green balls. In bucket 2, there are 8 blue balls, 6 green balls, and 2 red balls. We choose a bucket at random, uniformly, and then choose a ball, uniformly from that bucket.

You observe that the chosen ball is blue. What is the Kind of the FRP representing the selected bucket? Give the Kernel and Distribution operator for this Kind.

```

bucket = either(1, 2)
ball = conditional_kind({
  1: weighted_as(1, 0, weights=[5, 10]),
  2: weighted_as(1, 0, 2, weights=[8, 6, 2]),
})
bb = bucket >> ball
selected = Proj[1]( (bucket >> ball) | (Proj[2] == 1) )
# or
selected = bayes(1, bucket, ball)

selected.kernel
D_(selected)

```

Let's compute the Kernels and Distribution operator for these Kinds. Let B be the two-dimensional FRP representing the selected bucket and the chosen ball, with $B_i = \text{proj}_i(B)$ the two components.

$$\begin{aligned}
K_{B_1}(u) &= \frac{1}{2} \{u = 1\} + \frac{1}{2} \{u = 2\} \\
K_{B_2|B_1}(v | u) &= \{u = 1\} \left(\frac{2}{3} \{v = 0\} + \frac{1}{3} \{v = 1\} \right) \\
&\quad + \{u = 2\} \left(\frac{3}{8} \{v = 0\} + \frac{1}{2} \{v = 1\} + \frac{1}{8} \{v = 2\} \right) \\
K_B(u, v) &= K_{B_1}(u) K_{B_2|B_1}(v | u) \\
&= \frac{1}{2} \{u = 1\} \left(\frac{2}{3} \{v = 0\} + \frac{1}{3} \{v = 1\} \right) \\
&\quad + \frac{1}{2} \{u = 2\} \left(\frac{3}{8} \{v = 0\} + \frac{1}{2} \{v = 1\} + \frac{1}{8} \{v = 2\} \right) \\
&= \frac{1}{3} \{v = 0\} \{u = 1\} + \frac{1}{6} \{v = 1\} \{u = 1\} \\
&\quad + \frac{3}{16} \{v = 0\} \{u = 2\} + \frac{1}{4} \{v = 1\} \{u = 2\} + \frac{1}{16} \{v = 2\} \{u = 2\}. \\
D_B(\psi) &= \frac{1}{3} \psi(0, 1) + \frac{1}{6} \psi(1, 1) + \frac{3}{16} \psi(0, 2) + \frac{1}{4} \psi(1, 2) + \frac{1}{16} \psi(2, 2).
\end{aligned}$$

Now we constrain with the observation $\xi(u, v) = \{v = 1\}$, giving the FRP $B | B_2 = 1$. Applying proj_1 gives us $B_1 | B_2 = 1$.

$$K_B(u, v) \{v = 1\} = \frac{1}{6} \{v = 1\} \{u = 1\} + \frac{1}{4} \{v = 1\} \{u = 2\}.$$

Now, we re-normalize and project.

$$\begin{aligned} K_{B_1|B_2}(u | 1) &= \frac{2}{5} \{u = 1\} + \frac{3}{5} \{u = 2\} \\ D_{B_1|B_2}(\psi | 1) &= \frac{2}{5} \psi(1) + \frac{3}{5} \psi(2). \end{aligned}$$

19.8 Predicting with Expectations

Earlier, we derived the formula for $\mathbb{E}(\psi(X))$ or $\mathbb{E}(\psi(K))$ where X is an FRP, $K = \text{kind}(X)$, and ψ is a compatible statistic:

$$\mathbb{E}(\psi(X)) = \sum_{i=1}^n p_i \psi(v_i).$$

But this is just the expression for $D_X(\psi) = D_K(\psi)$! So, *the distribution operator just embodies our original formula for expectation.*

For the Kernel, $p_i = K_X(v_i)$, so we can write this as

$$\mathbb{E}(\psi(X)) = \sum_{i=1}^n K_X(v_i) \psi(v_i),$$

and since the Kernel is zero for all values not in the Kind:

$$D_X(\psi) = \mathbb{E}(\psi(X)) = \sum_v K_X(v) \psi(v) \quad (19.57)$$

where the sum is over all values.

Example 19.16. In an earlier example, we transformed the Kind

$$\langle \rangle \begin{array}{l} \frac{2}{12} \text{ --- } \langle -4 \rangle \\ \frac{1}{12} \text{ --- } \langle -1 \rangle \\ \frac{4}{12} \text{ --- } \langle 3 \rangle \\ \frac{2}{12} \text{ --- } \langle 6 \rangle \\ \frac{2}{12} \text{ --- } \langle 8 \rangle \\ \frac{1}{12} \text{ --- } \langle 11 \rangle \end{array}$$

by $\varphi(x) = (x - 1)^2$, obtaining the Kind

$$\langle \rangle \begin{cases} \frac{5}{12} & \langle 4 \rangle \\ \frac{4}{12} & \langle 25 \rangle \\ \frac{2}{12} & \langle 49 \rangle \\ \frac{1}{12} & \langle 100 \rangle \end{cases}$$

Using the Kernel, we have

$$\begin{aligned} \mathbb{E}(\varphi(X)) &= \mathbb{E}((X-1)^2) \\ &= \sum_v \mathsf{K}_X(v) (v-1)^2 \\ &= \frac{2}{12} (-4-1)^2 + \frac{1}{12} (-1-1)^2 + \frac{4}{12} (3-1)^2 \\ &\quad + \frac{2}{12} (6-1)^2 + \frac{2}{12} (8-1)^2 + \frac{1}{12} (11-1)^2 \\ &= \frac{5}{12} 4 + \frac{4}{12} 25 + \frac{2}{12} 49 + \frac{1}{12} 100 = \frac{318}{12}. \end{aligned}$$

The branches are automatically combined in this sum as before.

Using the Distribution operator, we simply compose the statistics and treat the form of the Distribution as a recipe for computing expectations.

$$\begin{aligned} \mathsf{D}_{\varphi(K)}(\psi) &= \mathsf{D}_K(\psi \circ \varphi) \\ &= \frac{5}{12} \psi(4) + \frac{4}{12} \psi(25) + \frac{2}{12} \psi(49) + \frac{1}{12} \psi(100) \end{aligned}$$

This gives $\mathbb{E}(\psi(\varphi(K))) = \mathbb{E}(\psi((X-1)^2))$, and plugging in $\psi = \text{id}$ yields

$$\mathbb{E}(\varphi(X)) = \frac{5}{12} 4 + \frac{4}{12} 25 + \frac{2}{12} 49 + \frac{1}{12} 100 = \frac{318}{12}.$$

Infinites, Large and Small

20

Chapter

Contents

20.1 Infinite Kinds	866
20.2 Mass and Density	870
20.3 On Infinitesimals, Differentials, and Measure	878
20.4 Distributions and Kernels Revisited	886
20.5 FRPs, Random Processes, and Random Variables	891

We are now ready to enlarge our view to encompass *infinite* systems. This will include random quantities that have an infinite set of possible values, and random systems that evolve over an infinite horizon of time or space. Our basic operations, their meaning, and the logic that we will use *will not change*. And we will also use the same language to describe and analyze random systems. The presence of infinities can affect how we do the calculations and the boundary cases we need to check, but the concepts are the same. This Chapter introduces infinite random systems into our framework.

An important distinction we have to make at the start is between two types of infinities: infinite sets that look like the integers and infinite sets that look like the real numbers. This is discussed in more detail in Sections 9.4 and 14.1 (see, e.g., Definition 14.1). We will only hit the high points here.

Two sets $\mathcal{A}$ and $\mathcal{B}$ have the same cardinality if we can find a bijection $f: \mathcal{A} \rightarrow \mathcal{B}$. This f is a one-to-one correspondence between elements of the two sets; think of $f(a)$ as labeling a with a value of $b \in \mathcal{B}$. A bijection means that every a gets a label and every b is used as a label *exactly once*.

A set $\mathcal{A}$ is **countably infinite** if there is a bijection between $\mathcal{A}$ and the set of integers. That is, we can give each $\mathcal{A}$ a distinct integer label, using all the integers. For example, the set of even integers is countably infinite as is the set of rational numbers.¹

A set $\mathcal{B}$ is **uncountably infinite** if there is a bijection between $\mathcal{B}$ and the set of real numbers. That is, we can give each $\mathcal{B}$ a distinct numeric label, using all the real

¹This is somewhat counterintuitive as the evens are a strict subset of the integers, which are a strict subset of the rational. Infinities can be weird.

numbers.

No matter how we try label an uncountable set with integers, there are always more to label. The cardinality of an uncountably infinite set is strictly bigger than that of a countably infinite set. If we think of the elements of a countably infinite set as discrete objects, like labeled marbles, the elements of an uncountably infinite set comprise a continuous smearing or fluid where identifying individual points is hard. This motivates language that we will use throughout:

- A quantity whose possible values are in a finite or countably infinite set, or an object relating to such a quantity, we call **discrete**.
- A quantity whose possible values are in an uncountably infinite set, or an object relating to such a quantity, we call **continuous**.

While this distinction is profound at some level, for our purposes it will be mostly superficial. A few details will differ, mostly in how we actually calculate averages, but conceptually all the operations will be the same.

As a first step, we consider how we might think about what a Kind tree that has an infinite number of leaves would look like. Infinite trees are harder to work with in practice than finite trees, so we will work mostly with Kernels and Distributions, which are equivalent representations. Still, thinking about Kind trees in the infinite case will help us connect our familiar operations and see how to translate them into the Kernel and Distribution representations.

20.1 Infinite Kinds

A Kind tree has a values on its leaves and weights on its edges. Consider what such a tree would look like for a Kind with an infinite number of branches. Figure 20.1 shows two such trees, one countably infinite and the other uncountably infinite. As with their finite counterparts, these trees give a weight for each value, but unlike the finite case, some details are implicit. For instance, the countable tree on the left² has ... to show the continuing pattern of weights and values for branches that we cannot fit on the picture. The uncountable tree on the right has a continuous smearing of branches, so it represents the weight by the length of the branch and shows the values as a range. This is not as precise and not as easy to work with as in the finite case. The corresponding Kernels are shown below the trees.

Still, we can operate these trees with the Big 3+1. Figure 20.2 shows for the left (countable) tree a transform by a statistic, one subtree in a join with a conditional

² K_1 is not in canonical form; the factor $\frac{1}{3}$ in the Kernel comes from normalizing the weights.

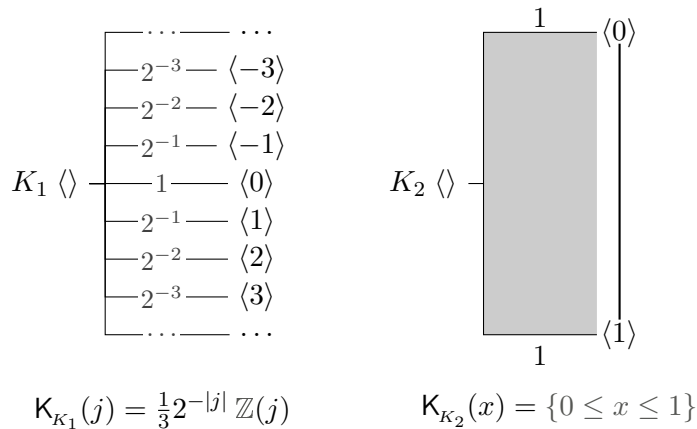


FIGURE 20.1. Kind trees with an infinite number of branches and the corresponding Kernels underneath. (Left) Countably infinite, and (Right) Uncountably infinite.

Kind, a observational constraint applied, and its expectation. All these operations are familiar. For instance, in transforming by a statistic, we map the values and then combine branches that map to a common value, adding their weights. In constraining with an observation, we simply erase all the branches that are inconsistent with the constraint. In building a join, we add the tree at the second stage (shown in the figure) to the leaf node of the source at the corresponding value. In predicting with an expectation, we take a weighted average of a statistic over the possible values and weighted by the weights.

For the uncountably infinite tree on the right of Figure 20.1, these operations are the same. Here though the uncountable infinities lead to a notable change – how we interpret the weights. This primarily impacts how we do the “combine branches, adding their weights” step during transformation with a statistic and will be discussed in detail in the next several sections. Figure 20.3 shows the trees for the analogous operations. For instance, there are more branches that map to small values under $\psi(x) = x^2$ than those that map to large values, which is why the weights accumulate near zero. For the moment, take the weights in the transformed tree on faith; we will see a nice explanation shortly.

The takeaway here is that our understanding of the Big 3+1 carries over to the infinite case. We will have to get some intuition about how to combine branches under the transform in the continuous case and about the interpretation of the weights. But these are details, and the structure our calculations is unchanged. We now turn to those unresolved questions.

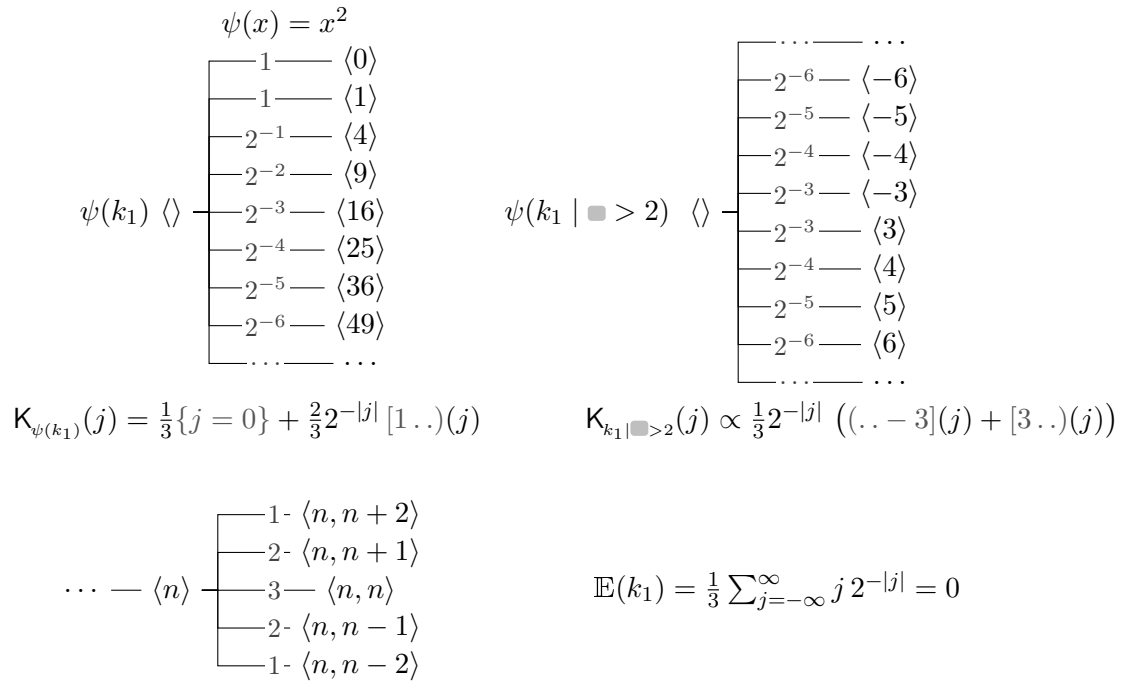


FIGURE 20.2. The Big 3+1 on the countable (left) Kind in Figure 20.1. (Top left) Transform by a statistic $\psi(x) = x^2$. (Top Right) The second-stage subtree attached to node n in a join with a conditional Kind. (Bottom left) Constrained with an observational. (Bottom right) Expectation.

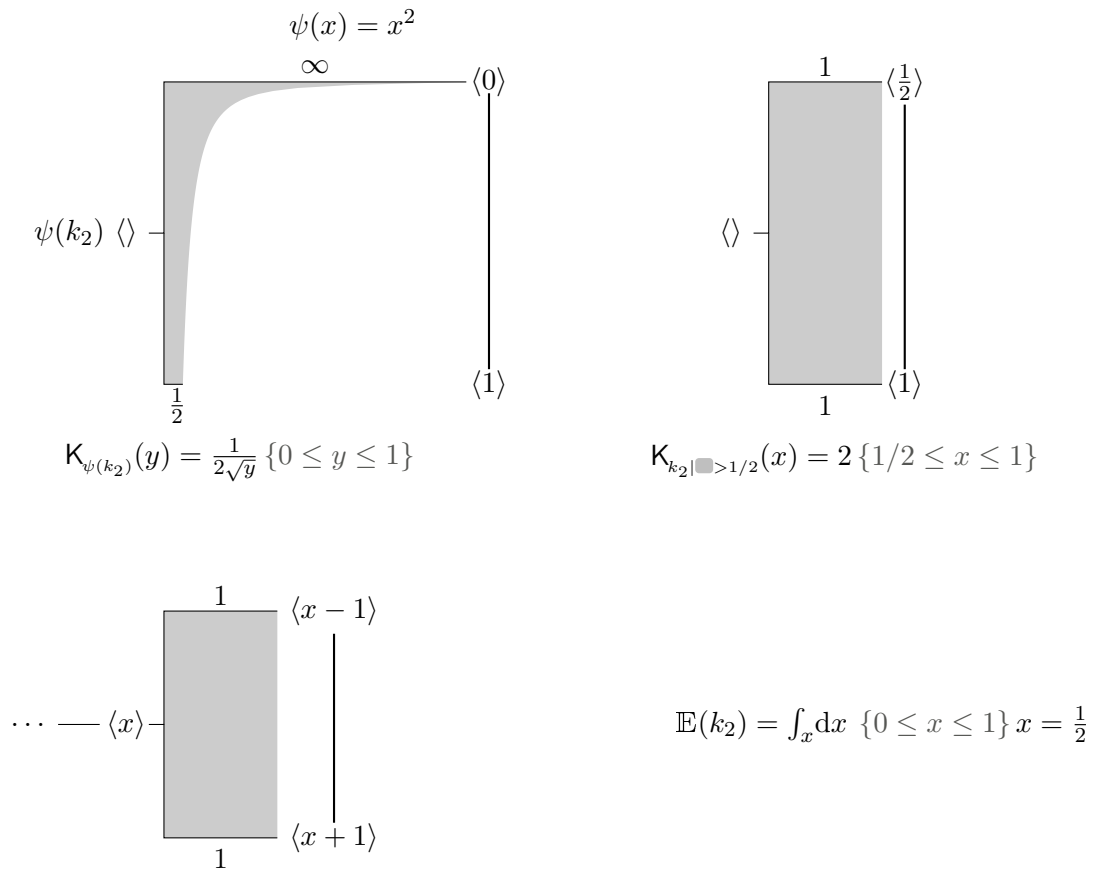


FIGURE 20.3. The Big 3+1 on the uncountable (right) Kind in Figure 20.1. (Top left) Transform by a statistic $\psi(x) = x^2$. (Top Right) The second-stage subtree attached to node x in a join with a conditional Kind. (Bottom left) Constrained with an observation. (Bottom right) Expectation.

20.2 Mass and Density

We start with a metaphor. We have 1 kilogram of stuff and some space to put it, like a line or a flat sheet or a sphere, with coordinates marked on it. There are two basic ways we can place the stuff. We can break off a discrete chunk of stuff, press it into a small particle, and place the particle *exactly* at one point, recording the coordinates of that point. (Think of it having a very sharp pin attached that lets us place it exactly.) *Or* we can atomize some of the stuff into a fluid and spray it around different locations, thicker in some places, thinner in others. Of course, we can do some of both types to distribute the 1kg of stuff across the space.

How do we measure how much stuff is at a location? In the first case, it's fairly straightforward. We look to see if there is a particle pinned to that spot, and if so, we measure its mass. To get the total mass in discrete particles, we do this over every location and sum up the particle masses.

In the second case, things are not as easy. The stuff is in a fluid spread over the space, like the air in a room or water in a lake. We cannot really grab a *single* molecule, and indeed if we look exactly at a given point we will almost certainly find no molecules there. All we can do is measure how much stuff is *near* a particular point. We take a small region around the target point and measure the mass of stuff in that region. We want the region to be small so that our measure reflects what is happening at our chosen point and not at nearby points. The smaller we make the region, the more representative our measure becomes, but the *mass* of stuff in the region vanishes. Instead of measuring the mass, however, we can measure the *density*, mass of stuff per unit volume (or length or area as appropriate) in the small region we choose. As the region gets smaller, the mass in the region approaches zero, but the density converges to a stable value. To get the mass in a small region near the location, we multiple the density by the size of the region (length for a line, area for a sheet, volume for a region, ...). To get the total mass, we do this at every location: density times the size of a very small (infinitesimal) region around the location. Density is in units of mass per unit size, so density times size is in units of mass. This gives us an infinitesimal mass at each location, which we add up over the continuum of points in the space (with an interval).

This distinction between placing discrete chunks of stuff and smearing a continuous spread of stuff holds a useful metaphor for the distinction in how we measure the concentration of discrete and continuous quantities. Discrete quantities are like the chunks of stuff. Their values are individuals that we can list out, label, and act on

them. And we can identify specific other nearby values. Continuous quantities are like a fluid. Their values form a continuum, a smearing, where we cannot identify individual nearby values, and we cannot count the values in any region.

Suppose we have a Kernel K_X . It has two types of values, which we call *discrete* and *continuous*. A possible value x can be a discrete value – an identifiable branch on the tree – or a continuous value, part of a continuum of possibilities.³

³More in Section 20.4.

- If x is a **discrete** value of X , we think of the weight $K_X(x)$ as giving a **mass**.
- If x is a **continuous** value of X , we think of the weight $K_X(x)$ as giving a **density**, mass per unit “volume.”

We call a Kind, Kernel, or Distribution discrete if all its possible values are discrete, and continuous if all its possible values are continuous. A continuous Kind is necessarily infinite.

Making this distinction between discrete and continuous values is the cost we pay for the convenience of allowing infinities into our models. The infinities make it easier to do some calculations that would otherwise be combinatorially complex, as we will see, but they are still infinite! Using continuous quantities introduces a few adjustments to our calculations, but overall, these are details. We will see that there is simple translation between the two cases that reveals a common structure in the operations and our analysis.

We allow Kinds to be unnormalized because it can be convenient and we can usually see the difference immediately. With Kernels and Distributions, however, it is harder to tell, so we insist that both remain in canonical form. Canonical form for a Kernel means that the **total mass equals 1**.

Definition 50. Total Mass. For any Kernel K_X , we define **total(K_X)** to be the total mass. When K_X is discrete,

$$\text{total}(K_X) = \sum_v K_X(v),$$

the sum of the weights over all values. When K_X is continuous, analogously,

$$\text{total}(K_X) = \int_v dv K_X(v),$$

just replacing $\sum_v$ with $\int_v dv$.

Again: we insist that the total mass equals 1. For discrete values, we just add up the mass (i.e., the weights), as we have been doing up to now. For continuous values, we need to convert the density to a mass by multiplying by a small unit of “volume” (dv above), and we add up all the small units of mass that result. Remember that *density times volume equals mass*. In the discrete case, we add up the weights with sums $\sum_x K_X(x)$, and in the continuous case, we add up the weights with integrals $\int_x dx K_X(x)$, where the dx represents an infinitesimal “volume” at x . We think of both $\sum_x$ and $\int_x dx$ as summation operations, and practically speaking, the difference between discrete and continuous calculations is just translating one to the other.

In both cases, we have the following useful notion:

Definition 51. Possible Values and Range. A value x is said to be a **possible value** of X if $K_X(x) > 0$. This is true whether x is a discrete or continuous value.

We denote the set of X ’s possible values by $\text{range}(X)$.

We will explore the parallels between discrete and continuous quantities here with some examples. The next two sections present the ideas more fully.

Example 20.1 Uniforms

Let U be an FRP with Kind `uniform(0, 0.01, 0.02, ..., 1)`. This Kind has Kernel

$$K_U(s) = \frac{1}{101} [0 \dots 100](100s).$$

The Indicator term is just a tricky way of saying that s belongs to the set $\{0, 0.01, \dots, 1\}$. The weights are equal for all 101 values, so they are $1/101$ for those values because we always keep Kernels in canonical form. These 101 values are the only possible values, so U is discrete.

Now, let W have Kernel $K_W(t) = \{0 \leq t \leq 1\}$. This is 1 when $t \in [0, 1]$ and 0 otherwise. Notice that every real number in the unit interval is a *possible value* of W , so W is continuous.

Suppose ψ is a statistic whose domain includes $[0, 1]$, so it is compatible with both U and W . To find $\mathbb{E}(\psi(U))$, we know just what to do:

$$\mathbb{E}(\psi(U)) = \sum_s K_U(s) \psi(s) = \frac{1}{101} \sum_{i=1}^{100} \psi(0.01i),$$

which is just the average of the values of ψ over the possible values of U .

To find $\mathbb{E}(\psi(W))$, we take the same type of average except we will replace $\sum_t$ by $\int_t dt$. This is because the integral is how we sum over uncountable sets.

$$\mathbb{E}(\psi(W)) = \int_t dt \, \mathbf{K}_W(t) \psi(t) = \int_{t=0}^1 dt \, \psi(t),$$

which is also an average. Observe the direct mapping between these two equations.

For example, if $\psi(x) = \{x > 3/5\}$,

$$\begin{aligned} \mathbb{E}(\psi(U)) &= \sum_i \mathbf{K}_U(i) \psi(i) \\ &= \frac{1}{101} \sum_{i=1}^{100} \{0.01i > 3/5\} \\ &= \frac{1}{101} \sum_{i=1}^{100} \{i > 60\} \\ &= \frac{1}{101} (100 - 60) = \frac{40}{101}, \end{aligned}$$

and

$$\begin{aligned} \mathbb{E}(\psi(W)) &= \int_t dt \, \mathbf{K}_W(t) \psi(t) \\ &= \int_{t=0}^1 dt \, \{t > 3/5\} \\ &= \int_{t=3/5}^1 dt \\ &= 1 - 3/5 = 2/5. \end{aligned}$$

The parallel is direct.

One small distinction to keep in mind for later is that if we had defined the statistic as $\tilde{\psi}(x) = \{x \geq 3/5\}$, this would give a *slightly different* $\mathbb{E}(\tilde{\psi}(U))$, but $\mathbb{E}(\tilde{\psi}(W))$ would be the same. The reason is that the point where $t = 3/5$ adds nothing to the integral but does add something to the sum. *Integrals are unchanged when we change their integrand at only a finite set of points.*

Example 20.2 What do you remember?

Start with a random quantity L with Distribution

$$D_L(\psi) = \sum_{j=1}^{\infty} p(1-p)^{j-1} \psi(j),$$

for some $p \in [0, 1]$. L 's possible values are all positive integers, which is a countable set. So all its values are discrete, and we say that L is discrete.

Let's find $\mathbb{E}(L)$ and $\mathbb{E}(\{L > 10\})$. These correspond to the statistics id and $\langle t \rangle \mapsto \{t > 10\}$.

$$\begin{aligned} D_L(\text{id}) &= \sum_j K_L(j) \text{id}(j) \\ &= \sum_{j=1}^{\infty} p(1-p)^{j-1} j = \frac{1}{p} \\ D_L(\langle t \rangle \mapsto \{t > 10\}) &= \sum_j K_L(j) \{j > 10\} \\ &= \sum_{j=1}^{\infty} p(1-p)^{j-1} \{j > 10\} \\ &= \sum_{j=11}^{\infty} p(1-p)^{j-1} \\ &= p \sum_{k=10}^{\infty} (1-p)^k = (1-p)^{10} \end{aligned}$$

Interlude C shows a derivation of these sums, but we won't be distracted with that here. One point though from the penultimate equality: we "changed variables" in the sum $j - 1 \rightarrow k$ with $j = 11$ corresponding to $k = 10$. Such shifts can be useful for simplifying sums like this.

Now, suppose we learn that $L > 15$, but we do not know anything else about L 's value. How do we update our knowledge? We can constrain with an observation with the condition $\xi(t) = \{t > 15\}$ to obtain $L \mid \xi$. We know from earlier that the observational constraint yields:

$$D_{L \mid \xi}(\psi) = \frac{D_L(\psi \cdot \xi)}{D_L(\xi)},$$

where the argument to D_L is the *product* of statistics ψ and ξ . We can use our

calculation above to find the denominator, just replacing 10 with 15:

$$D_L(\xi) = (1 - p)^{15}.$$

For the numerator, it's just plug and chug:

$$\begin{aligned} D_L(\psi\xi) &= \sum_j K_L(j) \psi(j) \xi(j) \\ &= \sum_{j=1}^{\infty} p(1-p)^{j-1} \psi(j) \xi(j) \\ &= \sum_{j=1}^{\infty} p(1-p)^{j-1} \psi(j) \{j > 15\} \\ &= \sum_{j=16}^{\infty} p(1-p)^{j-1} \psi(j). \end{aligned}$$

Putting it together:

$$\begin{aligned} D_{L|\xi}(\psi) &= (1-p)^{-15} \sum_{j=16}^{\infty} p(1-p)^{j-1} \psi(j) \\ &= (1-p)^{-15} \sum_{j=1}^{\infty} p(1-p)^{j+15-1} \psi(j+15) \\ &= \sum_{j=1}^{\infty} p(1-p)^{j-1} \psi(j+15) \\ &= D_L(\psi \circ (\square + 15)). \end{aligned}$$

This Distribution looks just like the Distribution of L except it is shifted larger by 15. Put another way: $(L - 15) | \xi$ has the same distribution as L .

Next, we consider an analogous situation with a continuous quantity. Let T have Distribution

$$D_T(\psi) = \int_u du \lambda e^{-\lambda u} \{u \geq 0\} \psi(u),$$

for some parameter $\lambda > 0$. Separating the statistic and the “summing” operator, this gives us that T 's Kernel is

$$K_T(u) = \lambda e^{-\lambda u} \{u \geq 0\}.$$

So,

$$\begin{aligned}
 D_T(\text{id}) &= \int_u du \, K_T(u) \, \text{id}(u) \\
 &= \int_u du \, \lambda e^{-\lambda u} \{u \geq 0\} u = \frac{1}{\lambda} \\
 D_T(\langle t \rangle \mapsto \{t > 10\}) &= \int_u du \, \lambda e^{-\lambda u} [0_-](u) \{u > 10\} \\
 &= \int_u du \, \lambda e^{-\lambda u} \{u > 10\} \\
 &= \int_{u=10}^{\infty} du \, \lambda e^{-\lambda u} = e^{-10\lambda}.
 \end{aligned}$$

As before, imagine that we observe only that $T > 15$, the Distribution of $T \mid T > 15$ tell us what we know about T given that information.

$$D_{T|\xi}(\psi) = \frac{D_T(\psi\xi)}{D_L(\xi)},$$

and using our calculation above

$$\begin{aligned}
 D_T(\xi) &= e^{-15\lambda} \\
 D_T(\psi\xi) &= \int_u du \, K_T(u) \, \psi(u)\xi(u) \\
 &= \int_u du \, \lambda e^{-\lambda u} \{u \geq 0\} \psi(u) \{u > 15\} \\
 &= \int_{u=15}^{\infty} du \, \lambda e^{-\lambda u} \psi(u) \\
 D_{T|\xi}(\psi) &= e^{15\lambda} \int_{u=15}^{\infty} du \, \lambda e^{-\lambda u} \psi(u) \\
 &= \int_{u=15}^{\infty} du \, \lambda e^{-\lambda(u-15)} \psi(u) \\
 &= \int_{v=0}^{\infty} dv \, \lambda e^{-\lambda v} \psi(v+15) \\
 &= D_T(\psi \circ (\blacksquare + 15)).
 \end{aligned}$$

The penultimate equality here just shifts the variable of integration by 15, and again, the result has the same form as the original but for a shift in the argument to the statistic. And indeed $(T - 15) \mid T > 15$ has the same Distribution as T , meaning that all our predictions are the same for both.

These parallels work in any dimension not just with scalars. When integrating in

higher dimensions we have a product of multiple infinitesimals. For instance, in the next example, we will think about $da\,db$ as a small unit of *area* and the Kernel K_P returning a density in the units of mass per unit area.

Example 20.3 Inner Circle

We pick a random point P inside the unit circle, i.e., within the region $\mathcal{C} = \{ \langle x, y \rangle \mid x^2 + y^2 \leq 1 \}$. The Kernel of P is given by

$$K_P(x, y) = \frac{1}{\pi} \mathcal{C}(x, y).$$

The factor of π is the *area* of the region inside the circle. Why area? Consider the total mass, which must be 1:

$$\begin{aligned} 1 &= \int_x dx \int_y dy \, K_P(x, y) \\ &= \int_x dx \int_y dy \, \frac{1}{\pi} \mathcal{C}(x, y) \\ &= \frac{1}{\pi} \int_x dx \int_y dy \, \mathcal{C}(x, y) \\ &= \frac{1}{\pi} \pi. \end{aligned}$$

The last integral a function that equals 1 inside the circle and 0 outside the circle gives the area of the circle. It “adds up” the continuum of *infinitesimal areas* $dx\,dy$ over all points inside the circle, yielding the total area.

What is the probability that P lies inside a circle of radius 1/2?

$$\begin{aligned} D_P \left(\langle x, y \rangle \mapsto \{x^2 + y^2 \leq 1/4\} \right) &= \int_x dx \int_y dy \, K_P(x, y) \{x^2 + y^2 \leq 1/4\} \\ &= \int_x dx \int_y dy \, \frac{1}{\pi} \mathcal{C}(x, y) \{x^2 + y^2 \leq 1/4\} \\ &= \frac{1}{\pi} \int_x dx \int_y dy \, \{x^2 + y^2 \leq 1/4\} \\ &= \frac{1}{\pi} \pi \left(\frac{1}{2} \right)^2 = \frac{1}{4}. \end{aligned}$$

The key step here comes in the third equation. The product of indicators $\mathcal{C}(x, y) \{x^2 + y^2 \leq 1/4\}$ is the indicator of the *logical-and* of the two conditions: the point is inside the unit circle *and* inside the circle of radius 1/2. But this is

| just the latter condition.

Remember that a conditional Kind takes a value and returns a Kind. Similarly, a conditional Kernel takes a value (after the given | bar) and returns a Kernel. If we have a conditional Kernel $K_{Y|X}(y | x)$, where X and Y can have any dimension, then this is a Kernel as a function of y . That is, for every possible x , i.e., with $K_X(x) > 0$,

$$\text{total}(K_{Y|X}(\blacksquare | x)) = 1$$

Example 20.4. A conditional Kernel is defined by

$$K_{B|A}(b | a) = \sum_{i=1}^3 \sum_{j=1}^2 c_a 2^{-(i+j)} \{b = j\} \{a = i\},$$

for some constant c_a , which may depend on the given a . What is c_a ?

We require that $\text{total}(K_{B|A}(\blacksquare | a)) = 1$ for $a \in [1..3]$. (For other a , the conditional Kernel is not well defined.) So,

$$1 = K_{B|A}(1 | a) + K_{B|A}(2 | a) = \frac{3}{4} c_a \sum_{i=1}^3 2^{-i} \{a = i\},$$

giving $c_1 = 8/3$, $c_2 = 16/3$, and $c_3 = 32/3$.

20.3 On Infinitesimals, Differentials, and Measure

Using continuous quantities compels us to deal with uncountably many values in our calculations. Increments become intervals, and sums become integrals for this reason. We think of integrals as sums in which we add up an (uncountably) infinite number of infinitesimal quantities. An integral like $\int_u du K_X(u) \psi(u)$ is an average of the values of $\psi(u)$ weighted by the infinitesimal masses $du K_X(u)$. The differential du is typically viewed as just a formal placeholder indicating the variable of integration, but there is both practical and conceptual value in taking the idea of an infinitesimal seriously.⁴

If x is a scalar value, we will treat an infinitesimal like dx as a type of scalar: a non-zero number that is *closer to zero than any finite number*. We call infinitesimals like dx that appear in our integrals, *differentials*. In the scalar case, we think of this as the *infinitesimal length* of an infinitesimal interval at the point x , like $[x\_x + dx)$. These infinitesimals are numbers with five “bonus features,” the first three of which

⁴And this can be made rigorous in several ways we won’t be concerned with.

are:

1. *Arithmetic Works.* We can add, subtract, multiply, and divide differentials as though they were numbers. In particular, if the same differential appears on opposite sides of an equation or in a numerator and denominator, *we can cancel them.*

We can also scale infinitesimals by regular numbers, like $4\,dx$ or $c\,du$. Indeed, our integrands scale the differentials in the integral, and it is these infinitesimal quantities that we are “adding up.”

2. *Higher-Order Terms Vanish.* For any scalar u , the differential at u is du

$$(du)^2 = 0. \quad (20.1)$$

This seems counter-intuitive at first. How can the square of a non-zero thing be zero. The idea behind this is that the differential du is an infinitesimally small length, so $(du)^2$ is infinitesimally smaller than infinitesimally small and thus *negligible*. More formally, du represents a limiting operation as a length goes to 0, so $(du)^2$ represents a quantity that is going to zero much faster and will thus vanish from the limit.

3. *Differentials Remember Their Identity.* We often have products of *different* differentials, like $dx\,dy$ or $da\,db\,dc$. These *do not vanish* because they each represent infinitesimal in a different dimension (along a different variable). Whereas we think of a term like dx as an infinitesimal length, terms like $dx\,dy$ and $da\,db\,dc$ represent infinitesimal areas and infinitesimal volumes.

In general, it is useful to have a catch-all term for the idea of length, area, and volume in general dimension.

Definition 52. We use **measure** to refer to the extent/capacity of space in any dimension. Thus, in 1 dimension, measure means length; in 2 dimensions, measure means area, in 3 dimensions, measure means volume, and so on.

Density, therefore, is the mass per unit measure. In 1 dimension, this is mass per unit length, in 2 dimension mass per unit area, in 3 dimensions mass per unit volume, and so on.

Differentials represent *infinitesimal units of measure*. If x, y, z are scalar variables, the differential dx is an infinitesimal length, the differential $dx\,dy$ is an infinitesimal

area, and $dx dy dz$ is an infinitesimal volume. This can continue to an arbitrary number of dimensions, and indeed if v is an n -dimensional tuple, we write dv for $dv_1 dv_2 \cdots dv_n$.

At continuous values, Kernels return a *density* in the units of *mass per unit measure*. The differentials like dx , $dx dy$, $dx dy dz$, and so on that we see in integrals are infinitesimal units of measure and products like $dx K_X(x)$, $dx dy K_{XY}(x, y)$, $dx dy dz K_{XYZ}(x, y, z)$, and so on have units of $\text{measure} \times \text{mass}/\text{measure} = \text{mass}$! Our integrals are thus just adding up or taking a weighted average with *infinitesimal pieces of mass*.

Notation. You will notice that we write our integrals “physicist style” with the differentials near the integral sign, e.g., $\int_x dx K_X(x) \psi(x)$. There are several good reasons for that, and I encourage you to follow this practice why you are learning.

When most of us learn calculus, integrals are written at the end of the integrand, like a delimiter. But we do not need a delimiter in practice. Integration is a summation operation, like $\sum$, and the ordinary order of operations disambiguates whether a term is inside or outside. Adding a term is outside

But why write integrals this way? Among the reasons:

- Integration is a continuous summation operation. We can think of it as summing up an uncountable number of infinitesimal values. An uncountable number of finite positive values will always sum to infinity, so this is essential to the idea of integration. Keeping the differential with the integration sign ties these two ideas together visually.
- Our operations and analysis with continuous and discrete operations are more than analogous, they are isomorphic. While there differences in the details of the calculations with integrals and sums, the operations themselves are connected by a simple translation between two types of summation operations: $\sum_x$ and $\int_x dx$.

Whenever we see $\sum_x$ in a discrete calculation, the high-level expression for the corresponding continuous calculation will look the same, just use $\int_x dx$ as the summation operation. This also works in reverse.

To emphasize this parallel, we will also write limits on our integrals just like we do on sums, e.g., $\int_{x=a}^b dx$ analogous to $\sum_{x=a}^{b-1}$.

- We will always write our expectations with three pieces, in order: summation operation, Kernel, statistic. By keeping the differential near the

Kernel, we emphasize visually that we are adding up infinitesimal pieces of mass $dx K_X(x)$.

We will see the benefits of this approach as we proceed.

To illustrate these ideas, we will consider two examples, applying a projection statistic and the transformation in Figure 20.3 with the statistic $\psi(x) = x^2$ that gave an apparently odd result of a Kernel whose value grows arbitrarily large. This will lead us to the last two features of infinitesimals.

First, consider the 3-dimensional S with $A = \text{proj}_1(S)$, and assume that S is continuous. We want to find the Kernel of A from the Kernel of S . For any statistic φ compatible with A , $\varphi(A) = \varphi(\text{proj}_1(S))$, so

$$\begin{aligned} D_A(\varphi) &= D_S(\varphi \circ \text{proj}_1) \\ &= \int_a da \int_b db \int_c dc K_S(a, b, c) \varphi(\text{proj}_1(a, b, c)) \\ &= \int_a da \int_b db \int_c dc K_S(a, b, c) \varphi(a) \\ &= \int_a da \left(\int_b db \int_c dc K_S(a, b, c) \right) \varphi(a). \end{aligned}$$

It follows that

$$K_A(a) = \int_b db \int_c dc K_S(a, b, c),$$

where we simply “integrate out” the other components. The same logic works for any projection. For instance, letting $\langle A, C \rangle = \text{proj}_{1,3}(S)$, try to demonstrate that $K_{AC}(a, c) = \int_b db K_S(a, b, c)$. The argument is almost exactly the same.

Somewhat informally we can view the previous argument as combining/branches adding weights like we would with a sum:

$$\int_a da \int_b db \int_c dc K_S(a, b, c) \{\text{proj}_1(a, b, c) = x\} = dx \int_b db \int_c dc K_S(x, b, c) = dx K_A(x).$$

In the first equality, the condition selects the slice of the integral at x , leaving an infinitesimal mass. But this is just the mass we would get by adding masses over all values of S that map to the value x of A , which is the infinitesimal mass $dx K_A(x)$. Canceling the infinitesimals,⁵ we get the same K_A as above. While this argument is less rigorous, it shows that the calculation of the projection Kernel K_A just comes from combining branches/adding weights as before.

Second, we return to the example in Figure 20.3, where we transformed an X with Kernel $K_X(t) = \{0 \leq t \leq 1\}$ with the statistic $\psi(x) = x^2$. This seemed to give a

⁵The key rule to keep in mind is that no result we want to use in the world can end up with infinitesimals in it.

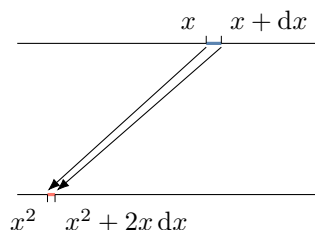
Kernel that blew up towards infinity near the value 0. To explain this, we want to find the Kernel or Distribution of the transformed $\psi(X)$.

As we have done many times, consider at the steps in transforming a Kind or Kernel. We map each possible value to a transformed value with the statistic and then combine/add the weights for all values in the original that map to the same transformed value. The weight on the transformed value thus adds up over all the ways that value can be produced by transforming the original values.

At a continuous value x , we are dealing with a continuum of possible values nearby. Think of this as an infinitesimal interval from x to $x + dx$. When we transform these values by ψ , that interval maps to the interval from $\psi(x)$ to $\psi(x + dx)$, with

$$\psi(x + dx) = (x + dx)^2 = x^2 + 2 dx x + (dx)^2 = x^2 + dx 2x,$$

using the negligibility of higher-order infinitesimals. This looks like:



Notice that the interval has shrunk after the transformation. (It's still infinitesimal but scaled to be smaller.) So the original mass in the interval from x to $x + dx$ has now been concentrated to a *higher density*. The density is mass per unit measure (length in this case), which is

$$K_Y(x^2) = \frac{dx K_X(x)}{dx 2x} = \frac{1}{2x} K_X(x),$$

using our ability to cancel infinitesimals. The density is growing without bound as x gets smaller, as we saw in in Figure 20.3. Taking $y = x^2$ and $x = \sqrt{y}$, this becomes

$$K_Y(y) = \frac{1}{2\sqrt{y}} K_X(\sqrt{y}),$$

which gives us K_Y in terms of K_X , as desired.

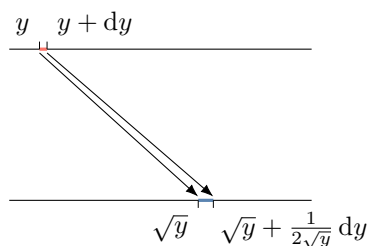
In practice, it is often cleaner to do this argument in reverse, starting from an infinitesimal interval around some possible value y of Y . We will get the same answer, but it is somewhat more direct. That is actually what we did when transforming of

Kinds and Kernels, referring to the weight of the *transformed value*. So let's turn the previous argument around and map from the values of Y to the values of X . Here, define the inverse function $\psi^-(y) = \sqrt{y}$ which exists in this case, at least over the possible of X and Y . So, $Y = \psi(X)$ and equivalently $X = \psi^-(Y)$. Now, we look at the infinitesimal interval from y to $y + dy$; this transforms to an interval from $\psi^-(y)$ to $\psi^-(y + dy)$ ⁶

⁶ f^- is the *inverse image* of f . See Section 10.1.

$$\psi^-(y + dy) = \sqrt{y + dy} \approx \sqrt{y} + \frac{1}{2\sqrt{y}} dy + (\dots)(dy)^2 = \sqrt{y} + \frac{1}{2\sqrt{y}} dy,$$

using the fact that higher-order infinitesimals are zero. The above picture becomes



The mass $dy K_Y(y)$ in the interval from y to $y + dy$ is the same as the mass $\frac{1}{2\sqrt{y}} dy K_X(\sqrt{y})$ in the interval from $\sqrt{y}$ to $\sqrt{y} + \frac{1}{2\sqrt{y}} dy$, and so the density is, dividing by dy

$$K_Y(y) = \frac{1}{2\sqrt{y}} K_X(\sqrt{y}).$$

Thus, because Kernels at continuous values measure *density*, we need to take into account what is happening at nearby values when we transform those Kernels with a statistic. The process is still fundamentally the same – adding up the weights of values that map to a transformed value – but the local behavior of the statistic (i.e., its derivative) determines how a small neighborhood of the value stretches and shrinks and thus affects the density. Techniques for computing Kernels of transforms with statistics are explored in detail Chapter 25.

The last two features of infinitesimals codify these ideas.

4. *The Chain Rule Applies.* We sometimes need to take the differential of a function evaluated at a point, such as $dh(u)$. In this case, all the usual rules about differentials from calculus apply.

In particular, if h is a real-valued function of a single variable, then

$$dh(u) = du h'(u), \quad (20.2)$$

where h' is the derivative of h . An easy way to remember this is that dividing both sides by du gives the tautology $\frac{dh(u)}{du} = h'(u)$. (The connection to the chain rule comes if u were a function of another variable t , then dividing equation (20.2) by dt gives the chain rule $\frac{dh(u(t))}{dt} = \frac{dh}{du} \frac{du}{dt}$.)

If h is function that takes an m -tuple and returns a scalar, then

$$dh(u_1, u_2, \dots, u_m) = \sum_{i=1}^m du_i \frac{\partial h}{\partial u_i}(u_1, u_2, \dots, u_m). \quad (20.3)$$

This is again just the chain rule in disguise.

5. *Change of Variables Acts on Differentials.* The chain rule tells us how to transform differentials of one variable to differentials of another. If h is a differentiable function from scalars to scalars and writing $y = h(u)$, equation (20.2) gives us dy in terms of du .⁷ Then if $h'(u) \neq 0$ on $[a, b]$

$$\int_{y=h(a)}^{h(b)} dy \psi(y) = \int_{u=a}^b du h'(u) \psi(h(u)), \quad (20.4)$$

⁷If $h' < 0$, $h(a) > h(b)$, negate both sides, reversing the limits of integration on the left and using $|h'(u)|$ on the right.

where we simply replace y with $h(u)$ and dy with $du h'(u)$.

This works in higher dimensions too. If a function h maps m -tuples to m -tuples (i.e., $h: \mathbb{R}^m \rightarrow \mathbb{R}^m$), then we can write $h = (h_1, \dots, h_m)$ where each component function h_i takes an m -tuple and returns a number. The⁸ derivative matrix of h , which we denote by h' , is defined by

⁸This matrix is also sometimes denoted by Dh .

$$h'(u_1, \dots, u_m) = \begin{bmatrix} \frac{\partial h_1}{\partial u_1}(u_1, \dots, u_m) & \cdots & \frac{\partial h_1}{\partial u_m}(u_1, \dots, u_m) \\ \vdots & \ddots & \vdots \\ \frac{\partial h_m}{\partial u_1}(u_1, \dots, u_m) & \cdots & \frac{\partial h_m}{\partial u_m}(u_1, \dots, u_m) \end{bmatrix} \quad (20.5)$$

This takes an m -tuple and returns a matrix of partial derivatives at that point. The first row contains all the partial derivatives of h_1 with respect to each parameter, the second row all the partial derivatives of h_2 , and so on, all in order from u_1 through u_m .

The **Jacobian** of h is the absolute value of the determinant of h' , $|\det h'|$, as a function of $u_1, \dots, u_m$, where $\det M$ denotes the determinant of a matrix M .

This gives us the general **change of variables formula**. If $y_1 = h_1(u_1, \dots, u_m)$,

$y_2 = h_2(u_1, \dots, u_m)$, and so on through $y_m = h_m(u_1, \dots, u_m)$, then

$$dy_1 dy_2 \cdots dy_m = du_1 du_2 \cdots du_m |\det h'(u_1, \dots, u_m)|, \quad (20.6)$$

which we can simply plug into to integrals as

$$\begin{aligned} \int \cdots \int_{h^{-1}(\mathcal{A})} dy_1 dy_2 \cdots dy_m \psi(y_1, \dots, y_m) \\ = \int \cdots \int_{\mathcal{A}} du_1 du_2 \cdots du_m |\det h'(u_1, \dots, u_m)| \psi(h(u_1, \dots, u_m)). \end{aligned} \quad (20.7)$$

Here, $\mathcal{A}$ is a region of integration over which the Jacobian is non-zero, and $h^{-1}(\mathcal{A})$ is the region $\mathcal{A}$ maps to with h .

Example 20.5 Cartesian to Polar Coordinates

If we are integrating over a circle or disk, it is often easier to change from Cartesian coordinates $\langle x, y \rangle$ to use *polar coordinates* $\langle r, \theta \rangle$ that describe a point by its distance from the origin $r \geq 0$ and its angle θ from the positive x -direction.

The mapping gives

$$\begin{aligned} x &= r \cos(\theta) \\ y &= r \sin(\theta) \end{aligned}$$

which has derivative matrix

$$\begin{bmatrix} \cos(\theta) & -r \sin(\theta) \\ \sin(\theta) & r \cos(\theta) \end{bmatrix}$$

and Jacobian $r \cos^2(\theta) - r \sin^2(\theta) = r$.

So, we replace $dx dy$ by $d\theta dr r$ in our integral.

Example 20.6 Differential Shifts

Suppose we want to compute

$$\int_{a=0}^1 \int_{b=0}^1 db da \varphi(a+b),$$

where we want to replace $a + b$ with a variable s .

We can do the change of variables from a, b to r, s directly, replacing $r = a$ and $s = a + b$.

The derivative matrix of the transformation is

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

which has determinant 1. So, $dr ds = da db$. Under this transformation, the unit square we are integrating over maps to a parallelogram, giving

$$\int_{r=0}^1 \int_{s=r}^{1+r} dr ds \varphi(r) = \int_{a=0}^1 \int_{b=0}^1 db da \varphi(a+b).$$

Another way to see the change of variables in this case is to note that $dr = da$ and $ds = da + b = da + db$, which gives

$$dr ds = da (da + db) = (da)^2 + da db = da db.$$

Reminder. Remember that an integral is not changed by changing the integrand at only a finite number of points. Indeed, an integral over points of dimension n is not changed by changing the integrand on a set of dimension $m < n$. So, for example,

$$\int_x dx f(x) \{x > 0\} = \int_x dx f(x) \{x \geq 0\}.$$

Changing the indicator to exclude or include 0 has no effect on the result.

20.4 Distributions and Kernels Revisited

For the vast majority of the Kernels we will use, their possible values are *either* discrete or continuous. At a discrete value, there is an identifiable chunk of “mass” (i.e., probability) at that value, the Kernel evaluated at that value returns the probability at that value. For a continuous value, there is a smearing of probability mass over a neighborhood of the value. We cannot identify a distinct chunk of mass at that value, but we can measure the density in an infinitesimal neighborhood *near* that value. The Kernel then returns the (probability) density, and when multiplied by an infinitesimal measure gives an infinitesimal (probability) mass. Kernels with this

property are called **simple**.

Definition 53. A Kernel is a **simple kernel** if every possible value can be classified as either a *discrete value*, where the Kernel returns a mass, or a *continuous value*, where the Kernel returns a density.

If X is a random variable that has a simple kernel, then we can partition the possible values of X into discrete and continuous parts. The set of possible values is $\text{range}(X)$, and we write $\text{range}_d(X)$ and $\text{range}_c(X)$ for the discrete and continuous parts.

For values in $\text{range}_d(X)$, $K_X(v)$ returns a mass; for values in $\text{range}_c(X)$, $K_X(v)$ returns a density. We can therefore decompose K_X into its **discrete part** and its **continuous part** as follows:

$$K_X(x) = K_X(x) \{x \in \text{range}_d(X)\} + K_X(x) \{x \in \text{range}_c(X)\} \quad (20.8)$$

$$:= p_X(x) + f_X(x). \quad (20.9)$$

The second equation *defines* two functions, p_X and f_X ; we call

- p_X the probability **mass function** of K_X , and
- f_X the probability **density function** of K_X .

We define $p_X(v) = 0$ for every $v \notin \text{range}_d(X)$ and $f_X(v) = 0$ for every $v \notin \text{range}_c(X)$.

There are two common special cases. If $\text{range}_c(X)$ is empty, then $f_X(v) \equiv 0$ and we say that K_X and X are **discrete**. If $\text{range}_d(X)$ is empty, then $p_X(v) \equiv 0$ and we say that K_X and X are **continuous**.

The distinction between discrete and continuous parts is essentially important only for determining how we calculate expectations. Specifically, it tells us how we “add things up” in our weighted averages, with sums or integrals, and over which values. The distinction has a few practical implications, but for the most part it is a detail of calculation. The conceptual structure of the operations is the same whether a random variable is discrete, continuous, or a combination of both.

When we define the Kernel, the discrete part is almost always a function with the property that in a sufficiently small region around (and excluding) a possible value, the mass function is zero. What is left over is the continuous part. This is usually easy to see, but it can get tricky when the discrete values and the continuous values are contiguous. Because integrals are not changed by adjusting integrands at a finite

set of values, we have some flexibility in defining the density function at boundaries or points near the set of discrete values.

Example 20.7. What are the possible values of each Kernel, and which are continuous and which discrete?

1. $K_X = \frac{1}{10} [1..10]$
2. $K_Y(v) = \frac{1}{2} \{-2 \leq v \leq -1\} + \frac{1}{2} \{1 \leq v \leq 2\}$
3. $K_A = \frac{1}{3} \{0\} + \frac{2}{27} [1..10]$
4. $K_B(v) = \frac{1}{3} \{v = 0\} + \frac{2}{27} \{0 < v \leq 9\}$

In order, the answers are: (1) integers from 1 to 10, all discrete values, no continuous values, so $p_X = K_X$ and $f_X \equiv 0$; (2) real numbers between -2 and -1 and between 1 and 2, all continuous values, no discrete values, so $f_Y = K_Y$ and $p_Y \equiv 0$; (3) the value 0 (discrete) and real numbers from 1 to 10 (continuous); (4) the value 0 (discrete) and real numbers from 0 to 9 (continuous).

It is not really necessary to ever use the mass and density functions in our calculations, as they are just synonyms for the kernel. But when a kind is wholly discrete or continuous using the synonyms clearly signals the type.

Example 20.8. Let Z have Kernel

$$K_Z(t) = \lambda e^{-\lambda t} \{t \geq 0\}. \quad (20.10)$$

for some fixed number $\lambda > 0$. The possible values of Z are all non-negative real numbers. Define statistics:

- $\psi(s) = \lfloor s \rfloor$, which for any s returns the largest integer $\leq s$, and
- $\varphi(s) = s \{s \geq 1\}$.

Define $N = \psi(Z)$ and $M = \varphi(Z)$. Find the Kernels or Distributions of N and M . (We will handle M in the next example.)

Notice first that N is discrete with possible values $[0..)$. To find $K_N(n)$ we add up the weights of all the branches s for which $\psi(s) = n$. With n a natural number:

$$K_N(n) = \int_s ds K_Z(s) \{\psi(s) = n\} \quad (20.11)$$

$$= \int_{s=n}^{n+1} ds \, K_Z(s) \quad (20.12)$$

$$= \int_{s=n}^{n+1} ds \, \lambda e^{-\lambda s} \quad (20.13)$$

$$= \int_{t=\lambda n}^{\lambda(n+1)} dt \, e^{-t} \quad (20.14)$$

$$= (-e^{-t}) \Big|_{t=\lambda n}^{\lambda(n+1)} \quad (20.15)$$

$$= e^{-\lambda n} - e^{-\lambda(n+1)} \quad (20.16)$$

$$= e^{-\lambda n} (1 - e^{-\lambda}). \quad (20.17)$$

Hence, in general

$$K_N(n) = e^{-\lambda n} (1 - e^{-\lambda}) \{n \geq 0\}. \quad (20.18)$$

Notice that this looks like our answer to the Waiting for Heads example in Chapter 8 taking $p = 1 - e^{-\lambda}$ and shifting n by 1.

So far, all the Kernels we have seen have been simple Kernels that were wholly discrete or wholly continuous. But the continuation of the previous example with M shows an example with points of different types.

Example 20.9. Continuing the previous example, we now consider M , using the same idea. We add up the weights of all branches for which φ has a common value. If $s \geq 1$, $\varphi(s) = s$ and no other value of Z produces this output, so $K_M(s) = s$. If $0 \leq s < 1$, $\varphi(s) = 0$, so all these branches map to zero. The possible values of M are thus 0 and $[1_+)$. $M = 0$ if and only if $0 \leq L < 1$, and equivalent conditions imply equal events. So, $\mathbb{E}(\{M = 0\}) = \mathbb{E}(\{0 \leq L < 1\})$, and thus

$$K_M(0) = \int_s ds \, K_Z(s) \{0 \leq s < 1\} \quad (20.19)$$

$$= \int_{s=0}^1 ds \, \lambda e^{-\lambda s} \quad (20.20)$$

$$= 1 - e^{-\lambda}. \quad (20.21)$$

Putting this together:

$$K_M(s) = 1 - e^{-\lambda} \{0\}(s) + \lambda e^{-\lambda s} [1_-](s). \quad (20.22)$$

M has 0 as a discrete value and $[1_-]$ as continuous values. Notice that

$$\text{total}(K_M) = (1 - e^{-\lambda}) + \int_{s=1}^{\infty} ds \lambda e^{-\lambda s} = 1. \quad (20.23)$$

One other type of Kernel arises with some regularity in practice: Kernels whose values have both discrete and continuous components. We call these *split kernels*.

Example 20.10. Let FRP C represent the flip of a coin with Kind `binary(p)` for some $p \in [0_+1]$.

If $C = 0$ (tails), we generate a random value with Kernel

$$K_{T|C}(t | 0) = (1 - t/2) \{0 \leq t \leq 2\}.$$

If $C = 1$ (heads), we generate a random value with Kernel

$$K_{T|C}(t | 1) = \frac{1}{4}(1 - t/8) \{0 \leq t \leq 8\}.$$

The join $\langle C, T \rangle = C \triangleright (T | C)$ has Kernel

$$K_{CT}(c, t) = (1 - p)(1 - t/2) \{0 \leq t \leq 2\} \{c = 0\} + \frac{1}{4}p(1 - t/8) \{0 \leq t \leq 8\} c = 1$$

This is discrete in the C component and continuous in the T component.

Fortunately, that poses no problem. For instance:

$$\mathbb{E}(\psi(C, T)) = \sum_c \int_t dt K_{CT}(c, t) \psi(c, t)$$

and

$$\begin{aligned}
\mathbb{E}(\varphi(T)) &= D_{CT}(\varphi \circ \text{proj}_2) \\
&= \sum_c \int_t dt \, K_{CT}(c, t) \varphi(\text{proj}_2(c, t)) \\
&= \sum_c \int_t dt \, K_{CT}(c, t) \varphi(t) \\
&= \int_t dt \sum_c K_{CT}(c, t) \varphi(t) \\
&= \int_t dt \, (K_{CT}(0, t) + K_{CT}(1, t)) \varphi(t) \\
&= \int_t dt \left((1-p)(1-t/2) \{0 \leq t \leq 2\} + \frac{1}{4}p(1-t/8) \{0 \leq t \leq 8\} \right) \varphi(t).
\end{aligned}$$

Hence,

$$K_T(t) = (1-p)(1-t/2) \{0 \leq t \leq 2\} + \frac{1}{4}p(1-t/8) \{0 \leq t \leq 8\}.$$

We sum over the first component and integrate over the second. The result may look a bit ugly, but we just turned the crank.

Definition 54. X with $\dim(X) > 1$ has a **split kernel** when some of its components are discrete and some are continuous. The i th component is discrete or continuous if the scalar $\text{proj}_i(X)$ is discrete or continuous.

20.5 FRPs, Random Processes, and Random Variables

We have now seen random quantities with *infinite Kinds*. These describe more general Random Processes (RPs) and

- can have an **infinite number of possible values**, either “countably” or “uncountably” infinite;
- can occur over an **infinite horizon of time and space**; and
- can have **infinite dimension**.

FRPs generate a random value once, and keep it for all time. We can operate on fresh FRPs without knowing their values because we have their Kind that embodies

our *knowledge* of the FRPs value. We can describe the Kind by the tree or by its Kernel or Distribution.

FRPs are models for *Finite* Random Processes, systems that can produce a finite number of possible values over a finite horizon of time and space. In Part I, we were able to push this constraint and use FRPs to solve problems that were ostensibly infinite.

To expand our description to systems that

- can have an **infinite number of possible values**, either “countably” or “uncountably” infinite,
- can occur over an **infinite horizon of time and space**, or
- can have **infinite dimension**,

we broaden our view to include general Random Processes (RPs). We call a general RP a **random variable**. **Every FRP is a random variable, but a random variable might not be an FRP if its Kind is infinite.**

Random variables share most of the features of an FRP. We think of them, denote them, and operate on them in the same way. Like FRPs, a general random variable is also fresh until activated (observed) and it also produces one value, one time. We operate on the random variable before observing it and derive our predictions about its value. The Big 3+1 remain the same, in their interpretation and action: transforming with statistics, building with joins, constraining with observations, and predicting with expectations.

Random variables (including FRPs) represent the random quantities that we observe and model. Our schema extends seamlessly to Random Variables; see Figure 20.4. For instance, the “data random variable” in that schema represents a random quantity whose value, when activated, captures all the data that we might collect to answer our question. Using this “data” random variable, we transform with statistics, build with joins, constrain with observations, and predict with expectations to answer our questions.

The infinities that RPs allow mean that it is frequently more convenient to describe our knowledge of the system with Kernels and Distributions than with trees. The introduction of uncountable infinities also adjusts some of the details of the calculations (e.g., integrals versus sums) and broadens our interpretation of the Kernels (e.g., mass versus density). Conceptually though, little will change.

The infinities will add some complexity to our calculations but at the same time increases the scope and flexibility of our modeling. Random variables represent the

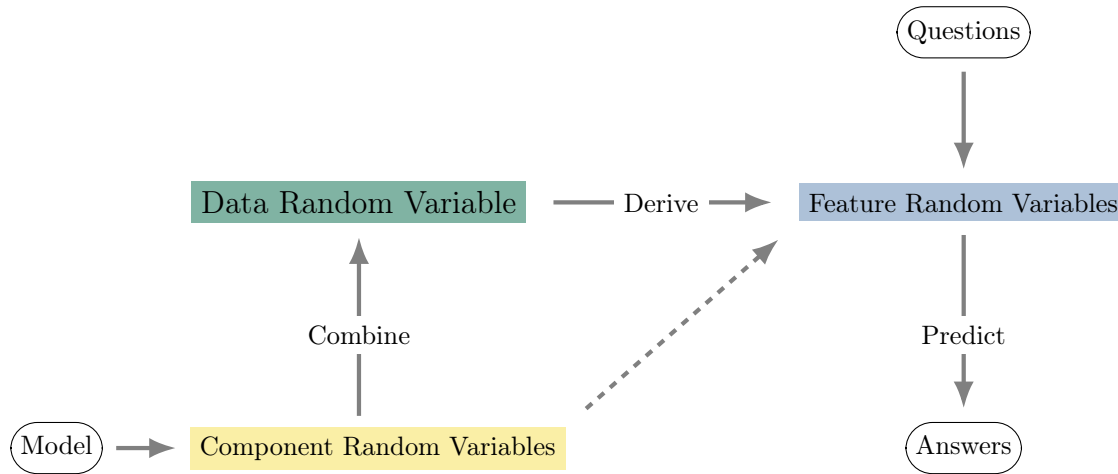


FIGURE 20.4. Update to Figure 2.8 that shows the role of the model, motivating questions, and predictions of their answers, using general random variables in lieu of FRPs. The dashed arrow indicates that some feature random variables may be defined in terms of selected components and thus may be activated before the data random variable is.

random quantities that we observe and model. Their Distributions (and Kernels and Kinds) describe our knowledge about their values at any point. We will use the same pattern language for building systems, like transformation $\psi(X)$, joins $X \mid (Y \mid X)$, observations $X \mid \xi$, and expectations $\mathbb{E}(\psi(X) \mid \xi)$. Just like with FRPs, the Big 3 applied to random variables produce new random variables. So, if X is a random variable, ψ a compatible statistic, and ξ a compatible condition, all of these are random variables: transforming by $\psi(X)$, constraining with an observation $X \mid \xi$, and forming a join $X \mid (Y \mid X)$. Do not lose sight of the fact that, though our expressions of these operations may vary, the operations themselves, their meaning, and high-level implementation *have not changed*.

Case Study: Symmetry and the Uniform Distributions

21

Chapter

Contents

21.1 Symmetry for Finite Sets	902
21.2 Symmetry for Continuous Regions	910

Key Take Aways

A system has symmetry when some feature of the system does not change when it is transformed in particular ways. A sphere is symmetric under rotations, for instance. For random systems, an assumption of symmetry posits that particular transformations of the system *do not change our predictions*.

If X is a random variable with finite range $\mathcal{V}$, we say that the Distribution of X is a **DiscreteUniform** $\langle\mathcal{V}\rangle$ if all of our predictions about X remain unchanged when we transform it by a permutation of $\mathcal{V}$.

This means: if ρ is any permutation of $\mathcal{V}$ and ψ is any compatible statistic,

$$D_X(\psi \circ \rho) = D_X(\psi), \quad (21.1)$$

or equivalently,

$$\mathbb{E}(\psi(\rho(X))) = \mathbb{E}(\psi(X)). \quad (21.2)$$

It follows that

$$D_X(\psi) = \frac{1}{\#\mathcal{V}} \sum_{i=1}^{\#\mathcal{V}} \psi(v_i), \quad (21.3)$$

so X 's Kernel is

$$K_X(v) = \frac{\mathcal{V}(v)}{\#\mathcal{V}}. \quad (21.4)$$

This is *constant* on $\mathcal{V}$, making all values in $\mathcal{V}$ equally likely. We often say in this case that X is “a **DiscreteUniform** $\langle\mathcal{V}\rangle$ random variable.”

Moreover, if X has a $\text{DiscreteUniform}\langle\mathcal{V}\rangle$ distribution and ξ is a compatible condition, then $X \mid \xi$ is *also uniform*; its distribution is $\text{DiscreteUniform}\langle\mathcal{V}_1\rangle$, where

$$\mathcal{V}_1 = \{v \in \mathcal{V} \mid \xi(v) = 1\}, \quad (21.5)$$

the subset of $\mathcal{V}$ that is consistent with the condition.

We can define an analogous notion of symmetry for *regions* in $\mathbb{R}^n$, contiguous open sets with a boundary of smooth pieces, like a ball or triangle. The *measure* of region quantifies the region's extent or capacity. When $n = 1$, the measure is *length*; when $n = 2$, the measure is *area*; and when $n = 3$, the measure is *volume*. In general, think of measure as an n -dimensional “volume.”

A *measure-preserving map* on a region, analogous to a permutation, rearranges the points in the region without changing the measure of any subset – no stretching or shrinking.

If X is a random variable whose range is a region $\mathcal{R}$, and thus continuous, we say that X has a **Uniform** $\langle\mathcal{R}\rangle$ distribution if our predictions about X do not change under a measure-preserving map of $\mathcal{R}$. Specifically, if f is such a function and ψ is any X -compatible statistic,

$$D_X(\psi \circ f) = D_X(\psi), \quad (21.6)$$

or equivalently,

$$\mathbb{E}(\psi(f(X))) = \mathbb{E}(\psi(X)). \quad (21.7)$$

It follows that

$$D_X(\psi) = \frac{1}{\text{measure}(\mathcal{R})} \int_{\mathcal{R}} dx \psi(x)$$

$$K_X(u) = \frac{\mathcal{R}(u)}{\text{measure}(\mathcal{R})}.$$

The Kernel is constant on $\mathcal{R}$, meaning that the probability is spread evenly, with constant density across the region. In this case, we often say that X is “a **Uniform** $\langle\mathcal{R}\rangle$ random variable.”

Moreover, if ξ is a compatible condition, then $X \mid \xi$ is also uniform; it's

distribution is $\text{Uniform}(\mathcal{R}_1)$, where

$$\mathcal{R}_1 = \{u \in \mathcal{R} \mid \xi(u) = 1\}, \quad (21.8)$$

the subset of $\mathcal{R}$ that is consistent with the condition.

In this Chapter, we look one of the most basic and intuitive modeling assumptions – symmetry. A symmetry assumption posits that under certain transformations of the system, *our predictions do not change*. Let’s motivate this with a simple and concrete example.

Example 21.1. I offer you the following game. I will select a random mechanism to generate two monetary rewards (one positive, one negative) and offer one to you, keeping the other for myself. However, before you accept the reward, you have the option of swapping them (you get mine and vice versa). We both know the mechanism I use to generate the rewards and can make predictions about our payoff from this game, and if the predictions are unfavorable for either us, we will not play. What random mechanism can I use that would make us both willing to play?

Call the rewards A and B , where you get A if you do not swap. If we predict that $\mathbb{E}(A) > 0$, you would not swap, and I will not play. If we predict that $\mathbb{E}(A) < 0$, you would swap, and I will not play.

If, however, $\mathbb{E}(A) = 0$, you would be indifferent to swapping, and both of us would be willing to play. Another way to put this is in this case, our predictions about A and B do not change if you swap. This is *symmetry* between the two rewards.

Another example illustrates the kinds of transformations we are talking about.

Example 21.2. A random point P is chosen in the unit disk

$$\mathcal{D} = \{\langle x, y \rangle \mid x^2 + y^2 \leq 1\}.$$

Pick a rotation ψ of the disk, and let $P' = \psi(P)$ be the position of P after rotating. If P and P' have the same Distribution, our predictions were not changed by the extra spin of the disk. If that is true for *any* rotation, it tells us that our predictions do not depend on the orientation of the disk; they are

symmetric under rotations.

In general, a symmetry assumption posits that the Distribution of $f(X)$ is the same as the Distribution of X for all functions f in some specified set $\mathcal{F}$ of transformations $\text{range}(X) \rightarrow \text{range}(X)$. We can use equation (19.42) to express the Distribution of the transformed random variable $f(X)$ in terms of D_X .

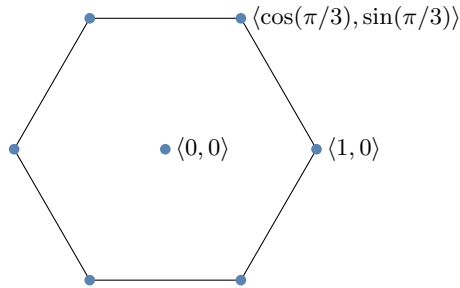
Definition 55. Let $\mathcal{V}$ be a set of values and $\mathcal{F}$ be a collection of functions $\mathcal{V} \rightarrow \mathcal{V}$. A random variable X with $\mathcal{V} = \text{range}(X)$ has a Distribution that is `[distribution!symmetry]symmetric` with respect to $\mathcal{F}$ if

$$D_X(\psi \circ f) = D_X(\psi) \text{ for all } f \in \mathcal{F} \quad (21.9)$$

for all compatible statistics ψ .

This definition applies to any random variable, whether its discrete, continuous, or something else and whether it has dimension 1 or higher.

Example 21.3. Let the random variable H represent a random point in the plane that is either the origin or a vertex of a regular hexagon with side length 1.



All the vertices of this hexagon lie on the unit circle, so

$$\mathcal{V} = \{\langle 0, 0 \rangle\} \cup \left\{ \left\langle \cos\left(k\frac{\pi}{3}\right), \sin\left(k\frac{\pi}{3}\right) \right\rangle \mid k \in [0..5] \right\}.$$

Suppose that the Distribution of H is symmetric with respect to all rotations by multiples of 60 degrees, clockwise or counter-clockwise. If f is such a rotation,

then there is an integer m such that

$$\begin{aligned} f(\langle 0, 0 \rangle) &= \langle 0, 0 \rangle \\ f(\langle \cos(k\frac{\pi}{3}), \sin(k\frac{\pi}{3}) \rangle) &= \langle \cos((k+m)\frac{\pi}{3}), \sin((k+m)\frac{\pi}{3}) \rangle \text{ for all } k \in [0..5]. \end{aligned}$$

Then, equation (21.9) tells us that for all such f , i.e., for all integers m ,

$$D_X(\psi \circ f) = D_X(\psi).$$

Let us expand each side. First, the left-hand side:

$$\begin{aligned} D_X(\psi \circ f) &= K_H(0, 0)\psi(f(0, 0)) + \sum_{k=0}^5 K_H(\cos(k\frac{\pi}{3}), \sin(k\frac{\pi}{3})) \psi(f(\cos(k\frac{\pi}{3}), \sin(k\frac{\pi}{3}))) \\ &= K_H(0, 0)\psi(0, 0) + \sum_{k=0}^5 K_H(\cos(k\frac{\pi}{3}), \sin(k\frac{\pi}{3})) \psi(\cos((k+m)\frac{\pi}{3}), \sin((k+m)\frac{\pi}{3})). \end{aligned}$$

Second, the right-hand side:

$$\begin{aligned} D_X(\psi) &= K_H(0, 0)\psi(0, 0) + \sum_{k=0}^5 K_H(\cos(k\frac{\pi}{3}), \sin(k\frac{\pi}{3})) \psi(\cos(k\frac{\pi}{3}), \sin(k\frac{\pi}{3})) \\ &= K_H(0, 0)\psi(0, 0) + \sum_{k=0}^5 K_H(\cos(k\frac{\pi}{3}), \sin(k\frac{\pi}{3})) \psi(\cos(k\frac{\pi}{3}), \sin(k\frac{\pi}{3})) \end{aligned}$$

Now, plugging the statistic $\{\langle x, y \rangle = \langle 1, 0 \rangle\}$ into both equations yields:

$$K_H(\cos(-m\frac{\pi}{3}), \sin(-m\frac{\pi}{3})) = K_H(1, 0)$$

for all integers m . This means that K_H gives the same weight to every hexagon vertex. That is, for some $0 \leq p \leq 1/6$ we have

$$K_H(x, y) = (1 - 6p) \{\langle x, y \rangle = \langle 0, 0 \rangle\} + \sum_{k=0}^5 p \{\langle x, y \rangle = \langle \cos(k\frac{\pi}{3}), \sin(k\frac{\pi}{3}) \rangle\}.$$

In the previous example, the symmetry transformations in $\mathcal{F}$ map each point of $\mathcal{V}$ to a distinct point of $\mathcal{V}$. This is not a coincidence. Though it is not stated explicitly, when $\mathcal{V}$ is finite, the definition implies that the functions in $\mathcal{F}$ must all be *bijections*.¹ To see why, suppose here is a $v' \in \mathcal{V}$ not in $\text{range}(f)$. Choose $\varphi(x) = \{x = v'\}$. Then, $D_X(\varphi \circ f) = 0$ because $f(X)$ cannot have value v' , but

¹See Section 14.1.

$D_X(\varphi) > 0$ because $\mathcal{V} = \text{range}(X)$ is the set of X 's possible values. So equation (21.9) fails, and f must therefore be a *surjection*. Now, suppose that there were $v_1 \neq v_2 \in \mathcal{V}$. Because $\mathcal{V}$ is finite and $f: \mathcal{V} \rightarrow \mathcal{V}$, this means that there must be a $v' \in \mathcal{V}$ that is not in $\text{range}(f)$. So equation (21.9) fails, and f must be an *injection*.

Consider two similar symmetries in the continuous case.

Example 21.4. Let P be a random point (2-dimensional) in the plane for which P and $\text{swap}(P)$ have the same Distribution where

$$\text{swap}(x, y) = \langle y, x \rangle,$$

which is a reflection across the $y = x$ line.

If P is discrete, for example, we can apply equation (19.42) to see

$$D_P(\psi) = \sum_{x,y} K_P(x, y) \psi(x, y) = \sum_{x,y} K_P(x, y) \psi(\text{swap}(x, y)) = D_P(\psi \circ \text{swap}),$$

where $\psi(\text{swap}(x, y)) = \psi(y, x)$. For any $\langle x_0, y_0 \rangle$, apply this to $\psi(x, y) = \{\langle x, y \rangle = \langle x_0, y_0 \rangle\}$, with $\psi(\text{swap}(x, y)) = \{y = x_0 \wedge x = y_0\}$, yielding

$$K_P(x_0, y_0) = K_P(y_0, x_0) \text{ for all } x_0, y_0.$$

The Distribution/Kernel of P assigns the same weight to points that are reflections of each other across the diagonal.

If P is continuous, the argument is analogous:

$$\begin{aligned} D_P(\psi) &= \iint_{x,y} dx dy K_P(x, y) \psi(x, y) \\ &= \iint_{x,y} dx dy K_P(x, y) \psi(\text{swap}(x, y)) \\ &= \iint_{x,y} dx dy K_P(x, y) \psi(y, x) = D_P(\psi \circ \text{swap}). \end{aligned}$$

Because integrals are unaffected by changes to the integrand at any finite set of points, we need to close out this argument a bit more carefully. Specifically, this equation tells us that for every compatible ψ

$$\iint_{x,y} dx dy (K_P(x, y) - K_P(y, x)) \psi(x, y) = 0.$$

Taking $\psi(x, y) = K_P(x, y) - K_P(y, x)$, an odd but perfectly valid statistic, gives us

$$\iint_{x,y} dx dy (K_P(x, y) - K_P(y, x))^2 = 0.$$

Because $(K_P(x, y) - K_P(y, x))^2 \geq 0$, this tells us that

$$K_P(x, y) = K_P(y, x)$$

for all x, y , except for at most an isolated set of points that do not change the integral, and thus do not affect any predictions with the Distribution. (The need for such provisos – even though they have no practical effect on our results – is one cost of allowing uncountable infinities.) The probability density of P is thus symmetric across the diagonal as well.

Example 21.5. Let Z be a continuous random variable with possible values in the unit disk $\mathcal{D} = \{\langle x, y \rangle \mid x^2 + y^2 \leq 1\}$. Suppose that Z 's Distribution is symmetric with respect to all rotations of the plane around the origin.

It will convenient to represent points in the plane in *polar coordinates*. (See Example 20.5.) A point $\langle x, y \rangle$ is represented by the pair $\langle r, \theta \rangle$ where r is the distance from the origin to the point and θ is the angle (in radians) measured counter-clockwise from the positive x -axis. Our summation operation translates as

$$\iint_{x,y} dx dy = \int_{r=0}^{\infty} r dr \int_{\theta=0}^{2\pi} d\theta$$

So, for any $f: \mathbb{R} \rightarrow \mathbb{R}$ that is a *rotation around the origin*, equation (21.9) becomes

$$\begin{aligned} D_Z(\psi) &= \int_{r=0}^{\infty} r dr \int_{\theta=0}^{2\pi} d\theta K_Z(r, \theta) \phi(r, \theta) \\ &= \int_{r=0}^{\infty} r dr \int_{\theta=0}^{2\pi} d\theta K_Z(r, \theta) \phi(f(r, \theta)) \\ &= D_Z(\psi \circ f). \end{aligned}$$

Any rotation f has an inverse rotation that rotates an equal and opposite amount.

So,

$$\int_{r=0}^{\infty} r \, dr \int_{\theta=0}^{2\pi} d\theta \, K_Z(r, \theta) \phi(f(r, \theta)) = \int_{r=0}^{\infty} r \, dr \int_{\theta=0}^{2\pi} d\theta \, K_Z(f^{-1}(r, \theta)) \phi(r, \theta).$$

By the same logic as in the previous example, we have that

$$K_Z(r, \theta) = K_Z(f^{-1}(r, \theta))$$

for all rotations f . So the Kernel is *constant on circles*.

In the following Sections, we will study symmetry in detail with respect to permutations of finite sets and a similar collection of transformations over contiguous regions.

21.1 Symmetry for Finite Sets

Let $\mathcal{V} = \{v_1, \dots, v_n\}$ be a finite set. A **permutation**² of $\mathcal{V}$ is a bijection $\rho: \mathcal{V} \rightarrow \mathcal{V}$. A permutation of n items can be pictured as a rearrangement of the labels $\langle 1, 2, \dots, n \rangle$: $\langle 1, 2, \dots, n \rangle \rightarrow \langle \rho(1), \rho(2), \dots, \rho(n) \rangle$. Think of a permutation of $\mathcal{V}$ as rearranging the labels on the v_i 's to $v_{\rho(i)}$'s. There are $n!$ permutations of an n -item set.

²See Interlude F Sections 14.1, 15.3, and 18.10 for much more on permutations.

A *Uniform Distribution* on a finite set is symmetric under all permutations of that set. This is equation (21.9) where $\mathcal{F}$ is the set of all permutations of $\mathcal{V}$.

Definition 56. Let X be random variable with a finite set of possible values $\mathcal{V}$. We say that X has a **discrete Uniform Distribution on $\mathcal{V}$** if for any permutation ρ of $\mathcal{V}$ and every statistic ψ compatible with X ,

$$D_X(\psi \circ \rho) = D_X(\psi),$$

or equivalently,

$$\mathbb{E}(\psi(\rho(X))) = \mathbb{E}(\psi(X)).$$

We call this a **DiscreteUniform** $\langle \mathcal{V} \rangle$ distribution.

If X has a DiscreteUniform $\langle \mathcal{V} \rangle$ Distribution, then our predictions of the output of any statistic are invariant under permutations of the values. So, we have

$$D_X(\psi) = \frac{1}{n!} \sum_{\rho} D_X(\psi \circ \rho)$$

$$\begin{aligned}
&= \frac{1}{n!} \sum_{\rho} \mathbb{E}(\psi(\rho(X))) \\
&= \mathbb{E} \left(\frac{1}{n!} \sum_{\rho} \psi(\rho(X)) \right) \\
&= \mathbb{E} \left(\frac{(n-1)!}{n!} \sum_{i=1}^n \psi(v_i) \right) \\
&= \mathbb{E} \left(\frac{1}{n} \sum_{i=1}^n \psi(v_i) \right), \\
&= \frac{1}{n} \sum_{i=1}^n \psi(v_i).
\end{aligned}$$

We apply the Additivity and Scaling Rules in the second equation. The fourth equality here is the tricky one. As we go over all permutations ρ , $\rho(X)$ goes over all possible values of X *no matter what value X has*. Moreover, it hits each value exactly $(n-1)!$ times because there are $(n-1)!$ permutations that fix a specific value. The sum in the expectation in the fifth equality is now constant, so the Constancy Rule gives us the result. This is a formula for computing any prediction of a uniformly distributed random variable; we simply average the statistic output over all possible inputs.

If X has a DiscreteUniform $\langle \mathcal{V} \rangle$ Distribution,

$$D_X(\psi) = \frac{1}{\#\mathcal{V}} \sum_{i=1}^{\#\mathcal{V}} \psi(v_i). \quad (21.10)$$

A uniform Distribution on a finite set thus simply *averages* the output of a statistic over all valid inputs. For example, if for $v \in \mathcal{V}$ we plug statistic $\langle x \rangle \mapsto \{x = v\}$ into equation (21.10), we get

$$K_X(v) = D_X(\langle x \rangle \mapsto \{x = v\}) = \frac{1}{n}$$

Thus all values have equal weight – and are thus “equally likely” – which is an expression of the symmetry.

The converse of this holds as well. If $K_X(v) = 1/n$ for all $v \in \mathcal{V}$, with $n = \#\mathcal{V}$, then X has a DiscreteUniform $\langle \mathcal{V} \rangle$ distribution. This is true because that Kernel leads directly to equation (21.10) by the mass-balancing equation.

Example 21.6. The roll of a balanced s -sided die D has a DiscreteUniform $\langle[1 \dots s]\rangle$ distribution:

$$D_D(\psi) = \frac{1}{s} \sum_{i=1}^s \psi(i).$$

$$\mathbb{E}(D) = D_D(\text{id}) = \frac{1}{s} \sum_{i=1}^s i = \frac{s+1}{2}$$

$$\mathbb{E}(D^2) = D_D(\blacksquare^2) = \frac{1}{s} \sum_{i=1}^s i^2 = \frac{(s+1)(2s+1)}{6}$$

$$\mathbb{E}(D^2 \{D \leq \frac{s}{2}\}) = D_D(\langle d \rangle \mapsto d^2[1 \dots s/2](d)) = \frac{1}{s} \sum_{i=1}^{s/2} i^2 = \frac{(s+1)(s+2)}{12}$$

$$\mathbb{E}(\{a \leq D \leq b\}) = D_D([a \dots b]) = \frac{1}{s} \sum_{i=\max(a,1)}^{\min(b,s)} 1 = \frac{\min(b,s) - \max(a,1) + 1}{s}$$

$$\mathbb{E}(\{D = i\}) = D_D(\{\blacksquare = i\}) = \frac{1}{s}.$$

Puzzle 166. If D is the random variable from the previous example, explain why $D \star \star n$ has a uniform distribution – and describe the set it is uniform over.

If X has a DiscreteUniform $\langle\mathcal{V}\rangle$ distribution and ξ is a compatible condition, then the Kind of $X \mid \xi$ simply eliminates all branches that are inconsistent with ξ . The *relative weights of the values does not change*. Because X is uniform, all weights are equal both before and after the observational constraint is applied. Hence, $X \mid \xi$ is *also uniform*, over the subset of values in $\mathcal{V}$ that are consistent with $\mathcal{V}$. it's distribution is DiscreteUniform $\langle\mathcal{V}_1\rangle$, where $\mathcal{V}_1 = \{v \in \mathcal{V} \mid \xi(v) = 1\}$, the subset of $\mathcal{V}$ that is consistent with the condition.

If X has a DiscreteUniform $\langle\mathcal{V}\rangle$ distribution and ξ is a compatible condition, then $X \mid \xi$ is *also uniform*; it's distribution is DiscreteUniform $\langle\mathcal{V}_1\rangle$, where

$$\mathcal{V}_1 = \{v \in \mathcal{V} \mid \xi(v) = 1\}, \quad (21.11)$$

the subset of $\mathcal{V}$ that is consistent with the condition.

Example 21.7. Let $M = S_1 \star S_2 \star \cdots \star S_n$ where the S_i 's all have common Kind

$$\langle \rangle \begin{cases} \frac{1}{2} \text{---} \langle 0 \rangle \\ \frac{1}{2} \text{---} \langle 1 \rangle \end{cases}$$

Then, M has a DiscreteUniform $\langle \mathcal{C} \rangle$ distribution where $\mathcal{C} = \{0, 1\}^n$. To see this, notice that

$$K_M(v) = K_{S_1}(v_1) K_{S_2}(v_2) \cdots K_{S_n}(v_n) = 2^{-n} \mathcal{C}(v),$$

which is constant on $\mathcal{C}$.

Now suppose we apply the observational constraint with condition $\xi(v) = \{\text{Sum}(v) = k\}$ for some $k \in [0..m]$. From the argument above, we should see that $M \mid \xi$ is also uniform on $\mathcal{C}_k = \{v \in \mathcal{C} \mid \text{Sum}(v) = k\}$.

And indeed this is true. There are $\binom{n}{k}$ elements in $\mathcal{C}_k$, each of which has k 1's and $n - k$ 0's. Applying the observation gives us

$$\begin{aligned} K_{M \mid \xi}(v) &\propto 2^{-n} \mathcal{C}(v) \{\text{Sum}(v) = k\} \\ &= \frac{2^{-n} \mathcal{C}_k(v)}{\sum_u 2^{-n} \mathcal{C}_k(u)} \\ &= \frac{\mathcal{C}_k(v)}{\#\mathcal{C}_k} \\ &= \frac{\mathcal{C}_k(v)}{\frac{1}{\binom{n}{k}}}. \end{aligned}$$

which is constant on $\mathcal{C}_k$. (Here, we are using sets as *indicator functions* as described in Chapter 12.)

Next, we consider a range of examples of symmetry and uniformity on finite sets.

Example 21.8. Let $B = \langle B_1, \dots, B_n \rangle$ for some $n \in [1..]$ be random bits, and assume that B has a DiscreteUniform $\langle \{0, 1\}^n \rangle$ Distribution. Let N be the number of bits in B that are 1. Find the Distribution of N .

First, notice that $N := \text{Sum}(B)$, with range $[0..n]$.

If our goal were only to compute $\mathbb{E}(N)$, we could use the Additivity Rule

$$\mathbb{E}(N) = \mathbb{E}(\text{Sum}(B)) = \text{Sum}(\mathbb{E}(B)) = \mathbb{E}(B_1) + \cdots + \mathbb{E}(B_n).$$

Then, if $b_{-i} = \langle b_1, \dots, b_{i-1}, b_{i+1}, \dots, b_n \rangle$, the vector of bits excluding b_i :

$$\mathbb{E}(B_i) = \mathbb{E}(\text{proj}_i(B)) = 2^{-n} \sum_{b_i=0}^1 b_i \sum_{b_{-i} \in \{0,1\}^{n-1}} 1 = 2^{-n}(0 + 2^{n-1}) = \frac{1}{2},$$

giving $\mathbb{E}(N) = n/2$. Here, b_{-i} refers to all the bits *except* the i th. The double sum has the value 2^{n-1} because only the $b_i = 1$ term is non-zero and this is the sum of 1 over all b_{-i} .

But we want to find the full Distribution of N , enabling us to compute any prediction about N . For any compatible statistic φ for N , we have

$$\begin{aligned} D_N(\varphi) &= \mathbb{E}(\varphi(N)) \\ &= \mathbb{E}(\varphi(\text{Sum}(B))) \\ &= D_B(\varphi \circ \text{Sum}) \\ &= 2^{-n} \sum_{b \in \{0,1\}^n} \varphi(b_1 + \dots + b_n) \\ &= \sum_{k=0}^n \varphi(k) \binom{n}{k} 2^{-n}, \end{aligned}$$

because there are exactly $\binom{n}{k}$ vectors b with k bits set. In Interlude C, we call this the Binomial $\langle n, \frac{1}{2} \rangle$ distribution.

With this Distribution, we can read off various expectations easily:

$$\begin{aligned} \mathbb{E}(N) &= D_N(\langle x \rangle \mapsto x) = \sum_{k=0}^n \binom{n}{k} 2^{-n} k \\ &= \frac{n}{2} \sum_{k=1}^n \binom{n-1}{k-1} 2^{-(n-1)} \\ &= \frac{n}{2} \sum_{k=j}^{n-1} \binom{n-1}{j} 2^{-(n-1)} = \frac{n}{2}, \\ \mathbb{E}(\{N = k\}) &= D_N(\langle x \rangle \mapsto \{x = k\}) = \binom{n}{k} 2^{-n} \\ \mathbb{E}((N - n/2)^2) &= D_N(\langle x \rangle \mapsto (x - n/2)^2) \\ &= \sum_{k=0}^n \binom{n}{k} 2^{-n} (k - n/2)^2 = \frac{n}{4}. \end{aligned}$$

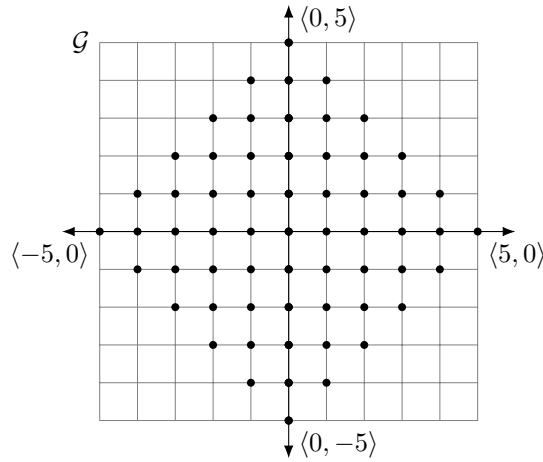


FIGURE 21.1. The range of Z in the Example 21.9. Note that $\#\mathcal{G} = 61$.

The first calculation uses that $\binom{n}{k}$ is the number of k subsets from size n , which sums over k to 2^n . The last calculation uses a bit of algebra to simplify, in particular

$$\mathbb{E}((N - n/2)^2) = \mathbb{E}(N^2 - nN + n^2/4) = \mathbb{E}(N(N - 1) - (n - 1)N + n^2/4).$$

Now apply additivity and the Distribution, noting that $k(k - 1)\binom{n}{k} = n(n - 1)\binom{n-2}{k-2}$. (See the [Help Sheet](#).)

Example 21.9. Let $Z = \langle X, Y \rangle$ be a random point chosen uniformly on the grid $\mathcal{G}$ below; that is Z has a $\text{DiscreteUniform}(\mathcal{G})$ Distribution. We thus know that

$$D_Z(\psi) = \sum_{\langle j, k \rangle \in \mathcal{G}} \frac{1}{61} \psi(j, k) = \frac{1}{61} \sum_{j=-5}^5 \sum_{k=|j|-5}^{5-|j|} \psi(j, k).$$

Note that in the double sum, we are slicing the grid into vertical slices. We could do the sum in the other order and slice it into horizontal slices instead

$$D_Z(\psi) = \frac{1}{61} \sum_{k=-5}^5 \sum_{j=|k|-5}^{5-|k|} \psi(j, k)$$

and get the same result.

We want to find:

1. $\mathbb{E}(Z)$,
2. $\mathbb{E}(\max(X, Y))$,
3. The Distribution of X ,
4. $\mathbb{E}(\{X = i\})$ for $i \in [-5 \dots 5]$ and the Distribution of X given $Y = j$ for $j \in [-5 \dots 5]$.

We will tackle each of these in turn in the following examples.

Example 21.10. Let Z be as in Example 21.9.

$$\begin{aligned}
 \mathbb{E}(Z) &= D_Z(\text{id}) \\
 &= \frac{1}{61} \sum_{j=-5}^5 \sum_{k=|j|-5}^{5-|j|} \langle j, k \rangle \\
 &= \frac{1}{61} \sum_{j=-5}^5 \langle j, 0 \rangle \\
 &= \langle 0, 0 \rangle.
 \end{aligned}$$

Example 21.11. Let Z be as in Example 21.9. For $\mathbb{E}(\max(X, Y))$, we can just compute the sum:

$$\begin{aligned}
 \mathbb{E}(\max(X, Y)) &= D_Z(\langle z \rangle \mapsto \max(z_1, z_2)) \\
 &= \frac{1}{61} \sum_{j=-5}^5 \sum_{k=|j|-5}^{5-|j|} \max(j, k) = \frac{84}{61}.
 \end{aligned}$$

Example 21.12. Let Z be as in Example 21.9. Because $X := \text{proj}_1(Z)$, the Distribution of X derives from D_Z by the now familiar pattern of composition.

Let φ be a compatible statistic for X :

$$\begin{aligned}
 D_X(\varphi) &= \mathbb{E}(\varphi(X)) = \mathbb{E}((\varphi \circ \text{proj}_1)(Z)) = D_Z(\varphi \circ \text{proj}_1) \\
 &= \frac{1}{61} \sum_{j=-5}^5 \sum_{k=|j|-5}^{5-|j|} \varphi(j) \\
 &= \sum_{j=-5}^5 \varphi(j) \frac{1}{61} \sum_{k=|j|-5}^{5-|j|} 1 \\
 &= \sum_{j=-5}^5 \varphi(j) \frac{11-2|j|}{61}.
 \end{aligned}$$

It's true, as always, that $D_X(1) = 1$, so again the Distribution is a weighted average of the possible outputs of the statistic, though here the weights are decidedly not uniform:

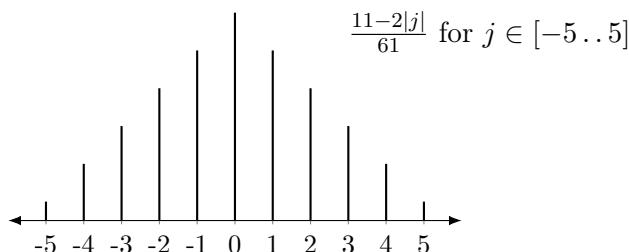


FIGURE 21.2. The averaging weights in D_X . Note that these are probabilities.

Notice that if we had done the calculation $\mathbb{E}(\{Y = i\})$, we would get the *same* answer because the grid $\mathcal{G}$ is symmetric under swapping coordinates and Z is uniform. (A coordinate swap is just a permutation of on $\mathcal{G}$.) So, X and Y have *the same Distribution*. (They are *not* the same as random variables, however.)

$\mathbb{E}(\{X = i\})$ is the probability that X has the value i ; we can see from the picture that there are more ways for X to be 0 than 5, so we do not expect a uniform Distribution here. And indeed, these probabilities are *just the weights* in D_X above. That is: $\mathbb{E}(\{X = i\}) = D_X(\langle x \rangle \mapsto \{i\}(x)) = (11 - 2|i|)/61$. These are just the averaging weights

What happens if we observe the value of Y ? Do our predictions about X

change? Yes! Let $j \in [-5 \dots 5]$.

$$\begin{aligned} D_{X|Y}(\varphi | j) &= \mathbb{E}(\varphi(X) | Y = j) = \frac{\mathbb{E}(\varphi(X)\{Y = j\})}{\mathbb{E}(\{Y = j\})} \\ &= \frac{\mathbb{E}(\varphi(X)\{Y = j\})}{\frac{11-2|j|}{61}}, \end{aligned}$$

because X and Y have the same Distribution. To compute the numerator, notice that it is just $D_Z(\langle x, y \rangle \mapsto \varphi(x)\{y = j\})$. Hence,

$$\begin{aligned} \mathbb{E}(\varphi(X)\{Y = j\}) &= \frac{1}{61} \sum_{b=-5}^5 \sum_{a=|b|-5}^{5-|b|} \varphi(a)\{b = j\} \\ &= \frac{1}{61} \sum_{a=|j|-5}^{5-|j|} \varphi(a). \end{aligned}$$

Notice that we sliced vertically here since we knew the vertical component would be bound. Putting these together, we have

$$D_{X|Y}(\varphi | j) = \frac{1}{11 - 2|j|} \sum_{a=|j|-5}^{5-|j|} \varphi(a).$$

We conclude that *given* $Y = j$, X has a DiscreteUniform($[|j| - 5 \dots 5 - |j|]$) Distribution.

21.2 Symmetry for Continuous Regions

An n -dimensional *region* is a full and contiguous set of points with a smooth boundary. The insides of a sphere or a cube or a triangle are all regions, as is any open interval of real numbers. Higher dimensional sets are harder to visualize but the same idea holds. The boundary of an n -dimensional region is finite collection of $n - 1$ -dimensional regions glued together, like the line segments that bound a triangle.

A **region** $\mathcal{R}$ is a contiguous, open set in $\mathbb{R}^n$ whose boundary consists of finitely many smooth $n - 1$ dimensional pieces.

For regions there is a measure of size. One-dimensional regions (intervals) are measured by length; two dimensional regions by area; and three dimensional regions by volume. Here we will use the term **measure** to describe this general quantity in

any dimension. So length is 1-dimensional measure, area is 2-dimensional measure, volume is 3-dimensional measure, and so on.

The **measure** of a region in $\mathbb{R}^n$ is the n -dimensional analogue of volume: length for $n = 1$, area for $n = 2$, volume for $n = 3$, and so forth. We write $\text{measure}(\mathcal{R})$ for the measure of a region $\mathcal{R}$.

A **measure-preserving map** is a one-to-one correspondence $\mathcal{R} \rightarrow \mathcal{R}$ such that the measure of $f^{-1}(\mathcal{A})$ is the same as the measure of $\mathcal{A}$ for any region $\mathcal{A} \subseteq \mathcal{R}$.³ A measure-preserving map does not have to be smooth; indeed, we are interested in those that are not. For instance, in the square region below (Figure 21.3), consider the map that cuts out and swaps the two equal area sub-squares

³ f^{-1} is the *image* of f . See the definition on 581 in Section 10.1.

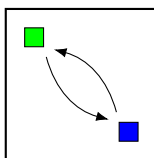


FIGURE 21.3. A region and a measure preserving map on that region that swaps the two shaded squares, leaving the rest of the points where they are.

A *Uniform Distribution on a region* is a Distribution that is symmetric with respect to collection of *measure-preserving maps* on that region. We call this a *continuous* uniform Distribution because a region contains an uncountable continuum of points.

Let X be continuous random variable all of whose possible values are in the region $\mathcal{R}$.

We say that X has a **(continuous) Uniform Distribution on $\mathcal{R}$** if for any measure-preserving map f on $\mathcal{R}$ and any statistic compatible with X ,

$$D_X(\psi \circ f) = D_X(\psi).$$

We call this a **Uniform($\mathcal{R}$)** distribution. In the common special case where $\mathcal{R}$ is a real interval $[a, b]$, we write this as **Uniform(a, b)**. (Note that it does not matter whether the endpoints are included or excluded from the interval.)

If X has a Uniform($\mathcal{R}$) distribution, then our predictions about the output of any statistic are invariant under transformations that do not stretch or shrink any part of the region. In particular, we have that

If X has a Uniform($\mathcal{R}$) Distribution, then for any statistic ψ compatible with X ,

$$D_X(\psi) = \mathbb{E}(\psi(X)) = \frac{1}{\text{measure}(\mathcal{R})} \int_{\mathcal{R}} dx \, \psi(x). \quad (21.12)$$

A Uniform Distribution on a region thus averages the output of a statistic over all valid inputs. This is directly analogous to the discrete case. The difference here is that we have a continuum of valid inputs, so the average is in the form of an *integral* rather than a finite sum.* The idea is the same, however, though as we will see the weights in this average have a different interpretation. Notice also that X can be a random variable of any dimension here. If it is, then the x in the integral is a vector too, so we could calculate it with a multiple integral in terms of the components, and dx represents an infinitesimal unit of measure.

A common special case of equation (21.12) is when the statistic ψ is a *condition* that represents the indicator of a sub-region $\mathcal{A} \subseteq \mathcal{R}$. Then, the integral of the statistic in equation (21.12) is just the measure of $\mathcal{A}$ and

$$\mathbb{E}(\mathcal{A}(X)) = D_X(\mathcal{A}) = \frac{\text{measure}(\mathcal{A})}{\text{measure}(\mathcal{R})}, \quad (21.13)$$

the relative measure of $\mathcal{A}$ within the region $\mathcal{R}$.

Keep in mind that *including or excluding points on the boundary of the region does not affect the integral*, so we can be flexible in defining uniform distribution over sets that are regions plus none, some, or all of their boundary points, as convenient. So for example, we can define a Uniform($\langle(0\_1)\rangle$) or Uniform($\langle[0\_1]\rangle$) or Uniform($\langle(0\_1]\rangle$) or Uniform($\langle[0\_1)\rangle$) and get equivalent distributions. And similarly for $\mathcal{D} = \{\langle x, y \rangle \mid x^2 + y^2 < 1\}$ and $\overline{\mathcal{D}} = \{\langle x, y \rangle \mid x^2 + y^2 \leq 1\}$; we treat Uniform($\langle\mathcal{D}\rangle$) and Uniform($\langle\overline{\mathcal{D}}\rangle$) as equivalent.

Example 21.13. Let $S = \langle X, Y \rangle$ be a two-dimension Uniform($\langle\mathcal{S}\rangle$) where $\mathcal{S}$ is the unit square $\mathcal{S} = \{\langle x, y \rangle \mid x, y \in [0\_1]\}$. We can answer the following questions:

1. What is the probability that $\mathcal{S}$ lies within the circle of radius $\frac{1}{4}$ centered at $\langle 1/4, 1/4 \rangle$?
2. What is the probability that $|X - Y| < \frac{1}{4}$?
3. What is the probability that $X < 1/3$ and $Y > 7/8$?

Each of these questions can be answered with a condition:

1. $\psi_1(x, y) = \{(x - 1/4)^2 + (y - 1/4)^2 \leq 1/16\}$.
2. $\psi_2(x, y) = \{|x - y| < 1/4\}$.

*Convince yourself that for a function $f(x)$ on the interval $[a\_b]$, $\frac{1}{b-a} \int_a^b dx f(x)$ makes sense as the average value of the function over that interval.

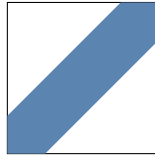
3. $\psi_3(x, y) = \{x < 1/3 \wedge y > 7/8\}$.

The expectations $\mathbb{E}(\psi_1(S))$, $\mathbb{E}(\psi_2(S))$, and $\mathbb{E}(\psi_3(S))$ give the probabilities of each of these conditions being true. We can use equation (21.13) to find the corresponding expectations by finding the area of the region within $\mathcal{S}$ for which each condition is true. This typically reduces to some geometry. Note that in this case $\text{measure}(\mathcal{S}) = \text{area}(\mathcal{S}) = 1$.

For question 1, ψ_1 equals 1 in a circular subregion of radius $1/4$, which has area $\pi/16$. Equation (21.13) tells us that

$$\mathbb{E}(\psi_1(S)) = \frac{\pi/16}{1} = \frac{\pi}{16}.$$

For question 2, ψ_2 equals 1 on the shaded region in the following picture:



The shaded area is 1 minus the area of the two unshaded triangles, which have side length $\frac{3}{4}$. So, we get $\mathbb{E}(\psi_2(S)) = 1 - (\frac{3}{4})^2 = \frac{7}{16}$.

For question 3, ψ_3 equals 1 in the square region with width $1/3$ and height $1/8$, so $\mathbb{E}(\psi_3(S)) = \frac{1}{3} \cdot \frac{1}{8} = \frac{1}{24}$.

Example 21.14. Suppose S has a Uniform $([-1, 1])$ Distribution:

$$D_S(\psi) = \frac{1}{2} \int_{t=-1}^1 dt \psi(t).$$

$$\mathbb{E}(S) = D_S(\text{id}) = \frac{1}{2} \int_{t=-1}^1 dt t = 0$$

$$\mathbb{E}(S^2) = D_S(\blacksquare^2) = \frac{1}{2} \int_{t=-1}^1 dt t^2 = \frac{1}{3}$$

$$\mathbb{E}(S^2\{S \leq 0\}) = D_S(\langle s \rangle \mapsto s^2 \{-1 \leq s \leq 0\}) = \frac{1}{2} \int_{t=-1}^0 dt t^2 = \frac{1}{6}$$

$$\mathbb{E}(\{a \leq S \leq b\}) = D_S(\langle s \rangle \mapsto \{a \leq s \leq b\}) = \frac{1}{2} \int_{t=\max(a, -1)}^{\min(b, 1)} dt = \frac{\min(b, 1) - \max(a, -1)}{2}$$

These are all directly analogous to the discrete example we saw earlier, with integrals replacing sums.

The last result does give an odd outcome when $a = b$:

$$\mathbb{E}(\{S = u\}) = D_S(\{\blacksquare = u\}) = 0.$$

So though S must have some value, every value has zero probability! How do we interpret this? We will look at this in detail later, but for now think of a gas in the room where you are sitting. If you look at any point in space, you will probably not find a gas molecule. They are wizzing through space, so that if you look at a small region around that point, you will be able to tell how dense the gas is. But a point is too focused to be able to detect it.

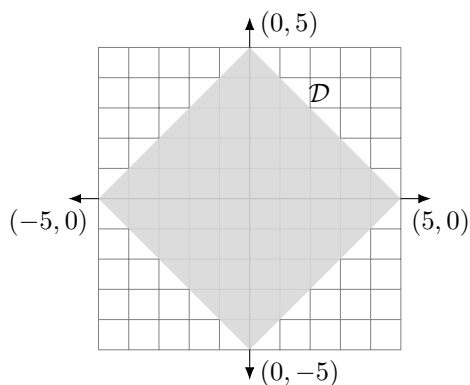


FIGURE 21.4. The range of Z in Example 21.15. Note that $\mathcal{D}$ has area 50.

Example 21.15. Let $Z = \langle X, Y \rangle$ be a random point chosen *uniformly* on the region $\mathcal{D}$ in Figure 21.4. Because this has a Uniform $\langle \mathcal{D} \rangle$ Distribution, we know

$$\begin{aligned} D_Z(\psi) &= \int_{\mathcal{D}} dz \, K_Z(z) \psi(z) \\ &= \frac{1}{50} \int_{\mathcal{D}} dz \, \psi(z) \\ &= \int_{u=-5}^5 \int_{v=|u|-5}^{5-|u|} dv \, du \, \frac{1}{50} \psi(u, v) \\ &= \int_{v=-5}^5 \int_{u=|v|-5}^{5-|v|} du \, dv \, \frac{1}{50} \psi(u, v). \end{aligned}$$

As with the sums, the second equation slices the diamond vertically and the third equation slices horizontally. But both sum over the same diamond and thus give the same result.

We are interested in finding

1. $\mathbb{E}(Z)$,
2. $\mathbb{E}(\{|X| + |Y| \leq 2\})$,
3. The Distribution of X and the Distribution of X given $Y > 0$.

We will compute these in the following examples.

Example 21.16. Let Z be as defined in Example 21.15.

$$\begin{aligned}
 \mathbb{E}(Z) &= D_Z(\text{id}) \\
 &= \int_z dz \, K_Z(z) \, \text{id}(z) \\
 &= \frac{1}{50} \int_{u=-5}^5 \int_{v=|u|-5}^{5-|u|} dv \, du \, \langle u, v \rangle \\
 &= \frac{1}{50} \int_{u=-5}^5 du \, \langle u, 0 \rangle \\
 &= (0, 0).
 \end{aligned}$$

Example 21.17. Let Z be as defined in Example 21.15. The set of points $\{\langle x, y \rangle \mid |x| + |y| \leq 2\}$ is a small diamond scaled to $2/5$ the size of $\mathcal{D}$. The area of this small diamond is 8.

$$\begin{aligned}
 \mathbb{E}(\{|X| + |Y| \leq 2\}) &= D_Z(\mathcal{T}) \\
 &= \int_z dz \, K_Z(z) \, \mathcal{T}(z) \\
 &= \frac{1}{50} \int_{u=-2}^2 \int_{v=|u|-2}^{2-|u|} dv \, du \\
 &= \frac{1}{50} \int_{u=-2}^2 du \, (4 - 2|u|) \\
 &= \frac{2}{50} \int_{u=0}^2 du \, (4 - 2u) \\
 &= \frac{2}{50} (8 - 4) = \frac{8}{50},
 \end{aligned}$$

which is the relative area of the small diamond within $\mathcal{D}$.

Example 21.18. Let Z be as defined in Example 21.15. For any compatible statistic φ for X ,

$$D_X(\varphi) = \mathbb{E}(\varphi(X)) = E(\varphi(\text{proj}_1(Z))) = D_Z(\varphi \circ \text{proj}_1).$$

So,

$$\begin{aligned} D_X(\varphi) &= D_Z(\varphi \circ \text{proj}_1) \\ &= \int_z dz \, K_Z(z) \varphi(\text{proj}_1(z)) \\ &= \frac{1}{50} \int_{u=-5}^5 \int_{v=|u|-5}^{5-|u|} dv \, du \, \varphi(u) \\ &= \frac{1}{50} \int_{u=-5}^5 du \, (10 - 2|u|) \varphi(u). \end{aligned}$$

Notice that if we had done this calculation with Y instead of X we would get the same result. (Indeed, swapping coordinates is a measure-preserving map.) So X and Y have the same Distribution.

Given $Y > 0$, we can define $p(y) = \{0 < y < 5\}$ and $g(x, y) = \varphi(x)p(y)$. Then, by applying an observation

$$\begin{aligned} D_{X|Y>0}(\varphi) &= \mathbb{E}(\varphi(X) \mid Y > 0) \\ &= \frac{\mathbb{E}(\varphi(X)\{Y > 0\})}{\mathbb{E}(\{Y > 0\})} \\ &= \frac{D_Z(g)}{D_Y(p)}. \end{aligned}$$

The numerator is

$$\int_{u=-5}^5 \int_{v=0}^{5-|u|} dv \, du \, \frac{1}{50} \varphi(u) = \int_{u=-5}^5 du \, \frac{5-|u|}{50} \varphi(u).$$

The denominator is

$$\frac{1}{50} \int_{v=0}^5 dv \, (10 - 2|v|) = \frac{1}{2}.$$

So,

$$D_{X|Y>0}(\varphi) = \int_{u=-5}^5 du \frac{5-|u|}{25} \varphi(u).$$

Like before, more weight is given to values near zero than near ± 5 , but the predictions about X have changed given this information.

Predicting with Expectations

22

Chapter

Contents

22.1 The Expectation Rules	922
22.2 The Mass-Balancing Equation	928

Key Take Aways

In this Chapter, we compute expectations of arbitrary random variables, through both the expectation rules and direct calculation. To highlight that these techniques apply to any random variable, we show the expectations here as $\mathbb{E}(\blacksquare \mid \mathcal{C})$ with an arbitrary observational constraint applied, represented by a Boolean expression $\mathcal{C}$. If $\mathcal{C} = \top$, we can simply drop the $\mid$ bars.

Let X and Y be random variables of equal dimension, a and b scalar constants, and $u \in \text{range}(Y)$ a possible value of Y .

The expectation operators satisfy useful laws we call the **Expectation Rules**:

1. Constancy Rule. $\mathbb{E}(1 \mid \mathcal{C}) = 1$.
2. Scaling Rule. $\mathbb{E}(aX \mid \mathcal{C}) = a \mathbb{E}(X \mid \mathcal{C})$.
3. Additivity Rule. $\mathbb{E}(\text{Sum}(X) \mid \mathcal{C}) = \text{Sum}(\mathbb{E}(X \mid \mathcal{C}))$.
4. Ordering Rule. If **fact** $(X \leq Y)$, then $\mathbb{E}(X \mid \mathcal{C}) \leq \mathbb{E}(Y \mid \mathcal{C})$.
5. Substitution Rule. If Y equals the constant u whenever $\mathcal{C}$ is true, $\mathbb{E}(g(X, Y) \mid \mathcal{C}) = \mathbb{E}(g(X, u) \mid \mathcal{C})$.
6. Monotone Limits Rule. If **fact** $(X_1 \leq X_2 \leq \dots)$ and **fact** $(X_n \rightarrow X \text{ as } n \rightarrow \infty)$, then $\lim_{i \rightarrow \infty} \mathbb{E}(X_i \mid \mathcal{C}) = \mathbb{E}(X \mid \mathcal{C})$.

The **mass-balancing equation** tells us how to compute the expectation of any transformed random variable as an average of the statistic's values weighted by the masses from the Kernel of the original random variable. When X has a *simple* Kernel, the mass-balancing equation becomes

$$\begin{aligned}\mathbb{E}(\varphi(X) \mid \mathcal{C}) &= D_{X|\mathcal{C}}(\psi \circ \varphi) \\ &= \sum_u K_{X|\mathcal{C}}(u) \{u \in \text{range}_d(X)\} \varphi(u) + \int_v dv K_{X|\mathcal{C}}(v) \{v \in \text{range}_c(X)\} \varphi(v)\end{aligned}\quad (22.1)$$

$$= \sum_u p_{X|\mathcal{C}}(u) \varphi(u) + \int_v dv f_{X|\mathcal{C}}(v) \varphi(v). \quad (22.2)$$

When X has a *split* Kernel, the mass-balancing equation becomes

$$\begin{aligned}\mathbb{E}(\varphi(X) \mid \mathcal{C}) &= D_{X|\mathcal{C}}(\psi \circ \varphi) \\ &= \sum_u \int_v dv K_{X|\mathcal{C}}(u, v) \varphi(u, v),\end{aligned}\quad (22.3)$$

where we list the discrete components (u) first and the continuous components (v) second.

We compute expectations directly by taking weighted averages. With FRPs, it was the average of the possible values (at the leaves of the Kind tree) weighted by the weights on the branches. And indeed for an FRP X whose Kind has values $v_1, v_2, \dots, v_n$ with weights $w_1, w_2, \dots, w_n$, we found $\mathbb{E}(\varphi(X) \mid \xi)$ for any compatible statistic φ and a compatible condition ξ that can be satisfied:

$$\mathbb{E}(\varphi(X) \mid \xi) = \frac{w_1 \varphi(v_1) \xi(v_1) + w_2 \varphi(v_2) \xi(v_2) + \dots + w_n \varphi(v_n) \xi(v_n)}{w_1 \xi(v_1) + w_2 \xi(v_2) + \dots + w_n \xi(v_n)} \quad (22.4)$$

This is **the average of the transformed values, weighted by the weights over those branches that are consistent with our knowledge**. When ξ is the condition $\top$ that is always true, this reduces to $\mathbb{E}(\varphi(X)) = p_1 \varphi(v_1) + \dots + p_n \varphi(v_n)$, where the p_i 's are the canonical weights, and when in addition $\varphi = \text{id}$, it reduces to $\mathbb{E}(X) = p_1 v_1 + \dots + p_n v_n$.

This idea carries over to general random variables: expectation is a weighted average of the values weighted by the “mass” at each value. In the general case, we usually get the weights from the Kernel or Distribution. The only variation from the

finite case arises in how we interpret the weights and how we “sum up” the terms. We might sum over a infinite number of possible values, and near continuous points, the weights in the average are infinitesimal masses derived as the product of a *density* from the Kernel and an infinitesimal measure. The **mass-balancing equation** expresses this idea and gives us a general tool for calculating our predictions.

Often, however, it is possible to forgo a direct calculation. We saw in Chapter 6 that by the logic of risk-neutral prices, expectations must satisfy several laws, which we call the **Expectation Rules**. The core Rules are Constancy, Ordering, Scaling, Additivity hold as is for general random variables. A new core Rule, the Monotone Limits Rule, applies only to infinite systems, and we will see several other Rules along the way.

Throughout this Chapter, we derive the expectations of arbitrary random variables, but we will often show the expectations with an arbitrary compatible statistic φ transform and subject to constraint with an arbitrary observation. The idea is to emphasize in practice that how we handle expectations does not change after applying any combination of the Big 3 operations. In particular, all the rules and formulae apply given an arbitrary observation. Here, we will express the observation through a Boolean expression $\mathcal{C}$. Recall that in general, we can express an observation as a Boolean expression in terms of one or more random variables. For example, if $\xi(x) = \{x > 2\}$, this condition is satisfied for X if $X > 2$, and we can use ξ or $X > 2$ on the right-hand side of a $|$ bar as an observation. For any such Boolean expression $\mathcal{C}$, $\{\mathcal{C}\}$ is the *event*¹ that that condition is true, and $X | \mathcal{C}$ is the random variable representing X with the constraint applied. When needed we can also use $\mathcal{C}$ as the *condition* itself.

¹A 0-1-valued random variable.

Assuming that the constraint $\mathcal{C}$ holds, X and $X | \mathcal{C}$ have the same value, but they may have different Distributions because the knowledge that $\{\mathcal{C}\}$ occurs can change our predictions about X ’s value. The Boolean expression $\top$ corresponds to the condition $\xi = \text{const}_1$, which is always true, and then X is the same as $X | \top$. **When $\mathcal{C} = \top$, we can thus drop all the $| \mathcal{C}$ ’s.**

For a Boolean expression $\mathcal{B}$, we write $\text{fact}(\mathcal{B})$ to indicate that the event $\{\mathcal{B}\}$ is known *a priori* to occur, i.e., $\mathbb{E}(\{\mathcal{B}\}) = 1$.²

Writing the expectations below in terms of $X | \mathcal{C}$ rather than just X is not really necessary, as these are just random variables like any other, but it serves as a reminder that these results apply whatever our knowledge state.

²See the next Chapter for a thorough discussion of these points.

22.1 The Expectation Rules

From the logic of risk-neutral prices, we derived a set of properties that the expectations of FRPs much satisfy: Constancy, Ordering, Scaling, Additivity, and Substitution. All of these properties carry over unchanged to general random variables; we call them the **Expectation Rules**. The infinities of the general case also give rise to one new rule, the Monotone Limits Rule, that we will see below. These rules apply to the expectation of any random variable of any dimension and including those which have been constrained by an observation. In short, the Expectation Rules show that $\mathbb{E}$ is a monotone, linear operator that preserves constants and that sometimes allows limits to pass in or out.

The core Expectation Rules are as follows.

The Constancy Rule.

$$\mathbb{E}(1 \mid \mathcal{C}) = 1. \quad (22.5)$$

Our best prediction of a constant random variable is the constant itself.

The Ordering Rule. If $\text{fact}(X \leq Y)$,

$$\mathbb{E}(X \mid \mathcal{C}) \leq \mathbb{E}(Y \mid \mathcal{C}). \quad (22.6)$$

The expectations of ordered random variables remain ordered. We thus say that expectations are *monotone*.

The Scaling Rule. For a scalar constant a ,

$$\mathbb{E}(aX \mid \mathcal{C}) = a \mathbb{E}(X \mid \mathcal{C}). \quad (22.7)$$

Constants can pass freely in and out of the expectation operator. This applies to any quantity that is constant when $\mathcal{C}$ is known to be true.

The Additivity Rule. For random variables $X_1, X_2, \dots, X_n$ of equal dimension, we have that $\mathbb{E}(\text{Sum}(X) \mid \mathcal{C}) = \text{Sum}(\mathbb{E}(X \mid \mathcal{C}))$, or equivalently

$$\mathbb{E}(X_1 + X_2 + \dots + X_n \mid \mathcal{C}) = \mathbb{E}(X_1 \mid \mathcal{C}) + \mathbb{E}(X_2 \mid \mathcal{C}) + \dots + \mathbb{E}(X_n \mid \mathcal{C}). \quad (22.8)$$

The expectation of a sum is the sum of the expectations.

The Substitution Rule. If $\text{fact}(Y\{\mathcal{C}\} = u\{\mathcal{C}\})$, i.e., Y equals the constant u whenever $\mathcal{C}$ is true, then

$$\mathbb{E}(g(X, Y) \mid \mathcal{C}) = \mathbb{E}(g(X, u) \mid \mathcal{C}). \quad (22.9)$$

Known random variables can be replaced by their values inside expectations.

The Monotone Limits Rule. If $\text{fact}(X_1 \leq X_2 \leq \dots)$ and $\text{fact}(X_n \rightarrow X \text{ as } n \rightarrow \infty)$, then

$$\lim_{i \rightarrow \infty} \mathbb{E}(X_i \mid \mathcal{C}) = \mathbb{E}(X \mid \mathcal{C}). \quad (22.10)$$

This rather technical rule allows us exchange the order of limit and expectation operators in many situations.

Example 22.1 Transforming to a Constant

Suppose A is a random variable whose only possible values are -1 and 1 . Then, the random variable A^2 (which is the inline expression of $\psi(A)$ for $\psi(x) = x^2$) only has 1 as a possible value. Therefore, A^2 is constant, and $\mathbb{E}(A^2) = 1$.

Let X be a random variable whose possible values are -1 , 2 , and 5 . Define $Y = X^3 - 6X + 3X + 20$.

Then, $\mathbb{E}(Y) = 10$. Why? Notice that $\varphi(x) = x^2 - 6x^2 + 3x + 20 = 10 + (x - 2)(x - 5)(x + 1)$. For each possible value x of X , $\varphi(x) = 10$, so Y is in fact the constant 10 .

As we see in these examples, it is often easier and clearer to describe the transforms of random variables by simple statistics in their inline form, e.g., writing $X^3 - 6X + 3X + 20$ for $\varphi(X)$ or A^2 for $\psi(A)$.

Example 22.2 Complementary Events

Consider the events $\{X > 5\}$ and $\{X \leq 5\}$ for some scalar random variable X . Because X 's value must *either* be bigger than 5 or at most 5 , we have

$$\{X > 5\} + \{X \leq 5\} = 1.$$

Exactly one of the two events must occur. We call these **complementary events**. Taking expectations and applying the Constancy and Additivity Rules,

we have that

$$\begin{aligned} 1 &= \mathbb{E}(\{X > 5\} + \{X \leq 5\}) \\ &= \mathbb{E}(\{X > 5\}) + \mathbb{E}(\{X \leq 5\}), \end{aligned}$$

so,

$$\mathbb{E}(\{X > 5\}) = 1 - \mathbb{E}(\{X \leq 5\}).$$

The probabilities of two complementary events must add to 1.

In general, if V be an event, its **complementary event** is $W = 1 - V$. W occurs if and only V does not occur, and vice versa. So,

$$\mathbb{E}(V) + \mathbb{E}(W) = 1 \quad (22.11)$$

and

$$\mathbb{E}(W) = 1 - \mathbb{E}(V). \quad (22.12)$$

The probabilities of complementary events sum to 1, so we can always find one probability from the other. Sometimes a complementary event is easier to reason about, so a useful strategy to keep in mind is to remember that we can always work with the complement to find the probability of the original event. (See the next example.)

Example 22.3. Let $N = \text{Sum}(H_1 \star H_2 \star \cdots \star H_n)$, where random variables H_i all have Kind

$$\langle \rangle \begin{cases} \text{---} 1-p \text{---} \langle 0 \rangle \\ \text{---} p \text{---} \langle 1 \rangle \end{cases}$$

for some $0 \leq p \leq 1$. Find $\mathbb{E}(\{N > 0\})$.

There are many ways that N can be positive, i.e., many branches of the independent join tree that we need to combine for that event. Instead, we can use the observation in the previous example and consider the *complementary* event $\{N = 0\}$. This event occurs only if *all* the H_i 's equal 0, so multiplying (canonical) weights on every path from root to leaf in the unfolded Kind for $H_1 \star \cdots \star H_n$ gives

$$\mathbb{E}(\{N = 0\}) = K_{H_1}(0) \cdots K_{H_n}(0) = (K_{H_1}(0))^n = (1-p)^n.$$

So, $\mathbb{E}(\{N = 0\}) = (\mathbb{E}(\{H_1 = 0\}))^n$, and thus

$$\mathbb{E}(\{N > 0\}) = 1 - (1 - p)^n$$

because the events are complementary.

Example 22.4 Red Sphere

The entire surface of a sphere is colored red and blue, with a proportion β colored blue. We want to show that if $\beta = 0.1$, no matter how the two colors are distributed on the surface, it is possible to inscribe a cube inside the sphere with *all* of its vertices touching red. How large can we make β and still retain that property?

To solve this problem, we will generate a random cube that is inscribed on the sphere and show that it has positive probability of all its vertices touching red, under some conditions on β . We choose the cube *uniformly* over all spatial rotations of a fixed inscribed sphere. (See Chapter 21.) This symmetry assumption ensures that each vertex of the cube is individually uniform over the sphere.

Define two 8-dimensional random variables

- V represents the vertices of a randomly chosen cube inscribed in the sphere. ($V_1, V_2, \dots, V_8$ are V 's scalar components.)
- B represents the events that each vertex is touching blue, with component B_i being the event for vertex V_i .

By the Uniformity of the cube, each V_i has a Uniform Distribution over the sphere, so $\mathbb{E}(B_i) = \beta$

Let R be the event that all eight vertices are touching red. Then, $R = \varphi(B)$ where

$$\varphi(x) = \prod_{i=1}^8 (1 - x_i) = (1 - x_1)(1 - x_2) \cdots (1 - x_8),$$

where $\prod$ is the product, analogously to $\sum$ as a sum. We write this in *inline* form as

$$R := \prod_{i=1}^8 (1 - B_i) = (1 - B_1)(1 - B_2) \cdots (1 - B_8).$$

We can see that the event R only occurs if *none* of the B_i 's occur.

It follows that $1 - R$ occurs if *at least one* of the B_i 's occurs, so we have as a fact that

$$1 - R \leq \text{Sum}(B),$$

by the Ordering and Additivity Rules. Hence:

$$\mathbb{E}(1 - R) \leq \mathbb{E}(\text{Sum}(B)) = \text{Sum}(\mathbb{E}(B)) = 8\beta, \text{ which implies that } \mathbb{E}(R) \geq 1 - 8\beta.$$

It follows that $\mathbb{E}(R) > 0$ whenever $\beta < 1/8$. The probability of picking a cube whose vertices are all red is *positive*. An event having positive probability means that it is possible that the event occurs, so **some such cube must exist!**

Example 22.5 Accruing Rewards

A system evolves in discrete steps with initial state S_0 and state S_n after n steps. After at least 1 step, when the system is in state s , we obtain a reward $r(s)$, where r is a real-valued function defined on the set of states.

We calculate the *present value* of a reward after n steps by the reward scaled by α^n for some $0 < \alpha \leq 1$. This accounts for the fact that a given reward in the future is worth less than the same size reward in the present, due to e.g., inflation and our capacity to invest.

Our total reward T through n -steps can be expressed as

$$T = \sum_{i=1}^n \alpha^i r(S_i).$$

By the Scaling and Additivity Rules, the expected total reward is

$$\begin{aligned} \mathbb{E}(T) &= \mathbb{E}\left(\sum_{i=1}^n \alpha^i r(S_i)\right) \\ &= \sum_{i=1}^n \mathbb{E}(\alpha^i r(S_i)) \\ &= \sum_{i=1}^n \alpha^i \mathbb{E}(r(S_i)). \end{aligned}$$

Example 22.6 Random Sum

We have a collection of scalar random variables $X_1, X_2, \dots$, and a random natural number N . We choose N of the X_i 's and form the random sum

$$S = \sum_{i=1}^N X_i,$$

where S is 0 when $N = 0$.

If we *know* that $N = n$, S reduces to a sum of n terms. That is,

$$S \mid N = n = \sum_{i=1}^n X_i \mid N = n.$$

Applying the Substitution and Additivity Rules, we have

$$\begin{aligned} \mathbb{E}(S \mid N = n) &= \mathbb{E}\left(\sum_{i=1}^n X_i \mid N = n\right) \\ &= \sum_{i=1}^n \mathbb{E}(X_i \mid N = n). \end{aligned}$$

Notice that the observation $N = n$ applied on the left is maintained in every term on the right. Think of this as a calculation in the *context of* the knowledge that $N = n$.

Example 22.7 Infinite Sums

The Monotone Limits Rule allows us to exchange expectations and limits in some circumstances. We need some condition derived from this whenever we move a limit inside or outside an $\mathbb{E}$. A typical use case is with infinite sums.

Consider a random variable expressed as a sum over a collection of other random variables:

$$Z = \sum_{i=1}^{\infty} Y_i.$$

Such a sum may or may not be well-defined, but we can view Z as a limit of well-defined random variables:

$$Z = \lim_{n \rightarrow \infty} \sum_{i=1}^n Y_i.$$

Each of the sums in the limit is well defined, and we'd like to use the Additivity Rule to find $\mathbb{E}(Z)$. Unfortunately, the limit gets in the way of applying Additivity.

If, however, we have a condition like **fact** ($Y_i \geq 0$, for all i), the Monotone Limits Rule applies. (The limit exists here but may be infinite.) This lets us write

$$\begin{aligned}\mathbb{E}(Z) &= \mathbb{E} \left(\lim_{n \rightarrow \infty} \sum_{i=1}^n Y_i \right) \\ &= \lim_{n \rightarrow \infty} \mathbb{E} \left(\sum_{i=1}^n Y_i \right) \\ &= \lim_{n \rightarrow \infty} \sum_{i=1}^n \mathbb{E}(Y_i) \\ &= \sum_{i=1}^{\infty} \mathbb{E}(Y_i),\end{aligned}$$

where we apply Monotone Limits moving from the first to second equality and Additivity from the second to third.

We will not use this very often, but it is good to have in our back pocket for such situations.

22.2 The Mass-Balancing Equation

The **mass-balancing equation** embodies the idea that an expectation is a weighted average. We derived this first using the properties of risk-neutral prices (see equation 6.29), and this now extends to general random variables. We average the values of a random variable, weighting those values by probability mass at or near the value. The phrase “at or near” here addresses the difference between discrete values – with weight reflecting the mass *at* each specific point – and continuous values – with a weight reflecting the local density of mass (infinitesimally) *near* each point.

To start, assume that X is a discrete random variable and ψ is a compatible statistic. We want to compute $D_X(\psi) = \mathbb{E}(\psi(X))$. Earlier, we have seen that

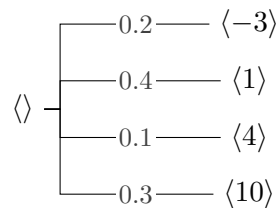
$$\mathbb{E}(\psi(X)) = \sum_u \mathbf{K}_X(u) \psi(u) = \sum_v \left(\sum_u \mathbf{K}_X(u) \{ \psi(u) = v \} \right) v = \sum_v \mathbf{K}_{\psi(X)}(v) v.$$

All of these equations express the expectation as a weighted average. Let's take them from right to left. In the third equality, the expectation is an average of the possible

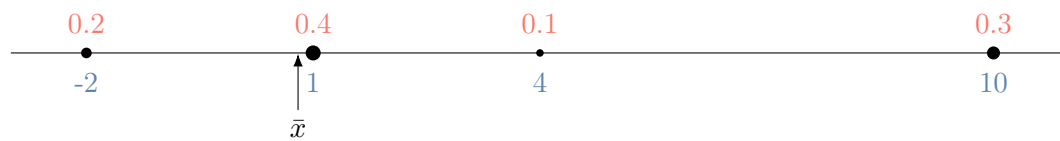
values of $\psi(X)$ weighted by the probabilities that $\psi(X)$ has each value. In the second equality, we get the probabilities $K_{\psi(X)}(v)$ by combining the branches in the Kind of X whose values map to v , adding their weights. This expresses our procedure for transforming Kinds, followed by taking the expectation. And in the first equality, we express the expectation as an average of the values of the statistic weighted by the probabilities of each value of the original random variable.

This is the *mass-balancing equation*. It expresses expectation as a weighted average that can be interpreted as the balancing point for a distribution of mass in space. The Distribution is a recipe for distributing 1 unit of mass across space, either placing discrete chunks of mass at specific points (discrete values) or smearing a continuous paste of mass with varying density near points (continuous values).

To understand this perspective, we start with a simple example, a random variable X with Kind



Interpreting the weights as chunks of mass at the locations specified by the corresponding values, we can visualize this Kind as a distribution of mass along a (massless) line with a fulcrum placed under the line at $\bar{x}$.



As we move the fulcrum left or right, the line will tip this way or that. At what position does it balance? Answer: $\mathbb{E}(X)$.

We can see why directly for any discrete Kernel K_X . The mass at a location u is $K_X(u)$, and if the fulcrum is at $\bar{x}$, the torque³ on the line from the mass at u has magnitude proportional to the product $K_X(u)(u - \bar{x})$, mass times distance from the fulcrum. The net torque on the line is therefore

$$\sum_u K_X(u)(u - \bar{x}) = \sum_u K_X(u)u - \sum_u K_X(u)\bar{x} = \sum_u K_X(u)u - \bar{x}$$

³The torque is a rotational force on the line caused by the gravitational force on the masses. Positive torque will produce a clockwise rotation and negative torque a counter-clockwise rotation.

because $\text{total}(\mathbf{K}_X) = \mathbf{D}_X(\text{const}_1) = 1$, and the line will be balanced when the net equals 0. This yields

$$\bar{x} = \sum_u \mathbf{K}_X(u) u = \mathbb{E}(X).$$

This argument works exactly as is for X of any dimension.

We can easily extend this argument to a general simple Kernel. For a discrete value $u \in \text{range}_d(X)$, the mass at u is $\mathbf{K}_X(u)$ and for a continuous value $u \in \text{range}_c(X)$, the mass *infinitesimally near* u is $du \mathbf{K}_X(u)$, an infinitesimal measure times the density as a mass per unit measure. Writing $\mathbf{K}_X$ in terms of its discrete and continuous parts $\mathbf{p}_X(u) = \mathbf{K}_X(u) \{u \in \text{range}_d(X)\}$ and $\mathbf{f}_X(u) = \mathbf{K}_X(u) \{u \in \text{range}_c(X)\}$, the net torque becomes

$$\begin{aligned} 0 &= \sum_u \mathbf{p}_X(u) (u - \bar{x}) + \int_u du \mathbf{f}_X(u) (u - \bar{x}) \\ &= \sum_u \mathbf{p}_X(u) u + \int_u du \mathbf{f}_X(u) u - \sum_u \mathbf{p}_X(u) \bar{x} - \int_u du \mathbf{f}_X(u) \bar{x} \\ &= \sum_u \mathbf{p}_X(u) u + \int_u du \mathbf{f}_X(u) u - \bar{x} \left(\sum_u \mathbf{p}_X(u) + \int_u du \mathbf{f}_X(u) \right) \\ &= \sum_u \mathbf{p}_X(u) u + \int_u du \mathbf{f}_X(u) u - \bar{x}, \end{aligned}$$

because $\text{total}(\mathbf{K}_X) = \mathbf{D}_X(\text{const}_1) = 1$. Hence,

$$\bar{x} = \sum_u \mathbf{p}_X(u) u + \int_u du \mathbf{f}_X(u) u = \mathbb{E}(X).$$

The expectation is an average of the statistic's values weighted by the mass of probability at or near the values that produce them.

And this holds as well for transforms of X by arbitrary statistics generalizing equation (6.31), and after constraining with arbitrary observations, where we replace X with $X \mid \mathcal{C}$. For instance,

$$\mathbb{E}(X \mid \mathcal{C}) = \sum_u \mathbf{p}_{X \mid \mathcal{C}}(u) u + \int_u du \mathbf{f}_{X \mid \mathcal{C}}(u) u$$

and similarly in other cases.

The Mass-Balancing Equation. Let X be a random variable, ψ is a compatible statistic, and $\mathcal{C}$ an arbitrary observation.

If X has a *simple* Kernel K_X , then

$$\begin{aligned} D_{X|\mathcal{C}}(\psi) &= \mathbb{E}(\psi(X) \mid \mathcal{C}) \\ &= \sum_u K_{X|\mathcal{C}}(u) \{u \in \text{range}_d(X)\} \psi(u) + \int_v dv K_{X|\mathcal{C}}(v) \{v \in \text{range}_c(X)\} \psi(v) \end{aligned} \quad (22.13)$$

$$= \sum_u p_X(u) \psi(u) + \int_v dv f_X(v) \psi(v). \quad (22.14)$$

This is the **mass-balancing equation**, which establishes the expectation as the *center of mass* for the distribution of probability mass over values.

Here, $p_X(u) = K_X(u) \{u \in \text{range}_d(X)\}$ is the mass function and $f_X(u) = K_X(u) \{u \in \text{range}_c(X)\}$ is the density function of K_X .

If X has a *split* Kernel K_X , then the mass-balancing equation becomes

$$\begin{aligned} D_{X|\mathcal{C}}(\psi) &= \mathbb{E}(\psi(X) \mid \mathcal{C}) \\ &= \sum_u \int_v dv K_{X|\mathcal{C}}(u, v) \psi(u, v). \end{aligned} \quad (22.15)$$

Here, we have arbitrarily ordered the components of X with the discrete components then the continuous components. In general, we would interleave sums and integrals according to each component's type.

When $\psi = \text{id}$ and $\mathcal{C} = \top$, the mass-balancing equation for simple Kernels becomes a generalization of equation (6.29)

$$\mathbb{E}(X) = \sum_s p_X(s) s + \int_t dt f_X(t) t. \quad (22.16)$$

Again, the expectation is an average: the average of the random variable's values weighted by the probability mass at or near each value. We will see a more general derivation of equation (22.13) in Chapter 25.

Example 22.8 Uniform on an Interval

Let U be a continuous random variable with Kernel

$$K_U(t) = \frac{1}{b-a} \{a \leq t \leq b\}.$$

which has Kernel $K_U(t) = \frac{1}{b-a} \{a \leq t \leq b\}$. As in Chapter 21, we call this a $\text{Uniform}\langle a, b \rangle$ Distribution because it smears probability uniformly across the interval $[a, b]$. Notice that if $b - a < 1$, then the Kernel has value > 1 . This is fine because the Kernel returns a probability *density*, not a probability mass. The total mass is still 1.

Looking at the Kernel, we can see that the balancing point $\mathbb{E}(U)$ must lie in the middle of that uniform smearing of mass, at $(a + b)/2$. Indeed, the mass-balancing equation tells us that

$$\begin{aligned} D_U(\psi) &= \int_t dt \frac{1}{b-a} \{a \leq t \leq b\} \psi(t) \\ &= \frac{1}{b-a} \int_{t=a}^b dt \psi(t), \end{aligned}$$

which is the average value of the statistic ψ over $[a, b]$. When $\psi = \text{id}$, we get, as we intuited,

$$\begin{aligned} \mathbb{E}(U) &= \frac{1}{b-a} \int_{t=a}^b dt t \\ &= \frac{1}{b-a} \left. \frac{1}{2} t^2 \right|_{t=a}^b \\ &= \frac{1}{2} \frac{b^2 - a^2}{b-a} \\ &= \frac{1}{2} \frac{(a+b)(b-a)}{b-a} = \frac{a+b}{2}, \end{aligned}$$

Example 22.9 Counting Heads

Equation (19.52) shows how to compute the expectations of some statistics transforming an independent join. This relies on removing terms from a sum that do not depend on the summation index.

Suppose that $Z = X \star Y$ is discrete. Then:

$$\begin{aligned}
 \mathbb{E}((\psi \otimes \varphi)(Z)) &= \sum_i \sum_j \mathsf{K}_Z(i, j) \psi(i) \varphi(j) \\
 &= \sum_i \sum_j \mathsf{K}_X(i) \mathsf{K}_Y(j) \psi(i) \varphi(j) \\
 &= \sum_i \mathsf{K}_X(i) \psi(i) \sum_j \mathsf{K}_Y(j) \varphi(j) \\
 &= \left(\sum_i \mathsf{K}_X(i) \psi(i) \right) \left(\sum_j \mathsf{K}_Y(j) \varphi(j) \right).
 \end{aligned}$$

The third equality is the key to seeing this. The two groups of terms each only depend on one index. So we can pull all the i -indexed terms out of the j -sum, and then that sum is independent of i . The result is a product of the two sums. This is a generalization of the simple algebraic fact that $ab + ac + bc + bd = (a + b)(c + d)$.

The same idea works in the continuous case. Let $W = U \star V$ be continuous.

$$\begin{aligned}
 \mathbb{E}((\psi \otimes \varphi)(W)) &= \int_a da \int_b db \mathsf{K}_W(a, b) \psi(a) \varphi(b) \\
 &= \int_a da \int_b db \mathsf{K}_U(a) \mathsf{K}_V(b) \psi(a) \varphi(b) \\
 &= \int_a da \mathsf{K}_U(a) \psi(a) \int_b db \mathsf{K}_V(b) \varphi(b) \\
 &= \left(\int_a da \mathsf{K}_U(a) \psi(a) \right) \left(\int_b db \mathsf{K}_V(b) \varphi(b) \right).
 \end{aligned}$$

It works in the same way for any simple Kernel, though the expressions gets a bit messy. And for a split Kernel as well. For instance, if $C = A \star B$ where A is discrete and B continuous, we have

$$\begin{aligned}
 \mathbb{E}((\psi \otimes \varphi)(C)) &= \sum_a \int_b db \mathsf{K}_C(a, b) \psi(a) \varphi(b) \\
 &= \sum_a \int_b db \mathsf{K}_A(a) \mathsf{K}_B(b) \psi(a) \varphi(b) \\
 &= \sum_a \mathsf{K}_A(a) \psi(a) \int_b db \mathsf{K}_B(b) \varphi(b) \\
 &= \left(\sum_a \mathsf{K}_A(a) \psi(a) \right) \left(\int_b db \mathsf{K}_B(b) \varphi(b) \right).
 \end{aligned}$$

To make this more concrete, consider a case where the statistic does not separate cleanly into a product of terms for each join component. Specifically, consider n independent flips of a coin where 1 represents heads and 0 tails. Let $C = C_1 \star C_2 \star \dots \star C_n$ be the flips where all C_i 's have a common Kernel

$$K_{C_i}(x) = (1 - p) \{x = 0\} + p \{x = 1\} = p^x (1 - p)^{1-x} \{x = 0 \vee x = 1\},$$

for some $0 \leq p \leq 1$. Equation (19.44) tells us that for $\langle x_1, \dots, x_n \rangle \in \{0, 1\}^n$

$$\begin{aligned} K_C(x_1, x_2, \dots, x_n) &= K_{C_1}(x_1) K_{C_2}(x_2) \dots K_{C_n}(x_n) \\ &= p^{x_1} (1 - p)^{1-x_1} p^{x_2} (1 - p)^{1-x_2} \dots p^{x_n} (1 - p)^{1-x_n} \prod_{i=1}^n \{x_i = 0 \vee x_i = 1\} \\ &= p^{\sum_i x_i} (1 - p)^{n - \sum_i x_i} \prod_{i=1}^n \{x_i = 0 \vee x_i = 1\}. \end{aligned}$$

The product of indicators here ensures that all the x_i 's are either 0 or 1.

Let $H = \text{Sum}(C)$ represent the number of heads in the n flips. The mass-balancing equation gives us

$$\begin{aligned} \mathbb{E}(H) &= \sum_{x_1} \dots \sum_{x_n} K_C(x_1, \dots, x_n) \text{Sum}(x_1, \dots, x_n) \\ &= \sum_{x_1=0}^1 \dots \sum_{x_n=0}^1 p^{\sum_i x_i} (1 - p)^{n - \sum_i x_i} \sum_i x_i \\ &= \sum_{k=0}^n \binom{n}{k} p^k (1 - p)^{n-k} k. \end{aligned}$$

See Chapter 18.

The last equation follows by a counting argument. There are exactly $\binom{n}{k}$ sequences in $\{0, 1\}^n$ with sum k , each corresponding to a distinct subset of $[1 \dots n]$, and all of these sequences have the same weight in the sum $p^k (1 - p)^{n-k} k$. Now, using the fact that $\binom{n}{k} k = n \binom{n-1}{k-1}$, we have

$$\begin{aligned} \mathbb{E}(H) &= \sum_{k=0}^n \binom{n}{k} p^k (1 - p)^{n-k} k \\ &= \sum_{k=1}^n \binom{n}{k} p^k (1 - p)^{n-k} k \quad (k = 0 \text{ term is zero}) \end{aligned}$$

$$\begin{aligned}
&= n \sum_{k=1}^n \binom{n-1}{k-1} p^k (1-p)^{n-k} \\
&= np \sum_{k=1}^n \binom{n-1}{k-1} p^{k-1} (1-p)^{n-1-(k-1)} \\
&= np \sum_{j=0}^{n-1} \binom{n-1}{j} p^j (1-p)^{n-1-j} && \text{(change variables } k-1 \rightarrow j) \\
&= np(p+1-p)^{n-1} && \text{(by the Binomial theorem)} \\
&= np,
\end{aligned}$$

See Chapter 18 and Example 17.30 for more on the Binomial Theorem.

Example 22.10 Truncated Waiting Time

A customer waits in a phone queue for service. The time until the next agent will be available is represented by a random variable T , measured in minutes, with distribution

$$D_T(\psi) = \int_t dt \frac{1}{30} e^{-t/30} \{t > 0\} \psi(t).$$

However, the customer will wait at most ℓ minutes before leaving. What is the customer's expected waiting time? What is the probability that the customer leaves without being served?

First, for comparison, let's compute the expected time the customer would wait if they remained in the queue until being served ($\ell = \infty$). This is just $\mathbb{E}(T)$:

$$\begin{aligned}
\mathbb{E}(T) &= \int_t dt \frac{1}{30} e^{-t/30} \{t > 0\} t \\
&= \int_{t=0}^{\infty} dt \frac{1}{30} t e^{-t/30} \\
&= 30 \int_{t=0}^{\infty} dt \frac{1}{30} (t/30) e^{-t/30} \\
&= 30 \int_{u=0}^{\infty} du u e^{-u} \\
&= 30.
\end{aligned}$$

Let W represent the customer's actual waiting time. We have that $W = \varphi(T)$,

where $\varphi(t) = t \{t < \ell\} + \ell \{t \geq \ell\}$. We could also write this in inline form as

$$W = T \{T < \ell\} + \ell \{T \geq \ell\}, \quad (22.17)$$

keeping in mind that the underlying statistic in this transform is φ .

W has a simple Kernel with $\text{range}_d(W) = \{\ell\}$ and $\text{range}_c(W) = (0, \ell)$. Chapter 25 will show several techniques that can be used to find D_W . Here, we use the relationship between the Distributions of a random variable and a transformed random variable that we saw earlier in equation (19.42):

$$\begin{aligned} D_W(\psi) &= D_T(\psi \circ \varphi) \\ &= \int_t \frac{1}{30} e^{-t/30} \{t > 0\} \psi(\varphi(t)) \\ &= \int_t \frac{1}{30} e^{-t/30} \{t > 0\} \psi(t \{t < \ell\} + \ell \{t \geq \ell\}) \\ &= \int_{t=0}^{\ell} dt \frac{1}{30} e^{-t/30} \psi(t) + \int_{t=\ell}^{\infty} dt \frac{1}{30} e^{-t/30} \psi(\ell) \\ &= \int_{t=0}^{\ell} dt \frac{1}{30} e^{-t/30} \psi(t) + \psi(\ell) \int_{t=\ell}^{\infty} dt \frac{1}{30} e^{-t/30} \\ &= \int_t \frac{1}{30} e^{-t/30} \{0 < t < \ell\} \psi(t) + \psi(\ell) e^{-\ell/30}. \end{aligned}$$

Hence,

$$\begin{aligned} p_W(s) &= e^{-\ell/30} \{s = \ell\} \\ f_W(s) &= \frac{1}{30} e^{-t/30} \{0 < t < \ell\}, \end{aligned}$$

and the equation for $D_W(\psi)$ is just the mass-balancing equation:

$$D_W(\psi) = \int_t \frac{1}{30} e^{-t/30} \{0 < t < \ell\} \psi(t) + e^{-\ell/30} \psi(\ell). \quad (22.18)$$

Computing $\mathbb{E}(W) = D_W(\text{id})$ gives

$$\begin{aligned} \mathbb{E}(W) &= \int_{t=0}^{\ell} dt \frac{1}{30} e^{-t/30} t + e^{-\ell/30} \ell \\ &= 30 - \int_{t=\ell}^{\infty} dt \frac{1}{30} e^{-t/30} t + e^{-\ell/30} \ell \\ &= 30 - (30 + \ell) e^{-\ell/30} + e^{-\ell/30} \ell \end{aligned}$$

$$= 30(1 - e^{-\ell/30}),$$

using our expression for $\mathbb{E}(T)$ and the Appendix H (Help Sheet). The result makes sense: as ℓ grows, the expected waiting time approaches the $\ell = \infty$ case, and for small ℓ it is substantially below 30.

Example 22.11 The Price of Continuity

We use continuous random variables because it makes it easier in practice to model and calculate in some cases. The cost of continuity is that for certain questions, we get a surprising answer.

The key issue is that changing the value of an integrand on a set of values of smaller dimension than the domain of integration *does not change the integral*.

Thus, if we change the integrand on isolated points or on a line in 2-dimensions or a surface in 3 dimensions we get the same integral.

For instance, if X , Y , and Z are respectively 1-, 2-, and 3-dimensional random variables, then:

$$\begin{aligned} \int_r dr \, K_X(r) \{r = u\} &= 0 \\ \int_s ds \, K_Y(s) \{s = u\} &= 0 \\ \int_t dt \, K_Z(t) \{t = w\} &= 0, \end{aligned}$$

because the integrands differ from 0 only at the isolated points u , v , and w . Put another way, there is no *mass* at these isolated points. To get non-zero mass, we need to add up infinitesimals over an uncountable set of values; adding up an infinitesimal mass at a single point gives us no finite mass.

The same story holds for larger sets of smaller dimension than the random variable. For example, let a, b, c, d be scalars; then $\mathcal{L} = \{\langle s_1, s_2 \rangle \mid as_1 + bs_2 = c\}$ forms a line in the plane, and $\mathcal{P} = \{\langle t_1, t_2, t_3 \rangle \mid at_1 + bt_2 + ct_3 = d\}$ forms a plane in space. We have

$$\int_s ds \, K_Y(s) \mathcal{L}(s) = 0 = \int_t dt \, K_Z(t) \mathcal{P}(t).$$

Constraining with Observations

23

Chapter

Contents

23.1 Review: Conditions, Events, Boolean Expressions	941
23.2 Technique: The Logic of Events	945
23.3 Constraining with Observations	953
23.4 The Expectation Updating Equation	957
23.5 Conditional Distributions, Conditional Expectations	959

Key Take Aways

We constrain with an observation to update our knowledge and predictions for new information, real or hypothetical. When we observe (or hypothetically choose to believe) that a particular condition is true, we *erase all the branches that are inconsistent with that condition*. The **relative weights of the remaining branches do not change**.

We express observational constraints in a variety of ways. Each observation has an associated **condition**, a Boolean-valued statistic, where we take 0 to mean false and 1 to mean true. Put another way: a condition is a statistic that is an *indicator function*.

We can also specify an observation through a *Boolean expression* stated in terms of random variables. Each Boolean expression corresponds to a condition that is compatible with the variables in the expression. For general reference, if $\mathcal{C}$ is a Boolean expression, write $\text{cond}_{\mathcal{C}}$ for the associated condition, though in practice we may give that condition a name or use an anonymous function. Moreover, if $\mathcal{C}$ is a Boolean expression, then $\{\mathcal{C}\}$ denotes the *event* that the condition is true, a random variable that can have values 0 or 1.

A useful reasoning technique for doing our calculations is to convert logical relations between conditions/Boolean expressions into arithmetic relations between

events. An important instance of this technique is that **logically equivalent conditions produce equal events**. We can also relate events via the standard logical connectives (and, or, exclusive-or, not).

If X is a random variable and $\mathcal{C}$ an observation with corresponding condition $\text{cond}_{\mathcal{C}}$ and if $D_X(\text{cond}_{\mathcal{C}}) > 0$, then the Distribution and Kernel of X given $\mathcal{C}$ is

$$D_{X|\mathcal{C}}(\psi) = \frac{D_X(\psi \cdot \text{cond}_{\mathcal{C}})}{D_X(\text{cond}_{\mathcal{C}})} \quad (23.1)$$

$$K_{X|\mathcal{C}}(u) = \frac{K_X(u) \text{cond}_{\mathcal{C}}(u)}{\text{total}(K_X \text{cond}_{\mathcal{C}})}. \quad (23.2)$$

If $\xi = \text{cond}_{\mathcal{C}}$, we can also write $K_{X|\xi}$ for $K_{X|\mathcal{C}}$ if we like.

A direct consequence of equation (23.1) is the Expectation Updating equation

$$\mathbb{E}(Y | \mathcal{C}) = \frac{\mathbb{E}(Y \{ \mathcal{C} \})}{\mathbb{E}(\{ \mathcal{C} \})}, \quad (23.3)$$

for a random variable Y and an observation $\mathcal{C}$ with $\mathbb{E}(\{ \mathcal{C} \}) > 0$.

Observations arise in two ways. We can obtain partial information by observing the values of some random variables as the random system evolves. Or we can *imagine* that we have such partial information as we consider hypotheticals in our planning or analysis. However they arise, observational constraints are assertions about our *knowledge state*, so constraining with an observation can affect our predictions.

As defined in Chapter 5, observations are events that we assert (in reality or hypothetically) have *occurred*. We put these events on the right side of the “given” bar $|$ to indicate what we take as known. Constraining a random variable X with an observation $\mathcal{C}$ produces a *new*, related random variable $X | \mathcal{C}$. The observational constraint asserts that $\mathcal{C}$ is true; then the random variables X and $X | \mathcal{C}$ will have the same *values*, but their distributions may differ. As before, it is often sufficient to use the underlying condition to express the constraint, but we will frequently use Boolean expressions to express the condition in practice.

The key point to remember is that **constraining with an observation simply erases the branches of our tree that are inconsistent with the constraint**. The relative sizes of the weights on different branches *do not change*, even after renormalizing to canonical form. We saw how this manifested for Distributions and Kernels in equations (19.56) and (19.55). We will revisit these ideas here.

23.1 Review: Conditions, Events, Boolean Expressions

We start by reviewing a few concepts and definitions from Part I.

Definition 57. An **event** is a random variable whose possible values are 0 and 1.

When an event has the value 1, we say that the event *occurred*, and when it has the value 0, we say that the event did not occur.

We use events to represent observable outcomes that may or may not occur. Events typically arise by transforming a random variable by a *condition*, and such an event occurs when the condition is true. The expectation of an event is a number between 0 and 1 that measures our prediction about whether the event will occur. We define this to be the **probability** of the event.

Definition 58. A **condition** is a statistic that can only have values 0 or 1. (That is, a condition is a statistic that is an *indicator function*.)

We think of a condition as returning a *Boolean value*, where 0 represents “false” and 1 represents “true.”

Example 23.1. Conditions are statistics that ask yes-no questions.

Some 1-dimensional conditions:

- Is the value positive? $\langle x \rangle \mapsto \{x > 0\}$
- Is the value an integer with absolute value ≤ 10 ? $\langle k \rangle \mapsto \{k \in [-10..10]\}$
- Is the value smaller than its square? $\langle x \rangle \mapsto \{x \leq x^2\}$

Some 2-dimensional conditions:

- Are two values within 1 of each other? $\langle x, y \rangle \mapsto \{|x - y| \leq 1\}$
- Does a point lie within the unit circle? $\langle x, y \rangle \mapsto \{x^2 + y^2 \leq 1\}$
- Are two values the same? $\langle x, y \rangle \mapsto \{x = y\}$

If X is a random variable of any dimension and ξ is a compatible condition, then $\xi(X)$ is an event. And in general, given an event V of interest, we can typically write it as $V = \xi(X)$ for some relevant random variable we have defined, e.g., the “data” random variable from our schema.¹

While we can always define a function to specify conditions; in practice, we often consider enough different conditions, even for a single calculation, that defining separate functions for them all would lead to a proliferation of new names to remember.

¹Recall that for any random variables $A, B, C, \dots$, we can always find a random variable X for which these are projections of X .

So it is common to *inline* the condition, much as we do with other statistics (see discussion on page 131). An inlined condition is a *Boolean expression* that specifies the condition in terms the values of one or more random variables. A few simple examples will illustrate the idea.

Consider the two-dimensional random variable X with scalar components $\langle X_1, X_2 \rangle$ and the three-dimensional random variable T with scalar components $\langle U, V, W \rangle$. When we define a compatible condition, we also get a corresponding Boolean expression and an event. Here are several examples.

Condition	Boolean Expression	Event
$\psi(a) = \{a_1 > 0\}$	$X_1 > 0$	$\{X_1 > 0\}$
	$U > 0$	$\{U > 0\}$
$\varphi(b) = \{1 < b_2 - b_1 \leq 2\}$	$1 < X_2 - X_1 < 2$	$\{1 < X_2 - X_1 < 2\}$
	$1 < V - U < 2$	$\{1 < V - U < 2\}$
$\langle c \rangle \mapsto \{\text{Sum}(c) > 0\}$	$X_1 + X_2 > 0$	$\{X_1 + X_2 > 0\}$
	$\text{Sum}(T) > 0$	$\{\text{Sum}(T) > 0\}$
$\xi(s) = \{s_1 > 3 \wedge s_3 < -3\}$	$U > 3 \wedge W < -3$	$\{U > 3 \wedge W < -3\}$
$\langle s \rangle \mapsto \{s_1^2 + s_2^2 + s_3^2 \leq 1\}$	$U^2 + V^2 + W^2 \leq 1$	$\{U^2 + V^2 + W^2 \leq 1\}$

The first condition asks “Is the first component of this tuple positive?” This is compatible with both X and T , and for each we get the corresponding Boolean expression and the event derived from it. (Thus, different Boolean expressions can represent the same condition on different random variables.) Note that the events here are defined as

$$\{X_1 > 0\} = \psi(X) \quad \text{and} \quad \{U > 0\} = \psi(T),$$

the transforms of the random variables by the underlying condition. The inline notation for the events is both clear and mnemonic. Both the second and third conditions are compatible with X and T . The second asks “Is the absolute difference between the first two components between 1 (exclusive) and 2?” The third asks if the sum of components is positive. Notice that we can write the Boolean expression and events in terms of the components themselves, as for X , or in terms of **Sum** acting on the random variable, as for T . Both approaches are equivalent. The condition in the third case is specified by an anonymous function, but that’s fine. The last two conditions are only compatible with T . For instance, the last condition asks if a point

lies inside the unit sphere.

The key points here are that (i) the Boolean expression with particular random variables implicitly specifies the underlying condition by inlining; (ii) the event corresponding to a Boolean expression is written as the expression surrounded by *braces*; (iii) the event is a *random variable*, defined as a transform, that has possible values 0 and 1.

We use events and conditions frequently to express possible outcomes related to random variables of interest, so Boolean expressions are a convenience that save the boilerplate of defining an explicit function for every condition we consider. It simplifies the notation and can make reasoning easier. A Boolean expression implicitly defines a condition – a function that takes values to values – but is explicitly tied to particular random variables. Boolean expressions that define the same condition but relate to different random variables are distinct. For instance, $X_1 > 0$ and $U > 0$ both correspond to the condition ψ above, but because X and T are different random variables, these expressions are not the same.

Our main uses for Boolean expressions are to define events related to particular random variables (by surrounding the expression with braces) and to specify the observation associated with those events. When considering Boolean expressions in the abstract, we refer to them with script-style letters, like $\mathcal{C}$, the same style we use to depict sets. In this case, the corresponding event is written $\{\mathcal{C}\}$ and the underlying condition $\text{cond}_{\mathcal{C}}$.

A **Boolean expression** is an assertion about the values of some random variables, expressed as a logical proposition.

A simple expression is a statement involving zero or more random variables, like $X > 0$ or $|X - Y| \leq 3$. The trivial statement $\top$, for “true,” represents what we *know* to be true before our random system begins its evolution. (We colloquially refer to $\top$ as *common knowledge*.)

More generally, a Boolean expressions combines simple expressions with logical connectives like $\wedge$ (logical and) and $\vee$ (logical or).

If $\mathcal{C}$ is a Boolean expression, then $\{\mathcal{C}\}$ denotes the event that the condition is true, a random variable. The braces around the condition are a visual reminder that this is an event determined by the specific expression.

If the Boolean expression $\mathcal{C}$ is written in terms of the random variable X (of any dimension), we can write the condition implicitly defined by $\mathcal{C}$ as $\text{cond}_{\mathcal{C}}$, though we are free to give it a name as well. Then, $\{\mathcal{C}\} = \text{cond}_{\mathcal{C}}(X)$.

Expressing events in terms of Boolean expressions makes it easier to recall what the event means and to reason about the event. For example, we might have events $\{X = 2\}$, $\{Y > 4 \wedge |Y - Z| \leq 2\}$, $\{U > -2\}$, $\{A = C \vee C = 17\}$, or $\{D \bmod 5 = 0\}$. The event $\{X = 2\}$, for instance, is the random variable that is 1 when X equals 2 and 0 otherwise. The event $\{D \bmod 5 = 0\}$ is equal to 1 when D is divisible by 5 and 0 otherwise. And so on. If V is itself an event, then $V = \{V = 1\}$.

Puzzle 167. Let events $B_1, B_2, \dots, B_8$ represent the bits (from least to most significant) in a random byte recorded on a digital channel. What do the following random variables represent:

$$\sum_{k=1}^8 \{B_k = 1\}$$

$$\{B_1 = 1\}\{B_2 = 0\}\{B_3 = 0\}\{B_4 = 1\}$$

$$\sum_{k=1}^8 2^{k-1}\{B_k = 1\}$$

Are any of them events?

Notice that for each k the events $\{B_k = 1\}$ and $\{B_k = 0\}$ can be expressed in terms of B_k . Indeed, it must be true $\{B_k = 1\} = B_k$ and $\{B_k = 0\} = 1 - B_k$. Can you explain why?

With this infrastructure in hand, we can now describe *observations* formally, following the definition in Chapter 5.

Definition 59. An **observation** is an event that we interpret as if **the event has in fact occurred**.

We can denote an observation by the event, by a Boolean expression that generates that event, or by a condition when the random variables involved are clear from context.

An observation can only appear, in any manifestation, on the right side of the $|$ bar.

Observations can arise from an actual observation or from a hypothetical, where we consider what would happen *if* we knew that some condition is true.

We use observations to annotate objects with the appropriate knowledge state. If X is a scalar random variable, for instance, then $X | X > 0$ is the related random

variable with the constraint imposed. Its Distribution is $D_{X|X>0}$; its Kernel is $K_{X|X>0}$, and its expectation is $\mathbb{E}(X | X > 0)$. Such quantities are meaningful and well-defined whether the condition $X > 0$ is a real observation or a hypothetical consideration. Indeed, we treat real and hypothetical information in the same way, with one exception. The value of the random variable $X | X > 0$ is the same as the value of X as long as the condition is true, but $X | X > 0$ is undefined² if the constraint is false.

²Analogously to the empty FRP empty.

Our primary use for observation is in updating our predictions, so we write Distributions $D_{X|X>0}$, Kernels $K_{Z|Z>0 \wedge X<1}$, expectations $\mathbb{E}(Y | Y \leq 4)$, and such. Random variables constrained with an observations (after the given bar) can appear in any labels we use. Note that for Distributions and Kernels,

$$K_{Y|X=a}(y) = K_{Y|X}(y | a) \quad (23.4)$$

$$DY | X = a(\psi) = D_{Y|X}(\psi | a). \quad (23.5)$$

That is, for observations that some random variables are equal to particular values, the constrained Kind/Kernel/Distribution is equivalent to the conditional Kind/conditional Kernel/conditional Distribution. In this case, we generally prefer the forms on the right, as they are more concise and easier to write.

23.2 Technique: The Logic of Events

A powerful technique for reasoning about probabilities and random systems is to use the logic of events. We frequently express events as the transform of some random variables by a condition, an indicator function. So, we can exploit the logic of indicators to relate events, analogously to Table 12.1 in Interlude F. This is particularly effective when dealing with events expressed as Boolean expressions, because logically related expressions correspond to logically related events. Expressing events in terms of conditions and Boolean expressions is more than just a convenience, it makes it easier to *reason about* the events.

The first and most basic principle in this technique is

Logically equivalent expressions/conditions produce equal events.

So for instance, if A is a scalar random variable, the events $\{A^2 \leq 2A + 3\}$ and $\{|A - 1| \leq 2\}$ are equal. To see this, we can see that the two Boolean expressions are

logically equivalent:

$$\begin{aligned}
 A^2 \leq 2A + 3 &\iff A^2 \leq 2A - 1 + 4 \\
 &\iff A^2 - 2A + 1 \leq 4 \\
 &\iff (A - 1)^2 \leq 4 \\
 &\iff |A - 1| \leq 2.
 \end{aligned}$$

The expressions defining two logically equivalent may look very different, and may even involve different random variables. We can use our knowledge of the particular random system in establishing equivalence. We call this technique **event reduction** because we reduce one event to another by logical reasoning.

Example 23.2. Let A be a random variable whose Kind `uniform(-2, -1, 0, 1, 2)`, or equivalently, whose Distribution is `Uniform([-2..2])`. Define random variable $D = A^2$, the transform of A with the statistic $\langle x \rangle \mapsto x^2$. Find K_D .

Notice that $K_D(d)$ is the probability associated with the value d , so $K_D(d) = \mathbb{E}(\{D = d\})$. To calculate this expectation, we can reduce the event $\{D = d\}$ to a form whose expectation we can find. Specifically, we translate the event into a form that relates to A using logical equivalence of the conditions to establish equality of the events:

$$\begin{aligned}
 \{D = 0\} &= \{A = 0\} \\
 \{D = d\} &= \{A = \sqrt{d} \vee A = -\sqrt{d}\} \\
 &= \{A = \sqrt{d}\} + \{A = -\sqrt{d}\} \quad \text{for } d \neq 0.
 \end{aligned}$$

So,

$$\begin{aligned}
 \mathbb{E}\{D = 0\} &= \mathbb{E}\{A = 0\} = \frac{1}{5} \\
 \mathbb{E}\{D = d\} &= \mathbb{E}\left(\{A = \sqrt{d}\} + \{A = -\sqrt{d}\}\right) \\
 &= \mathbb{E}\left(\{A = \sqrt{d}\}\right) + \mathbb{E}\left(\{A = -\sqrt{d}\}\right) \\
 &= \frac{2}{5} \quad \text{for } d \in \{1, 4\},
 \end{aligned}$$

because the possible value of D are 0, 1, and 4. Hence,

$$\begin{aligned} K_D(d) &= \frac{1}{5} \{d = 0\} + \frac{2}{5} \{d = 1 \vee d = 4\} \\ D_D(\psi) &= \frac{1}{5} \psi(0) + \frac{2}{5} \psi(1) + \frac{2}{5} \psi(4). \end{aligned}$$

Example 23.3 Sum of Two Dice

We roll two balanced six-sided dice, one red and one blue. Let R and B represent the values of the two dice. If we want to find the Kernel of $R + B$, we want to find the probability that $R + B = k$. This condition is logically equivalent to the assertion that

$$\text{for some } j \in [1 \dots 6], R = j \wedge B = k - j.$$

Hence, we can write

$$\{R + B = k\} = \sum_{j=1}^6 \{R = j \wedge B = k - j\}.$$

First, observe that at most one of the events in the sum can occur. Thus, the summation corresponds to the existential qualifier “for some j ” because the event on the right occurs only if exactly one of the events in the sum occurs. Second, the benefit of this reduction is that the event on the left couples the values of R and B , but the events on the right separate them. By the Expectation Additivity Rule, we have

$$\begin{aligned} \mathbb{E}(\{R + B = k\}) &= \mathbb{E} \left(\sum_{j=1}^6 \{R = j \wedge B = k - j\} \right) \\ &= \sum_{j=1}^6 \mathbb{E}(\{R = j \wedge B = k - j\}) \\ &= \sum_{j=1}^6 K_{RB}(j, k - j). \end{aligned}$$

So if we know the distribution of $\langle R, B \rangle$, we can find this. And in fact we do if we assume that $\langle R, B \rangle$ is an independent join:

$$\begin{aligned}
 K_{R+B}(k) &= \mathbb{E}(\{R + B = k\}) \\
 &= \sum_j K_{RB}(j, k - j) \\
 &= \sum_j K_R(j) K_B(k - j) \\
 &= \sum_j \frac{1}{6} \{j \in [1 \dots 6]\} \frac{1}{6} \{k - j \in [1 \dots 6]\} \\
 &= \frac{1}{36} \sum_{j=1}^6 \{k - j \in [1 \dots 6]\} \{k \in [2 \dots 12]\} \\
 &= \frac{1}{36} \sum_{j=\max(1, k-6)}^{\min(6, k-1)} \{k \in [2 \dots 12]\} \\
 &= \frac{\min(6, k-1) - \max(1, k-6) + 1}{36} \{k \in [2 \dots 12]\}.
 \end{aligned}$$

Example 23.4 Distance to Closest Point

We select a collection of n random points in the plane $P_1, P_2, \dots, P_n$, and let R be the distance from the origin to the point closest to the origin. We might want to compute the probability that $R \leq r$ for some $r > 0$.

Let $N_r = \sum_{i=1}^n \{|P_i| \leq r\}$, where $|P_i|$ is the distance of P_i from the origin. N_r counts the number of points within distance r of the origin. We have $R \leq r$ if and only if $N_r > 0$, so

$$\{R \leq r\} = \{N_r > 0\}.$$

Under some models, $\{N_r > 0\}$ is easy to compute. For example if all the P_i 's have the same Distribution and are independent, then

$$\begin{aligned}
 \mathbb{E}(\{N_r > 0\}) &= 1 - \mathbb{E}(\{N_r = 0\}) \\
 &= 1 - \mathbb{E}\left(\prod_{i=1}^n \{|P_i| > r\}\right) \\
 &= 1 - \mathbb{E}(\{|P_1| > r\})^n,
 \end{aligned}$$

which we can compute from the Distribution of P_1 . In the first two equations,

we used the logic of events discussed below. The first equality comes from the “logical not” $\{N_r > 0\} = 1 - \{N_r = 0\}$. The second equality comes from the “logical and” of all the conditions $|P_i| > r$ over $i \in [1 \dots n]$. The final equality uses the independent join equation (19.52).

A second principle is that if $\mathcal{C}_1$ *logically implies* $\mathcal{C}_2$, then $\{\mathcal{C}_1\} \leq \{\mathcal{C}_2\}$, meaning that when $\{\mathcal{C}_1\}$ occurs, we know also that $\{\mathcal{C}_2\}$ also occurs. This actually generalizes the first principle because if each condition implies the other, we get logical equivalence and equality of the corresponding events.

Example 23.5. If X is a scalar random variable and $s < t$, then logically

$$X \leq s \text{ implies that } X \leq t,$$

but the two expressions are not logically equivalent. Hence, it is a fact that

$$\{X \leq s\} \leq \{X \leq t\},$$

so by the Expectation Ordering Rule

$$\mathbb{E}(\{X \leq s\}) \leq \mathbb{E}(\{X \leq t\}).$$

Referring to Example 19.9, this implies that $s < t$ implies that $F_X(s) \leq F_X(t)$, where F_X is the CDF (or Cumulative Distribution Function) of X . The CDF is thus a monotone function. (See Section 14.2 in Interlude F for more on monotone functions.)

A third principle is that **logical operations on expressions/conditions correspond to arithmetic operations on the corresponding events**. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be any two Boolean expressions. Then,

- $\mathcal{C}_1 \wedge \mathcal{C}_2$ is their *logical and*, which is true only if both are true;
- $\mathcal{C}_1 \vee \mathcal{C}_2$ is their *logical or*, which is true only if either or both are true;
- $\mathcal{C}_1 \vee\!\!\!\wedge \mathcal{C}_2$ is their *logical exclusive or*, which is true only if one but not both are true.

We can also express the *logical not* (a.k.a. complement) of an expression $\mathcal{C}_1$ – which is true if $\mathcal{C}_1$ is false – by $!\mathcal{C}_1$, but this is only occasionally needed in practice as

we typically just rewrite the expression in complementary form. For instance, the complement of an expression like $X > 7$ is more easily and clearly written $X \leq 7$ than $!X > 7$.

When an expression combines multiple simpler expressions, we can use parentheses for grouping to disambiguate when necessary. We can minimize this with a convention where we assign *precedence* to the logical connectives. Specifically, we assign $\wedge$ (logical and) higher precedence than $\vee$ (logical or), and assign $\vee$ (logical or) higher precedence than $\underline{\vee}$ (logical exclusive-or). So these two conditions are equivalent:

$$(X \geq 0 \wedge |X - Y| < 2) \vee X > 6$$

$$X \geq 0 \wedge |X - Y| < 2 \vee X > 6.$$

We give all three logical connectives lower precedence than other mathematical operators, so parentheses are not needed in most cases.

These logical relations map to relations between the corresponding events

Operation	Relation on Events
Logical And	$\{\mathcal{C}_1 \wedge \mathcal{C}_2\} = \{\mathcal{C}_1\} \cdot \{\mathcal{C}_2\}$
Logical Or	$\{\mathcal{C}_1 \vee \mathcal{C}_2\} = \{\mathcal{C}_1\} + \{\mathcal{C}_2\} - \{\mathcal{C}_1\} \cdot \{\mathcal{C}_2\}$
Logical Xor	$\{\mathcal{C}_1 \underline{\vee} \mathcal{C}_2\} = \{\mathcal{C}_1\} + \{\mathcal{C}_2\} - 2 \cdot \{\mathcal{C}_1\} \cdot \{\mathcal{C}_2\}$
Complement	$\{!\mathcal{C}_1\} = 1 - \{\mathcal{C}_1\}$

For Logical And, the product of events is only 1 if both events occur. For Logical Or (inclusive), the sum of the events is positive if either event occurs, but it “double counts” giving 2 if both events occur, so we correct that with the third term. For logical exclusive-or, we fully disallow the possibility of both events occurring; compare to the inclusive-or on the line above. The Complement row tells us that for every event V there is a complementary event $1 - V$, where V occurs if and only if $1 - V$ does not occur.

Example 23.6. Let $P = X \star Y \star Z$ be an independent join of three scalar random variables, which we think of as a point in space. The event that P lies in a cube of side length 2 centered on the origin, we could use

$$\{|X| \leq 1 \wedge |Y| \leq 1 \wedge |Z| \leq 1\} = \{|X| \leq 1\} \cdot \{|Y| \leq 1\} \cdot \{|Z| \leq 1\}$$

because by independence of the components and equation (19.52)

$$\begin{aligned}\mathbb{E}(\{|X| \leq 1 \wedge |Y| \leq 1 \wedge |Z| \leq 1\}) &= \mathbb{E}(\{|X| \leq 1\} \cdot \{|Y| \leq 1\} \cdot \{|Z| \leq 1\}) \\ &= \mathbb{E}(\{|X| \leq 1\}) \mathbb{E}(\{|Y| \leq 1\}) \mathbb{E}(\{|Z| \leq 1\}),\end{aligned}$$

reducing the calculation to three simpler calculations.

Example 23.7. Let $H_1, H_2, \dots, H_9$ represent independent flips of a balanced coin. The event that we get three heads on the first three flips then a tails on the fourth is

$$H_1 H_2 H_3 (1 - H_4).$$

The event that we get a heads on either or both of the first two flips is $H_1 + H_2 - H_1 H_2$, and the event that we get exactly one flip in the first two is $H_1 + H_2 - 2H_1 H_2$. (Reason through all four cases to see this.) The event that we get exactly one flip in the first four flips is

$$\begin{aligned}H_1(1 - H_2)(1 - H_3)(1 - H_4) &+ (1 - H_1)H_2(1 - H_3)(1 - H_4) \\ &+ (1 - H_1)(1 - H_2)H_3(1 - H_4) + (1 - H_1)(1 - H_2)(1 - H_3)H_4.\end{aligned}$$

Sometimes we have a collection of conditions $\mathcal{C}_i$ indexed by elements i of a set $\mathcal{I}$, and it is usually more convenient to use an index than to list all the terms out explicitly.³ In this case, we write the logical connectives in indexed form, like:

- $\bigwedge_{i \in \mathcal{I}} \mathcal{C}_i$, the condition that all $\mathcal{C}_i$ are true;
- $\bigvee_{i \in \mathcal{I}} \mathcal{C}_i$, the condition that at least one $\mathcal{C}_i$ is true;
- $\underline{\bigvee}_{i \in \mathcal{I}} \mathcal{C}_i$, the condition that *exactly one* $\mathcal{C}_i$ is true.

³For the same reason, it is more convenient to use a `for` loop to sum the elements of an array.

Example 23.8. Define the scalar random variables $X, Y, Z_1, Z_2, Z_3, \dots$, and consider:

$$\begin{aligned}X &= 100, \\ X &> 3 \wedge (Y = 2 \vee Y > 17X), \\ Z_1 + Z_2 &= 0 \wedge Z_2 - 2Z_3 = 0 \wedge Z_1 + Z_2 + Z_3 = 1, \\ \bigvee_{i \in [1..]} \bigwedge_{j \in [2..i)} &Z_j = 0 \wedge Z_i = 1.\end{aligned}$$

The third expression is equivalent to $Z_1 = -2 \wedge Z_2 = 2 \wedge Z_3 = 1$. Notice also that when there are several adjacent connectives in indexed form, the inner index varies fastest: the last example is equivalent to

$$\bigvee_{i \in [1..]} \left(\bigwedge_{j \in [2..i)} Z_j = 0 \wedge Z_i = 1 \right)$$

and because the final term does not depend on j , also equivalent to

$$\bigvee_{i \in [1..]} \left(\bigwedge_{j \in [2..i)} Z_j = 0 \right) \wedge Z_i = 1.$$

Puzzle 168. If $\langle X, Y \rangle$ is a random variable representing a point in the plane, express the event that this point lies in the unit disk. Express the event that the point lies inside both the unit disk *and* the triangle with corners at $\langle 1/2, 0 \rangle$, $\langle 0, 0 \rangle$, and $\langle 0, 2 \rangle$?

Puzzle 169. If $Z_1, Z_2, Z_3, \dots$ is a sequence of events, what is the event that the first Z_i to occur has $i > 7$? What is the event that the first Z_i to occur has i even?

Because conditions give assertions about random variables, their truth or falsehood depends on what happens as the random system evolves. $X > 3$ might be true in one run of the system and false in another run. However, we sometimes need to identify assertions that we *know* to be true, either from structural constraints on the random system or from our baseline knowledge about the world. A condition/expression/event that is *known* to be true before the random system is run is called a **fact**. If we need to identify that $\mathcal{C}$ is a fact and it is not clear from context, we will either call it as such in prose⁴ or write **fact**($\mathcal{C}$). Thus, for example, **fact**($0 \leq Y \leq 100$) means that Y can only have values that are real numbers between 0 and 100. In some simple, common cases, we identify the fact in how we describe the random variable. For instance, rather than write **fact**($N \geq 0$) and **fact**($P > 0$), we might just say that N is a *non-negative random variable* and P is a *positive random variable*.

⁴With phrases like “is a fact,” “must be true”, “is always,” “can only have values.”

Puzzle 170. What’s the difference between **fact**($X = Y$) and $\{X = Y\}$?

Aside. If we have two random variables, X and Y , there are several senses in which we can talk about the “equality” of the random variables.

1. The general condition $X = Y$, which may or may not be true.
2. The fact $\text{fact}(X = Y)$ telling us that whatever the outcome of the random process, these two measurements will have the same value.
3. The definition $Y := X$. This implies $\text{fact}(X = Y)$ but carries with it the framing that Y is derived from X . (This is mostly used to define transformed random variables.)
4. Our predictions about X may be the same as our predictions about Y even if the condition $X = Y$ is always false.

23.3 Constraining with Observations

An observational constraint updates our predictions with the knowledge that a particular condition is true. When we constrain a Kind with an observation, we simply *erase the branches that are inconsistent with the constraint*. The relative weights of the remaining branches *do not change*. The same idea holds in general. Applying this constraint gives a new Distribution and Kernel that retains only the possible values from the original for which the condition is true.

To express how the Distribution and Kernel change under an observational constraint, we will consider a random variable X and a constraint specified by a condition ξ that is compatible with X . This includes constraints like $X \mid X_1 > 0 \wedge X_2 < 0$ or $X \mid X_1^2 + X_2^2 + X_3^2 > 1$, where the $X_i = \text{proj}_i(X)$ are the scalar components of X . If we want to consider a constraint like $X \mid Y > 0$, we first express this in terms of the join $\langle X, Y \rangle \mid Y > 0$ for which the constraint can be expressed by a condition.

With this setup, constraining with an observation is easy: we zero out the weights associated with values that are inconsistent with the constraint and then we renormalize the weights so the total mass equal to 1. This is just erasing branches – that keeps the relative sizes of the remaining weights unchanged – and converting into canonical form. Recall that $\text{total}(f)$ adds up the total mass (discrete and continuous)

for a function f , e.g.,

$$\text{total}(f) = \sum_{x \in \text{range}_d(f)} f(x) + \int_{x \in \text{range}_c(f)} dx f(x). \quad (23.6)$$

Observe also the condition that the **total mass equals 1** can be written for any X as

$$D_X(\text{const}_1) = 1, \quad (23.7)$$

where const_1 is the *constant function* that always returns 1.⁵

⁵See page 562 and Table 10.4 in Interlude F.

If X is a random variable of any dimension and ξ is a compatible condition for which $\xi(X)$ occurs with positive probability (i.e., $D_X(\xi) > 0$), the **Distribution and Kernel of X given ξ** , denoted by $D_{X|\xi}$ and $K_{X|\xi}$, are defined by

$$K_{X|\xi}(x) = \frac{K_X(x) \cdot \xi(x)}{\text{total}(K_X \xi)} \quad (23.8)$$

$$D_{X|\xi}(\psi) = \frac{D_X(\psi \cdot \xi)}{D_X(\xi)}. \quad (23.9)$$

Let's unpack these equations. Let's start with (23.8). The operation of erasing branches corresponds here to setting the weight to zero. The product $K_X(x)\xi(x)$ is equal to $K_X(x)$ when $\xi(x) = 1$ and is equal to 0 otherwise. So we have just erased the branches inconsistent with our condition. The denominator is just the total mass of the remaining branches; we divide by this to ensure that the total mass of the resulting Kernel becomes 1. This equation is just a formal version of what we have done for Kind trees from the beginning when constraining with an observation. Equation (23.9) has the same meaning. We transform the statistic ψ to the statistic $\psi \cdot \xi$, so the value $\psi(x)$ becomes $\psi(x)\xi(x)$. That is, we are averaging values of the statistic for all values consistent with the constraint; the remaining values are set to 0. The denominator renormalizes this average, adding up the mass only for those values that are consistent with the constraint. The denominator is the *probability* that the condition is satisfied. The average in the numerator is closer to 0 by a factor of precisely this probability.

By analogy, if we have a list of numbers 3, 7, 10, -2, -8, 1, 5, 4, 24, -2. It's average is

$$\frac{3 + 7 + 10 + -2 + -8 + 1 + 5 + 4 + 24 + -2}{10} = \frac{32}{10}$$

Now, imagine that we want to remove 3, 7, 10, 1, 5, 4 from the list and compute the average. We can do that by setting the elements inconsistent with our constraint to 0

$$\frac{0 + 0 + 0 + -2 + -8 + 0 + 0 + 0 + 24 + -2}{10} = \frac{12}{10}.$$

Oops! This is not the average of the remaining items in the list. We need to normalize by the proportion of items remaining

$$\frac{\frac{0+0+0+-2+-8+0+0+0+24+-2}{12}}{\frac{0+0+0+1+1+0+0+0+1+1}{10}} = \frac{\frac{12}{10}}{\frac{4}{10}} = \frac{12}{4} = 3.$$

By zeroing out the inconsistent terms, we bring our average too close to zero, so we adjust by scaling the average by the proportion of terms remaining. This is the pattern in both equations (23.8) and (23.9).

Notice that $K_{X|\xi}(x) > 0$ only if $\xi(x) = 1$, so the possible values of $X | \xi$ are the possible values of X that are consistent with the constraint. Similarly, for any statistic φ , if $\varphi \cdot \xi$ is always zero, then no $D_{X|\xi}(\varphi) = 0$.

Example 23.9. Let A and B be random variables with Kernels $K_A(j) = \frac{1}{3} \{j \in [0 \dots 2]\}$ and $K_B(k) = \frac{1}{5} \{k \in [-2 \dots 2]\}$. Let $Z = A \star B$, the independent join of A and B . Then

$$K_Z(j, k) = \frac{1}{15} \{j \in [0 \dots 2]\} \{k \in [-2 \dots 2]\}.$$

Now consider the condition $A + B = 0$. This Boolean expression represents the condition $\xi(j, k) = \{j + k = 0\}$. The Kernel $K_{Z|\xi}$ or equivalently $K_{Z|A+B=0}$ is given by

$$\begin{aligned} K_{Z|A+B=0}(j, k) &= \frac{\frac{1}{15} \{j \in [0 \dots 2]\} \{k \in [-2 \dots 2]\} \{j + k = 0\}}{\sum_{i, \ell} \frac{1}{15} \{i \in [0 \dots 2]\} \{\ell \in [-2 \dots 2]\} \{i + \ell = 0\}} \\ &= \frac{\{j \in [0 \dots 2]\} \{k \in [-2 \dots 2]\} \{j + k = 0\}}{\sum_{i, \ell} \{i \in [0 \dots 2]\} \{\ell \in [-2 \dots 2]\} \{i + \ell = 0\}} \\ &= \frac{1}{3} \{j \in [0 \dots 2]\} \{k \in [-2 \dots 2]\} \{j + k = 0\}. \end{aligned}$$

Don't let the messiness of the first expression intimidate you. The product of indicators in the numerator just selects out values from the original independent join in which the components sum to 0. The denominator is just the total mass of

the updated Kernel. Notice that the common factor of $\frac{1}{15}$ cancels. The resulting sum adds 1 for each of $\langle 0, 0 \rangle, \langle 1, -1 \rangle, \langle 2, -2 \rangle$.

Example 23.10. Let $Z = \langle X, Y \rangle$ be a random point chosen uniformly in the unit square. Specifically, $Z = X \star Y$ where $K_X = K_Y = [0\_1]$. Let ξ be the condition $\xi(u, v) = \{u + v > 1\}$, with corresponding Boolean expression $X + Y > 1$.

We will compute the updated Distribution of Z . $D_{Z|\xi} \equiv D_{Z|X+Y>1}$ is given by

$$\begin{aligned} D_{Z|X+Y>1}(\psi) &= \frac{D_Z(\psi \cdot \xi)}{D_Z(\xi)} \\ &= \frac{\int_s ds \int_t dt K_Z(s, t) \psi(s, t) \xi(s, t)}{\int_s ds \int_t dt K_Z(s, t) \xi(s, t)} \\ &= \frac{\int_{s=0}^1 ds \int_{t=0}^1 dt \psi(s, t) \xi(s, t)}{\int_{s=0}^1 ds \int_{t=0}^1 dt \xi(s, t)} \\ &= \frac{\int_{s=0}^1 ds \int_{t=1-s}^1 dt \psi(s, t)}{\int_{s=0}^1 ds \int_{t=1-s}^1 dt} \\ &= \int_s ds \int_t dt 2 \{0 \leq s \leq 1\} \{1-s \leq t \leq 1\} \psi(s, t), \end{aligned}$$

because ξ is non-zero on the triangle where $1-s \leq t \leq 1$ for $s \in [0\_1]$. We convert the indicators to limits of integration and vice versa as needed. This also gives us the Kernel:

$$K_{Z|X+Y>1}(s, t) = 2 \{0 \leq s \leq 1\} \{1-s \leq t \leq 1\}.$$

You may wonder what happens in the last example if we instead used the condition $\zeta(x, y) = \{x + y = 1\}$. The problem here is that $D_Z(\zeta) = 0$! The probability mass on the line where $\zeta(x, y) = 1$ is infinitesimal, which disappears when we compute the probability. As we have seen, this oddity is one challenge of allowing continuous values in our theory.

If $D_Z(\zeta) = 0$, our equations (23.8) and (23.9) no longer technically apply because the denominator is zero. The good news is that this problem does have a well-defined solution, and the answer is (in this case) pretty much what you would expect if you naively applied equation (23.9). We return to – and solve – this problem in Section 24.3 with the Updating equation of Section 24.3 and the transformation techniques in Chapter 25. The short story is that we *transform* Z by the statistic $\psi(x, y) = \langle x + y, x - y \rangle$ and then apply the Updating equation to find our answer.

23.4 The Expectation Updating Equation

Equation (23.9) is a useful tool in practice. To see this, we can write the equation out directly with $\psi = \text{id}$ and a condition represented by a Boolean expression $\mathcal{C}$. Assume that $\mathbb{E}(\{\mathcal{C}\}) > 0$, then

$$\mathbb{E}(Y \mid \mathcal{C}) = \frac{\mathbb{E}(Y \{\mathcal{C}\})}{\mathbb{E}(\{\mathcal{C}\})}. \quad (23.10)$$

This tells us how to update the expectation of Y for new information represented by the observation that $\{\mathcal{C}\}$ has occurred. We call this the **Expectation Updating equation**. In the numerator, we have a product of the two random variables Y and $\{\mathcal{C}\}$; this product is zero *unless* the event $\{\mathcal{C}\}$ occurs. The denominator is the probability that the event $\{\mathcal{C}\}$ occurs.

Think of the right-hand side of (23.10) as a ratio of two averages. The numerator averages the values of Y but zeroing out those that are inconsistent with the condition. This is like the average of only those values consistent with $\mathcal{C}$ except the extra zeros, which bring the average closer to 0. How much closer? By precisely the factor in the denominator, the probability that the value of Y is consistent with $\mathcal{C}$.⁶

⁶See the analogy discussed on page 954.

Example 23.11 How many more flips?

Let N be the number of flips of a coin until the first heads appears. Assume that each flip comes up heads with probability p and that all flips are independent. Let $H_1, H_2, \dots$ be the events that we get a heads on successive flips. If we know that that $N > k$, what is the expected number of extra flips we have to wait

$K_N(k) = \mathbb{E}(\{N = k\})$, the probability that the first heads appears on the k th flip. Use the event reduction method to reexpress the event $\{N = k\}$. The first head appears on the k th flip if and only if the first $k - 1$ flips are tails and the k th is heads. That is,

$$\begin{aligned} \{N = k\} &= \{H_1 = 0 \wedge \dots \wedge H_{k-1} = 0 \wedge H_k = 1\} \\ &= \{H_1 = 0\} \cdots \{H_{k-1} = 0\} \cdot \{H_k = 1\}. \end{aligned}$$

So, because we have an independent join

$$\begin{aligned} K_N(k) &= \mathbb{E}(\{N = k\}) \\ &= \mathbb{E}(\{H_1 = 0\} \cdots \{H_{k-1} = 0\} \cdot \{H_k = 1\}) \\ &= \mathbb{E}(\{H_1 = 0\}) \cdots \mathbb{E}(\{H_{k-1} = 0\}) \cdot \mathbb{E}(\{H_k = 1\}) \\ &= (1 - p) \cdots (1 - p)p \{k \in [1 \dots]\} \end{aligned}$$

$$= (1 - p)^{k-1} p \{k \in [1 \dots]\},$$

using equation (19.52).

We want to find $\mathbb{E}(N - k \mid N > k)$. The Additivity and Constancy Rules yield

$$\mathbb{E}(N - k \mid N > k) = \mathbb{E}(N \mid N > k) - \mathbb{E}(k \mid N > k) = \mathbb{E}(N \mid N > k) - k.$$

Then, by the Expectation Updating and Mass-balancing equations

$$\begin{aligned} \mathbb{E}(N \mid N > k) &= \frac{\mathbb{E}(N \{N > k\})}{\mathbb{E}(\{N > k\})} \\ &= \frac{\sum_j \mathbf{K}_N(j) j \{j > k\}}{\sum_j \mathbf{K}_N(j) \{j > k\}} \\ &= \frac{\sum_j (1 - p)^{j-1} p \{j \in [1 \dots]\} j \{j > k\}}{\sum_j (1 - p)^{j-1} p \{j \in [1 \dots]\} \{j > k\}} \\ &= \frac{p \sum_{j=k+1}^{\infty} (1 - p)^{j-1} j}{p \sum_{j=k+1}^{\infty} (1 - p)^{j-1}} \\ &= \frac{\sum_{i=1}^{\infty} (1 - p)^{k+i-1} (i + k)}{\sum_{i=1}^{\infty} (1 - p)^{k+i-1}} \\ &= \frac{\sum_{i=1}^{\infty} (1 - p)^{i-1} (i + k)}{\sum_{i=1}^{\infty} (1 - p)^{i-1}} \\ &= \sum_{i=1}^{\infty} p (1 - p)^{i-1} (i + k) \\ &= \mathbb{E}(N) + k, \end{aligned}$$

and thus

$$\mathbb{E}(N - k \mid N > k) = \mathbb{E}(N).$$

Our expected additional waiting time is the same as our expected waiting time at the beginning. The system has “forgotten” that we’ve already waited.

Example 23.12 Conditional Probabilities

If V and W are events, then the Expectation Updating equation tells us that

$$\mathbb{E}(V \mid W = 1) = \frac{\mathbb{E}(VW)}{\mathbb{E}(W)}.$$

The probability that V occurs *given that* W occurs is the ratio of two probabilities: the probability that W and V both occur and the probability that W occurs.

As a concrete example, suppose that the 2-dimensional random variable $\langle A, B \rangle$ has kernel

$$K_{AB}(a, b) = \sum_{i=-1}^1 \sum_{j=0}^{|i|} \frac{1}{5} \{a = i \wedge b = j\}.$$

If we observe that $B = 0$, what is the probability that $A = 1$?

$$\begin{aligned} \mathbb{E}(A = 1 \mid B = 0) &= \frac{\mathbb{E}(\{A = 1\} \{B = 0\})}{\mathbb{E}(\{B = 0\})} \\ &= \frac{\mathbb{E}(\{A = 1 \wedge B = 0\})}{\mathbb{E}(\{B = 0\})} \\ &= \frac{K_{AB}(1, 0)}{K_{AB}(-1, 0) + K_{AB}(0, 0) + K_{AB}(1, 0)} \\ &= \frac{1/5}{1/5 + 1/5 + 1/5} = \frac{1}{3}. \end{aligned}$$

The Expectation Updating equation is a special case of the more general Updating equation that we will discuss in Section 24.3.

23.5 Conditional Distributions, Conditional Expectations

In Chapter 4, we saw that every Kind and FRP can be considered a conditional Kind and conditional FRP that simply takes a 0-dimensional value as input, i.e., requires no input information to specify. Analogously, in this section, we have expressed Distributions, Kernels, and expectations conditionally on an arbitrary constraint $\mathcal{C}$. But if $\mathcal{C} = \top$, the condition that is always true, i.e., no constraint, we can simply drop all the $\mid \mathcal{C}$'s. In that sense, all Distributions are conditional Distributions, all Kernels are conditional Kernels, and all expectations are conditional expectations.

The special case alluded to earlier is common enough to bear repeating. When the expression $\mathcal{C}$ has the form $X = u$ for some random variable X and some possible value u , we write

$$\begin{aligned} D_{Y \mid \mathcal{C}}(\psi) &= D_{Y \mid X}(\psi \mid u) \\ K_{Y \mid \mathcal{C}}(w) &= K_{Y \mid X}(w \mid u). \end{aligned}$$

It is both more elegant and more convenient in practice to move the value up behind

the given bar and to write these in terms of the conditional Distribution and Kernel of Y given X .

The Distribution operator maps questions (in the form of statistics) to predicted answers (in the form of expectations). We can do this for any Kind whatsoever, with or without conditional constraints. The *conditional Distribution* given a constraint maps questions to predicted answers under the knowledge that the constraint is satisfied.

If X and Y are random variables of arbitrary (and non necessarily equal) dimension and ψ is any statistic compatible with Y , then the **conditional distribution of Y given X** is defined by the family of Distribution operators

$$D_{Y|X}(\psi | x) \quad (23.11)$$

$$= \mathbb{E}(\psi(Y) | X = x) \quad (23.12)$$

$$= \sum_v K_{Y|X}(v | u) \text{range}_d(Y)(v) \psi(v) + \int_v dv K_{Y|X}(v | u) \text{range}_c(Y)(v) \psi(v) \quad (23.13)$$

$$= \sum_v p_{Y|X}(v | u) \psi(v) + \int_v dv f_{Y|X}(v | u) \psi(v), \quad (23.14)$$

for all *possible* values x of X .

The first equality shows that conditional Distributions are Distribution operators just like any other. The subscript label simply lets us keep track of which random variables a Distribution is associated with and what knowledge is assumed known. The second and third equalities are just the mass-balancing equations.

The conditional expectation is defined in terms of the conditional Kind's Kernel, as shown in the previous section. For example, in the special cases where Y is either discrete or continuous, respectively, we get

$$\mathbb{E}(\psi(Y) | X = x) = \sum_y K_{Y|X}(y | x) \psi(y) \quad (\text{discrete})$$

$$\mathbb{E}(\psi(Y) | X = x) = \int_y dy K_{Y|X}(y | x) \psi(y) \quad (\text{continuous})$$

The operations are the same in both cases, only the way we calculate the “sum” changes. Notice again the three pieces in these expressions, in order: summation operation, kernel, statistic.

Example 23.13. Consider a two-dimensional random variable $\langle J, K \rangle$ where J and K are each *events*. The kernel K_{JK} is positive for only the four values $\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle$, and $\langle 1, 1 \rangle$. We can present this as a table

$$K_{JK}(j, k) \quad \begin{array}{cc} k = 0 & k = 1 \\ \begin{array}{c} j = 0 \\ j = 1 \end{array} & \begin{array}{cc} p_{00} & p_{01} \\ p_{10} & p_{11} \end{array} \end{array}$$

where $p_{00} + p_{01} + p_{10} + p_{11} = 1$. We want to find $D_{K|J}(\psi | j) = \mathbb{E}(\psi(K) | J = j)$, for any statistic ψ that accepts 0 and 1 as inputs.

First, notice that because K is an event

$$\psi(K) = \psi(0)(1 - K) + \psi(1)K. \quad (23.15)$$

This is $\psi(1)$ when $K = 1$ and $\psi(0)$ when $K = 0$. It follows that

$$\begin{aligned} \mathbb{E}(\psi(K) | J = j) &= \mathbb{E}(\psi(0)(1 - K) + \psi(1)K | J = j) \\ &= \psi(0)(1 - \mathbb{E}(K | J = j)) + \psi(1) \mathbb{E}(K | J = j) \\ &= \psi(0) - \psi(0) \mathbb{E}(K | J = j) + \psi(1) \mathbb{E}(K | J = j) \\ &= \psi(0) + (\psi(1) - \psi(0)) \mathbb{E}(K | J = j). \end{aligned} \quad (23.16)$$

So all we need to find is $\mathbb{E}(K | J = j)$.

Using the Updating Equation and equation (23.10), we have

$$\begin{aligned} \mathbb{E}(K | J = j) &= \frac{\mathbb{E}(K \{J = j\})}{\mathbb{E}(\{J = j\})} \\ &= \frac{\sum_{m=0}^1 \sum_{n=0}^1 n \{j\}(m) K_{JK}(m, n)}{K_{JK}(0, j) + K_{JK}(1, j)} \\ &= \frac{\sum_{n=0}^1 n K_{JK}(j, n)}{K_{JK}(0, j) + K_{JK}(1, j)} \\ &= \frac{K_{JK}(j, 1)}{K_{JK}(0, j) + K_{JK}(1, j)} \\ &= \frac{p_{j1}}{p_{j0} + p_{j1}}. \end{aligned}$$

Note that this is only well-defined when j is a possible value, i.e., $j \in \{0, 1\}$.

Example 23.14 Nearest Prime Let Z be a continuous random variable with Distribution

$$D_Z(\psi) = \int_{t=2}^{11} dt \frac{1}{9} \psi(t).$$

and let Q be the closest prime number to Z , breaking ties toward the larger prime. **Find the conditional distribution of Z given Q .**

Then, $Q = \psi(Z)$. What is the statistic ψ ?

$$\begin{aligned} \psi(z) &= 2 \cdot [2\_2.5](z) + 3 \cdot [2.5\_4](z) + 5 \cdot [4\_6](z) + 7 \cdot [6\_9](z) + 11 \cdot [9\_11](z) \\ &= 2 \cdot \mathcal{A}_2(z) + 3 \cdot \mathcal{A}_3(z) + 5 \cdot \mathcal{A}_5(z) + 7 \cdot \mathcal{A}_7(z) + 11 \cdot \mathcal{A}_{11}(z), \end{aligned}$$

defining the sets $\mathcal{A}_k$ term by term. (For instance, $\mathcal{A}_2 = [2\_2.5]$.)

To apply the Updating Equation, we need K_{ZQ} and K_Q . The first we can find with the join equation;

$$K_{ZQ}(z, q) = K_{Q|Z}(q | z) K_Z(z) = \{\psi(z)\}(q) \frac{1}{9} [2\_11](z) = \frac{1}{9} \mathcal{A}_q(z).$$

The second we can find by projecting on the first component:

$$\begin{aligned} K_Q(q) &= \int_z dz K_{ZQ}(z, q) \\ &= \int_z dz \frac{1}{9} \mathcal{A}_q(z) \\ &= \frac{\text{length}(\mathcal{A}_q)}{9} \\ &= \frac{1}{18} \{q = 2\} + \frac{3}{18} \{q = 3\} + \frac{2}{9} \{q = 5\} + \frac{1}{3} \{q = 7\} + \frac{2}{9} \{q = 11\}. \end{aligned}$$

Now, we apply the Updating equation

$$K_{Z|Q}(z | q) = \frac{K_{ZQ}(z, q)}{K_Q(q)} = \frac{\frac{1}{9} \mathcal{A}_q(z)}{\frac{\text{length}(\mathcal{A}_q)}{9}} = \frac{1}{\text{length}(\mathcal{A}_q)} \mathcal{A}_q(z).$$

Hence,

$$D_{Z|Q}(\zeta | q) = \int_z dz \frac{1}{\text{length}(\mathcal{A}_q)} \mathcal{A}_q(z) \zeta(z).$$

That is, given that $Q = q$, Z has a uniform distribution over the set $\mathcal{A}_q$.

Building with Joins

24

Chapter

Contents

24.1 Independence	966
24.2 Conditional Independence	982
24.3 The Updating Equation	986
24.4 The Multiplication Rule	990
24.5 The Mighty Conditioning Identity	999

Key Take Aways

Two random variables are *independent* if observing the value of one does not change our predictions about the other.

We can express this property in two equivalent but distinct ways. Let X and Y be two random variables of arbitrary (and possibly different) dimension. X and Y are independent if and only if any of the following holds

1. $D_{Y|X}(\psi | x) = D_Y(\psi)$
2. $K_{Y|X}(y | x) = K_Y(y)$
3. $D_{X|Y}(\varphi | y) = D_X(\varphi)$
4. $K_{X|Y}(x | y) = K_X(x)$
5. $D_{XY}(\varphi \otimes \psi) = D_X(\varphi) D_Y(\psi)$
6. $K_{XY} = K_X \otimes K_Y$, that is $K_{XY}(x, y) = K_X(x) K_Y(y)$.

We can extend these properties from two to a collection of three or more random variables, $Z_1, Z_2, \dots, Z_n$, and in that case, we say that $Z_1, Z_2, \dots, Z_n$

are *mutually* independent. And there are several other ways to express these conditions.

Items 1–4 captures the idea that observing the value of one variable does not change our predictions about the other. Items 5–6 capture the idea that independent variables can be analyzed separately.

The notion of independence carries over to the situation where we know the values of one or more other random variables. We call this **conditional independence**. We say that X and Y are **conditionally independent given a random variable Z** if and only if any of the following hold:

1. $D_{Y|XZ}(\psi \mid x, z) = D_{Y|Z}(\psi \mid z)$
2. $K_{Y|XZ}(y \mid x, z) = K_{Y|Z}(y \mid z)$
3. $D_{X|YZ}(\varphi \mid y, z) = D_{X|Z}(\varphi \mid z)$
4. $K_{X|YZ}(x \mid y, z) = K_{X|Z}(x \mid z)$
5. $D_{XY|Z}(\varphi \otimes \psi \mid z) = D_{X|Z}(\varphi) D_{Y|Z}(\psi \mid z)$ Recall: $(\varphi \otimes \psi)(x :: y) = \varphi(x) \cdot \psi(y)$
6. $K_{XY|Z}(x, y \mid z) = K_{X|Z}(x \mid z) K_{Y|Z}(y \mid z)$

Notice that these conditions exactly parallel those of independence with the inclusion of Z . If Z is a constant random variable – which by definition is known – we recover the original properties.

The **Updating equation** turns around the join equation to find a conditional Kernel:

$$K_{Y|X}(y \mid x) = \frac{K_{XY}(x, y)}{K_X(x)}. \quad (24.1)$$

The Expectation Updating equation we saw earlier is a special case.

The **Multiplication Rule** comes in two forms. The event form follows from

the Expectation Updating equation. If $A_1, A_2, \dots, A_n$ are *events*:

$$\begin{aligned}
 \mathbb{E}(A_1 A_2 \cdots A_n) &= \cdot \mathbb{E}(A_1) \\
 &\quad \cdot \mathbb{E}(A_2 \mid A_1 = 1) \\
 &\quad \cdot \mathbb{E}(A_3 \mid A_2 = 1 \wedge A_1 = 1) \\
 &\quad \cdots \\
 &\quad \cdot \mathbb{E}(A_{n-1} \mid A_{n-2} = 1 \wedge \cdots \wedge A_1 = 1) \\
 &\quad \cdot \mathbb{E}(A_n \mid A_{n-1} = 1 \wedge \cdots \wedge A_1 = 1). \tag{24.2}
 \end{aligned}$$

The Kernel form is a direct consequence of the Updating equation:

$$\begin{aligned}
 &K_{x_1 x_2 \cdots x_n}(x_1, x_2, \dots, x_n) \\
 &= K_{x_1}(x_1) K_{x_2|x_1}(x_2 \mid x_1) K_{x_3|x_1, x_2}(x_3 \mid x_1, x_2) \cdots K_{x_n|x_1, x_2, \dots, x_{n-1}}(x_n \mid x_1, x_2, \dots, x_{n-1}).
 \end{aligned}$$

The **Mighty Conditioning Identity** is a powerful tool for finding Distributions. When we have

- Y and X are arbitrary random variables (of any dimension).
- $\mathcal{C}$ is an arbitrary observational constraint; if $\mathcal{C} = \top$, we can drop it.
- φ is a statistic compatible with Y that embodies our question.

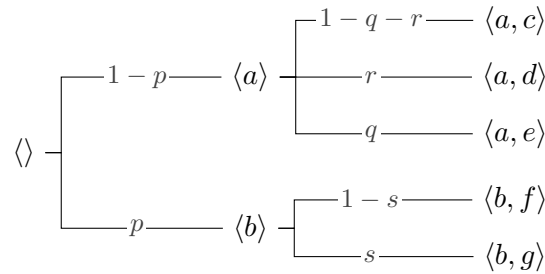
Then,

$$D_{Y|\mathcal{C}}(\varphi) = D_{X|\mathcal{C}}(\langle x \rangle \mapsto D_{Y|X \wedge \mathcal{C}}(\varphi \mid x))$$

This is an expression of the conditioning operator $\llbracket$ (`//` in `frplib`) that we saw earlier. We discuss various ways to express this identity and many examples.

We use the join operation to combine information produced at one stage in the evolution of a random system with the randomness produced at the next stage. Joins introduce new randomness into the system that is contingent on what came before. Of course, we can use the join operation without a notion of *time*, it gives us a way to talk about the behavior of multiple random variables jointly in terms of the contingent behavior of some given the values of the others.

Figures 4.1 and 4.3 in Chapter 4 illustrate the join process. A random value produced at one stage leads to a random value at the next stage that is contingent on the first value. In terms of the Kinds, we get trees that look like



To find the weight on a value, we follow a path from root to that value's leaf, multiplying (canonical) weights as we go. For instance, the weight on $\langle a, d \rangle$ is $(1-p)r$; we take the a branch at the first stage and then the d branch in the a -subtree. Similarly, the weight on $\langle b, f \rangle$ is $p(1-s)$; we take the b branch at the first stage and then the f branch of the b -subtree.

The join equation (19.45) is just an expression of this process:

$$K_{XY}(x, y) = K_X(x) K_{Y|X}(y | x). \quad (24.3)$$

To get the weight on value $x :: y$, we multiply the weight on the x branch of X by the weight on the y branch of Y in the x -subtree. This works for random variables of any dimension and of any type (discrete, continuous, simple, split, or otherwise).

The special case of an independent join occurs when the subtrees at every value of X are *the same*. So the value of Y is not influenced by the value of X (and vice versa). In this case, $K_{Y|X}(y | x)$ is just $K_Y(y)$, which gives

$$K_{XY}(x, y) = K_X(x) K_Y(y). \quad (24.4)$$

Our predictions about Y do not depend on the value of X . This is the concept behind *independence*.

In this section, we develop some of the implications of these join equations and several powerful techniques that derive from them.

24.1 Independence

As we have just seen, an *independent join* combines Kinds so that what happens at each stage does not influence what happens at any other stage. There is no contingency. When we form an independent join of Kinds, like $K_X \star K_Y$, we are putting strong constraints on K_{XY} . We say in this case that X and Y are *independent*.

Independence is a powerful assumption that allows us to decompose a system

into parts that we can consider separately. Independence makes a system easier to describe, understand, and analyze, so it is a commonly-made assumption in models for random systems. It can be a reasonable assumption in many cases, though its validity should be carefully considered.

We have two useful perspectives on what independence means. Ultimately, both are equivalent, but they lead to different formulations.

First, **two random variables are *independent* if observing the value of one does not change our predictions about the other, and vice versa**. We see this in the Kind tree of an independent join: the Kind of the value produced at the second stage is the same whatever the value of the first stage. There is no information flowing between independent parts of the system, so whatever happens with one has no influence on what happens with the other.

The second perspective is that **two random variables are independent if they can be analyzed separately**. Independent random variables represent decoupled parts of the system, and if our questions about the system decouple along the same lines, it becomes easier to predict answers to those questions. Independent parts may remain decoupled throughout, or we may combine them into new random variables that couple them. Either way, the independence of the variables gives us a powerful tool for simplifying things.

These two formulations are reflected in the definition, with two types of conditions that establish independence. Recall from Chapter 15 in Interlude F that $(\varphi \otimes \psi)(x :: y) = \varphi(x) \cdot \psi(y)$.

Let X and Y be two random variables of dimensions m and n , and let φ and ψ be statistics compatible respectively with X and Y .

Then, X and Y are **independent** if and only if any of the following holds:

$$1. \ D_{Y|X}(\psi \mid x) = D_Y(\psi) \quad (24.5)$$

$$2. \ K_{Y|X}(y \mid x) = K_Y(y) \quad (24.6)$$

$$3. \ D_{X|Y}(\varphi \mid y) = D_X(\varphi) \quad (24.7)$$

$$4. \ K_{X|Y}(x \mid y) = K_X(x) \quad (24.8)$$

$$5. \ D_{XY}(\varphi \otimes \psi) = D_X(\varphi) D_Y(\psi) \quad (24.9)$$

$$6. \ K_{XY} = K_X \otimes K_Y, \text{ that is } K_{XY}(x, y) = K_X(x) K_Y(y). \quad (24.10)$$

Items 1–4 reflects the first perspective, the idea that observing the value of one variable does not change our predictions about the other. Observing the value of X does not change what we know about Y , and vice versa. The condition of independence is *symmetric*.

Items 5–6 reflects the second, the idea that independent random variables can be analyzed separately. Notice here that statistics of the form $\psi \star \varphi$ are statistics that treat X and Y *separately*. (That is, $(\psi \star \varphi)(x, y) = \psi(x)\varphi(y)$, where x has dimension m and y has dimension n .) Thus, item 5 says that for any question that can be decomposed into separate sub-questions – one on X and one on Y – we can separately answer the two sub-questions and then combine the answers to answer the original question.

Example 24.1. Let Z have a $\text{DiscreteUniform}([1..4]^2)$ distribution, and let $\langle X, Y \rangle$ be its scalar components. We will show that X and Y are independent. Before doing this formally with the above equations, we can see this by reasoning from what we already know.

First, consider the Distribution of X constrained with the observation that $Y = k$ for some $k \in [1..4]$. Constraining with an observation prunes all the branches that are inconsistent with the constraint leaving the relative probabilities of the remaining options unchanged. Because Z has a Uniform distribution, all possibilities are equally likely before constraining with an observation, so $Z \mid Y = k$ is Uniform as well. The only remaining points under this constraint are those at $\langle 1, k \rangle$, $\langle 2, k \rangle$, $\langle 3, k \rangle$, and $\langle 4, k \rangle$, and so $X \mid Y = k$ must have a $\text{Uniform}([1..4])$ Distribution. But this is also the Distribution of X , with no observational constraint.

Second, let us see this formally using equation (3).

From what we know about Uniform distributions, we can see that this is an independent join, so if we compute D_X and D_Y , we would find that they are both $\text{DiscreteUniform}([1..4])$ distributions, and $Z = X \star Y$.

Consider $D_{X|Y}(\psi \mid k)$ for $k \in [1..4]$. Applying the Expectation Updating Equation, we have

$$\begin{aligned} D_{X|Y}(\psi \mid k) &= \mathbb{E}(\psi(X) \mid Y = k) \\ &= \frac{\mathbb{E}(\psi(X)\{Y = k\})}{\mathbb{E}(\{Y = k\})} \end{aligned}$$

$$\begin{aligned}
&= \frac{\sum_{i=1}^4 \sum_{j=1}^4 \frac{1}{16} \psi(i) \{j = k\}}{1/4} \\
&= \sum_{i=1}^4 \frac{1}{4} \psi(i) \sum_{j=1}^4 \{j = k\} \\
&= \sum_{i=1}^4 \frac{1}{4} \psi(i) \\
&= \sum_i \frac{1}{4} \{i \in [1 \dots 4]\} \psi(i) = D_X(\psi),
\end{aligned}$$

which is again a $\text{DiscreteUniform}\langle[1 \dots 4]\rangle$ distribution. So, $D_{X|Y}(\blacksquare | k) = D_X$ for every possible $k \in [1 \dots 4]$.

Knowing the value of Y does not change our predictions about X , and by similar calculation, vice versa. So, X and Y are independent random variables with a $\text{DiscreteUniform}\langle[1 \dots 4]\rangle$ distributions, and $Z = X \star Y$ is an independent join.

Example 24.2. Example 21.9 gives us further insight into the nature of independence. Again, we have two scalar components X and Y whose join $\langle X, Y \rangle$ is Uniform over a two-dimensional set of points.

In that example, we computed both $D_{X|Y=k}$ and D_X and found that they are not the same. It follows that X and Y are not independent.

Why? The distinction between this and the previous example is the *shape* of the set of possible values. When we observe Y , the observation restricts X 's values to the horizontal slice at that Y -value. But the size of that slice – and thus the conditional Distribution of X – *changes* with the value of Y . For instance, if you know that $Y = 5$, then X must be 0, the only valid remaining value. But if you know that $Y = 0$, then X must be Uniform on $[-5 \dots 5]$. This is the essence of dependence.

Example 24.3. Let $\varphi(x) = x$ and $\psi(y) = y$, then $(\varphi \otimes \psi)(x, y) = xy$. If X and Y are independent, we can conclude by equation (5) that

$$\mathbb{E}(XY) = \mathbb{E}(X)\mathbb{E}(Y).$$

(The converse is not true.)

Let's confirm this in Example 24.1:

$$\begin{aligned}
 \mathbb{E}(XY) &= \sum_{i=1}^4 \sum_{j=1}^4 \mathbf{K}_{XY}(i, j) (\varphi \otimes \psi)(i, j) \\
 &= \sum_{i=1}^4 \sum_{j=1}^4 \frac{1}{16} \varphi(i) \psi(j) \\
 &= \sum_{i=1}^4 \sum_{j=1}^4 \frac{1}{16} ij \\
 &= \left(\frac{1}{4} \sum_{i=1}^4 i \right) \left(\frac{1}{4} \sum_{j=1}^4 ij \right) \\
 &= \mathbb{E}(X) \mathbb{E}(Y).
 \end{aligned}$$

Check! This works because we showed independence in the earlier example. This property corresponds to our second interpretation of independence. The random variable XY is a measurement from the system that combines measurements X and Y from independent parts of the system. Independence tells us that the combination XY can be analyzed by analyzing the two parts separately.

It is important to restate that merely showing $\mathbb{E}(XY) = \mathbb{E}(X) \mathbb{E}(Y)$ is *not* sufficient to show independence.

Example 24.4. In Example 24.1, we showed that X and Y are independent using equation (1), so here we do so using equation (5) instead.

We need to show that $\mathbf{D}_{XY}(\varphi \otimes \psi) = \mathbf{D}_X(\varphi) \mathbf{D}_Y(\psi)$ for all compatible statistics φ and ψ . Here $\varphi \otimes \psi$ is an arbitrary statistic compatible with $\langle X, Y \rangle$ *except* that it acts separately on the X piece, by φ , and the Y piece, by ψ . Specifically, $(\varphi \otimes \psi)(X, Y) = \varphi(X) \psi(Y)$.

We can calculate this directly.

$$\begin{aligned}
 \mathbf{D}_Z(\varphi \otimes \psi) &= \mathbb{E}(\varphi(X) \psi(Y)) \\
 &= \sum_{i=1}^4 \sum_{j=1}^4 \frac{1}{16} \varphi(i) \psi(j) \\
 &= \frac{1}{16} \sum_{i=1}^4 \varphi(i) \sum_{j=1}^4 \psi(j)
 \end{aligned}$$

$$\begin{aligned}
&= \left(\frac{1}{4} \sum_{i=1}^4 \varphi(i) \right) \left(\frac{1}{4} \sum_{j=1}^4 \psi(j) \right) \\
&= D_X(\varphi) \cdot D_Y(\psi).
\end{aligned}$$

Again the shape of $\text{range}(Z)$ and the uniform weights makes it possible to decouple that double sum into a product of sums.

We can capture properties 1–4 in the definition of independence in an equivalent but slightly more general way. Let $\text{info}(Y)$ be the set of all conditions compatible with Y . These are all the conditions whose truth or falsehood can be determined once we observe the value of Y .

Proposition. Two random variables X and Y are independent if

$$D_{X|\mathcal{C}} = D_X \text{ for all conditions } \mathcal{C} \in \text{info}(Y).$$

If this equation is true, it is also true with X and Y interchanged.

Remember that the distribution of X captures all of our knowledge about X in isolation, all the predictions that we can make about its value. When information about Y does not change that knowledge, X and Y are independent.

When X and Y are independent random variables, the independent join $Z = X \star Y$ builds a random system by combining two simpler sub-systems in a way that reflects the lack of contingency between the sub-systems. We saw this earlier in the way Kind trees are combined by the $\star$ operation. The Y -subtree looks the same for every possible value of X . In this sense, we can say that Z *factored* into two parts represented by X and Y . Equations (5) and (6) capture this idea of factoring.

In equation (5), we consider questions – as embodied by statistics of the form $\varphi \otimes \psi$ – that reflect separate questions about the two parts of the system whose answers are then combined. Our predicted answers to those questions come from combining *separate* predictions about the two component questions.

In equation (6), we see that the Kernel itself factors. The weight on a branch $x :: y$ is the product of the weights on the path from root to leaf: the weight on the x -branch for X times the weight on the y -branch of Y . This factoring of the Kernels is just equations (19.43) and (19.44) again.

So, the idea of factoring – analyzing independent sub-systems separately – applies to the Distribution operator and to the Kernel, and it extends to many other representations of a distribution.

Factoring Distributions. If X and Y are independent random variables with independent join $Z = X \star Y$, then various representations of Z 's distribution *factors* into terms related to X and Y :

$$K_Z = K_X \otimes K_Y \quad (24.11)$$

$$p_Z = p_X \otimes p_Y \text{ when } Z \text{ is discrete} \quad (24.12)$$

$$f_Z = f_X \otimes f_Y \quad \text{when } Z \text{ is continuous} \quad (24.13)$$

$$F_Z = F_X \otimes F_Y, \quad (24.14)$$

where $(g \otimes h)(x, y) = g(x)h(y)$.

Here p and f denote the probability mass and density functions, and F denotes the **CDF** (Cumulative Distribution Function). For instance, $F_Z(t) = \mathbb{E}(\{Z \leq t\})$ is the CDF of Z , and similarly for X and Y .

To see the last equation, assume that Z , X , and Y have respective dimension $d_1 + d_2$, d_1 , and d_2 . For any $(d_1 + d_2)$ -tuple t , let $t = t_1 :: t_2$ with parts of dimensions d_1 and d_2 . Remember that for tuples $s \leq t$ if and only if $s_i \leq t_i$ for all components i . Then, by equation (5):

$$\begin{aligned} F_Z(t) &= \mathbb{E}(\{Z \leq t\}) \\ &= \mathbb{E}(\{X \leq t_1 \wedge Y \leq t_2\}) \\ &= \mathbb{E}(\{X \leq t_1\}) \mathbb{E}(\{Y \leq t_2\}) \\ &= F_X(t_1) F_Y(t_2). \end{aligned}$$

Put another way: $\{\blacksquare \leq t\} = \{\blacksquare \leq t_1\} \otimes \{\blacksquare \leq t_2\}$ because the product of indicators represents the *logical and* of the conditions.

Example 24.5. Let P be a random point in the positive quadrant of the plane with distribution

$$D_P(\psi) = \int_x dx \int_y dy e^{-(x+y)} \{x > 0 \wedge y > 0\} \psi(x, y).$$

Let $\langle X, Y \rangle$ be the scalar components of P , i.e., $X = \text{proj}_1(P)$ and $Y = \text{proj}_2(P)$. You observe that the X coordinate is π . Find the distribution of Y given this observation.

For our purposes, it suffices to find $K_{Y|X}(y \mid \pi)$. But reading from the

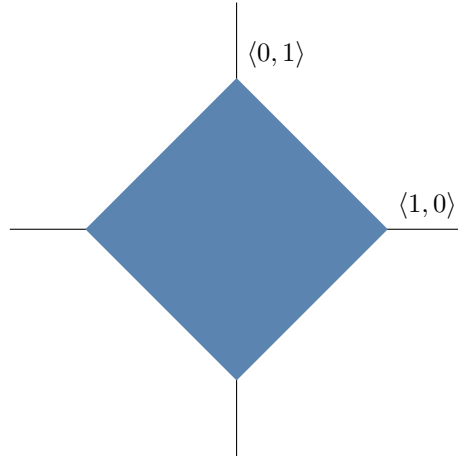
Distribution, we see that the Kernel for P is

$$K_P(x, y) = e^{-(x+y)} \{x > 0 \wedge y > 0\} = e^{-x} \{x > 0\} e^{-y} \{y > 0\}.$$

This Kernel factors, so X and Y are independent. Without any further calculation, we have that

$$K_{Y|X}(y | \pi) = K_Y(y) = e^{-y} \{y > 0\}.$$

Example 24.6. Let U be a 2-dimensional random variable with a Uniform($\mathcal{D}$) distribution, where $\mathcal{D}$ is the regular, diamond-shaped region:



Let $X = \text{proj}_1(U)$ and $Y = \text{proj}_2(U)$ be the scalar components of U . Note that $\mathcal{D}$ has area 2, so $K_U(x, y) = \frac{1}{2} \mathcal{D}(x, y)$.

We can see that X and Y are *not* independent. For instance, if you learn that $X > 0.99$, your prediction about Y will change because of the limited range for Y that that knowledge requires.

Define

$$S = Y + X$$

$$T = Y - X,$$

and notice that $X = (S - T)/2$ and $Y = (S + T)/2$.

Let $\zeta(x, y) = \langle y + x, y - x \rangle$. Notice that $\zeta^{\rightarrow}(\mathcal{D})$ is the *square region* with corners at $\langle \pm 1, \pm 1 \rangle$.

Then by equations (19.42) and (20.7), we have

$$D_{ST}(\varphi) = D_U(\varphi(\psi)) \quad (24.15)$$

$$= \int_x dx \int_y dy \frac{1}{2} \mathcal{D}(x, y) \varphi(y + x, y - x) \quad (24.16)$$

$$= \int_x dx \int_y dy \frac{2}{2} \frac{1}{2} \mathcal{D}(x, y) \varphi(y + x, y - x) \quad (24.17)$$

$$= \int_s ds \int_t dt \frac{1}{4} \{-1 \leq s \leq 1\} \{-1 \leq t \leq 1\} \varphi(s, t). \quad (24.18)$$

The last equation uses the fact that $\det \zeta'(x, y) = 2$ over $\mathcal{D}$ in equation (20.7). Thus, $V = \langle S, T \rangle$ is Uniform over the square region, with a Kernel that factors:

$$K_{ST}(s, t) = \frac{1}{2} \{-1 \leq s \leq 1\} \frac{1}{2} \{-1 \leq t \leq 1\} = K_S(s) K_T(t).$$

Hence, S and T are independent.

There are other representations of a distribution that we will see, but this factoring needs to be checked. Consider the Survival Distribution Function (SDF) defined by

$$S_Z(t) = \mathbb{E}(\{Z > t\}) = 1 - \mathbb{E}(\{Z \leq t\}) = 1 - F_Z(t).$$

In this cases, where $Z = X \star Y$,

$$\begin{aligned} S_Z(t) &= 1 - F_Z(t) \\ &= 1 - F_X(t_1) F_Y(t_2) \\ &= (1 - F_X(t_1))(1 - F_Y(t_2)) + F_X(t_1) + F_Y(t_2) \\ &\neq S_X(t_1) S_Y(t_2). \end{aligned}$$

This is true because the underlying statistic behind S is

$$\{\blacksquare > t\} = \{\blacksquare > t_1\} + \{\blacksquare > t_2\} - \{\blacksquare > t_1\} \otimes \{\blacksquare > t_2\},$$

which as a logical *or* does *not* factor. So, equation (5) does not hold.

Example 24.7. Two coins are flipped repeatedly until each comes up heads. Let M and N be the number of flips required for the first and second coin, respectively. (These counts include the last one that comes up heads.)

Assume that M and N are independent and have distributions:

$$\begin{aligned} D_M(\psi) &= \sum_{k=1}^{\infty} (1-p)^{k-1} p \psi(k), \\ D_N(\psi) &= \sum_{k=1}^{\infty} (1-q)^{k-1} q \psi(k), \end{aligned}$$

for some numbers $0 < p, q < 1$. We want to find $\mathbb{E}(\{M = N\})$ and $\mathbb{E}(\{M < N\})$.

Let $R = N \star M$. Because M and N are independent and discrete, we can write using the equation above:

$$D_R(\psi) = \sum_{j=1}^{\infty} \sum_{k=1}^{\infty} (1-q)^{j-1} q (1-p)^{k-1} p \psi(j, k),$$

for any statistic ψ compatible with R .

Now, $\{M = N\} = \psi_0(R)$ where statistic $\psi_0(r_1, r_2) = \{r_2 = r_1\}$. Thus,

$$\mathbb{E}(\{M = N\}) = D_R(\psi_0) = D_R(\langle m, n \rangle \mapsto \{m = n\}),$$

where the right-hand side expresses this equivalently with an anonymous statistic.

Hence,

$$\begin{aligned} \mathbb{E}(\{M = N\}) &= D_R(\psi_0) \\ &= \sum_j \sum_k K_R(j, k) \psi_0(j, k) \\ &= \sum_{j=1}^{\infty} \sum_{k=1}^{\infty} (1-q)^{j-1} q (1-p)^{k-1} p \{j = k\} \\ &= \sum_{j=1}^{\infty} \sum_{k=j}^j (1-q)^{j-1} q (1-p)^{k-1} p \\ &= \sum_{j=1}^{\infty} (1-q)^{j-1} q (1-p)^{j-1} p \\ &= pq \sum_{j=1}^{\infty} [(1-p)(1-q)]^{j-1} \\ &= pq \sum_{j=0}^{\infty} [(1-p)(1-q)]^j \end{aligned}$$

We can derive these distributions from first principles as well.

$$= \frac{pq}{1 - (1-p)(1-q)},$$

where the last equation derives from the sum of a geometric series. (See the **Help Sheet** in Appendix H.)

Similarly $\{M < N\} = \psi_1(R)$, with statistic $\psi_1(r_1, r_2) = \{r_2 - r_1 \in [1 \dots]\}$. So, transforming by this statistic, we have

$$\begin{aligned} \mathbb{E}(\{M < N\}) &= D_R(\langle m, n \rangle \mapsto \{m < n\}) \\ &= \sum_j \sum_k K_R(j, k) \{j < k\} \\ &= \sum_{j=1}^{\infty} \sum_{k=j+1}^{\infty} (1-q)^{j-1} q (1-p)^{k-1} p \\ &= q \sum_{j=1}^{\infty} (1-q)^{j-1} p \sum_{k=j+1}^{\infty} (1-p)^{k-1} \\ &= q \sum_{j=1}^{\infty} (1-q)^{j-1} (1-p)^j p \sum_{k=0}^{\infty} (1-p)^k \\ &= q(1-p) \sum_{j=1}^{\infty} (1-q)^{j-1} (1-p)^{j-1} p \sum_{k=0}^{\infty} (1-p)^k \\ &= q(1-p) \sum_{j=1}^{\infty} (1-q)^{j-1} (1-p)^{j-1} p \frac{1}{1-(1-p)} \\ &= q(1-p) \sum_{j=1}^{\infty} (1-q)^{j-1} (1-p)^{j-1} \\ &= q(1-p) \sum_{j=0}^{\infty} [(1-q)(1-p)]^j \\ &= \frac{q(1-p)}{1 - (1-p)(1-q)}. \end{aligned}$$

The calculation of $\mathbb{E}(\{M > N\})$ would be almost the same with q and p reversed.

As a check on our calculation, notice that by logical equivalence

$$\{M < N\} + \{M = N\} + \{M > N\} = 1,$$

so taking expectations and using the Additivity Rule, we should have

$$\frac{q(1-p) + qp + p(1-q)}{1 - (1-p)(1-q)} = 1,$$

which holds because

$$q(1-p) + qp + p(1-q) = p + q - pq$$

and

$$1 - (1-p)(1-q) = p + q - pq.$$

We defined independence for two random variables (of arbitrary dimension), and the concept extends directly to an arbitrary collection of random variables $Z_1, Z_2, \dots, Z_n$. We say that these are **(mutually) independent** if for every partition¹ of $[1..n]$ into two subsets $\mathcal{I}_1, \mathcal{I}_2$, we have that $Z_{\mathcal{I}_1} = ::_{i \in \mathcal{I}_1} Z_i$ and $Z_{\mathcal{I}_2} = ::_{i \in \mathcal{I}_2} Z_i$ are independent random variables.² All the earlier results extend to this case; for instance,

$$D_{Z_{\mathcal{I}_1} | Z_{\mathcal{I}_2}}(\psi | v) = D_{Z_{\mathcal{I}_1}}(\psi) \quad (24.19)$$

$$D_{Z_1 Z_2 \dots Z_n}(\psi_1 \otimes \dots \otimes \psi_n) = D_{Z_1}(\psi_1) D_{Z_2}(\psi_2) \dots D_{Z_n}(\psi_n) \quad (24.20)$$

$$K_{Z_1 Z_2 \dots Z_n} = K_{Z_1} \otimes K_{Z_2} \otimes \dots \otimes K_{Z_n}. \quad (24.21)$$

These equations can seem overwhelming at first, so let's take a closer look. Equation (24.19) says that if we learn the values of **any subset** of the Z_i 's, it **does not change our predictions** about the remaining Z_i 's. This is a direct generalization of our first interpretation of independence. Learning the values of some independent random variables does not change our predictions about the others.

Equation (24.20) says that **separately transforming independent random variables produces independent random variables**. Let $Z_1, Z_2, \dots, Z_n$ be independent random variables and $\psi_1, \psi_2, \dots, \psi_n$ be respectively compatible scalar statistics. Then,

$$(\psi_1 \otimes \psi_2 \otimes \dots \otimes \psi_n)(Z_1, Z_2, \dots, Z_n) = \psi_1(Z_1) \psi_2(Z_2) \dots \psi_n(Z_n).$$

So transforming $\langle Z_1, Z_2, \dots, Z_n \rangle$ with the statistic on $(\psi_1 \otimes \psi_2 \otimes \dots \otimes \psi_n)$ means (i) transforming each Z_i by a statistic ψ_i , separately, and (ii) combining the results in a product. Separate transforms of independent random variables are still independent. A common special case: for *independent*, scalar random variables, the expectation of

¹A partition of a set divides it into mutually exclusive and collectively exhaustive subsets. See Example 14.1.

² $::_{c \in \mathcal{C}} Z_c = Z_{c_1} :: Z_{c_2} :: \dots :: Z_{c_n}$, when $\mathcal{C} = \{c_1, \dots, c_n\}$. This gives the random variable that joins these variables' values into one large tuple.

a product is the product of the expectations.

Example 24.8. Let $B = \langle B_1, \dots, B_n \rangle$ for some $n \in [1..)$ be random bits. Assume that the B_i 's are *independent* and that $\mathbb{E}(B_i) = p$ for some number $0 < p < 1$.

Then for any $b \in \{0, 1\}^n$,

$$\begin{aligned} \mathbb{E}(\{B = b\}) &= \mathbb{E}(\{B_1 = b_1 \wedge \dots \wedge B_n = b_n\}) \\ &= \mathbb{E}(\{B_1 = b_1\} \cdot \dots \cdot \{B_n = b_n\}) \\ &= \mathbb{E}(\{B_1 = b_1\}) \cdot \dots \cdot \mathbb{E}(\{B_n = b_n\}) \\ &= p^{\sum_i b_i} (1 - p)^{n - \sum_i b_i}. \end{aligned}$$

That is, the probability of seeing a specific pattern of bits is a product with one factor of p for every 1 bit in the pattern and one factor of $1 - p$ for every 0 bit in the pattern. Note that $\sum_i b_i$ is the number of 1 bits in b , and $n - \sum_i b_i$ is the number of 0 bits in b .

Here, the second equality reflects the fact that the indicator of a logical and is the product of the indicators. Because the terms in that product are transforms of independent random variables, the terms themselves are independent by equation (24.20). The third equality then follows from independence. Here, the third equality comes from independence since the product of events is $\psi(B)$, where the statistic ψ is a tensor product $\psi_1 \otimes \dots \otimes \psi_n$ with $\psi_i(b_i) = \{1\}(b_i)$. The last equality follows because each $\sum_i b_i$ is the number of 1's in b , which is the number of the expectations whose value is p , with the rest $1 - p$.

It is worth pointing out that it is possible for every pair among $Z_1, Z_2, \dots, Z_n$ to be independent without the entire collection being mutually independent. Mutual independence is a stronger property and one we more commonly use with large collections. By default, if we assume that a collection of random variables is independent, we mean mutually independent.³

³This is a convention.

If a collection of random variables $X_1, X_2, \dots, X_n$ are mutually independent, then **every subcollection is mutually independent as well**. We can see this from equations (24.20) and (24.21) by transforming with the projection statistic that maps onto the subcollection. In particular, for a mutually independent collection of random variables, each pair of random variables is independent. But the converse is not true, as the next example shows.

Example 24.9 Pairwise but not Mutual

Let X have a uniform distribution over $[1 \dots 4]$. Define events $A = \{X = 1 \vee X = 2\}$, $B = \{X = 1 \vee X = 3\}$, and $C = \{X = 2 \vee X = 3\}$. Notice that

$$\begin{aligned} D_A(\psi) &= D_B(\psi) = D_C(\psi) = \frac{1}{2} \psi(0) + \frac{1}{2} \psi(1) \\ D_{A|B}(\psi | b) &= \frac{1}{2} \psi(0) + \frac{1}{2} \psi(1) \\ D_{A|C}(\psi | c) &= \frac{1}{2} \psi(0) + \frac{1}{2} \psi(1) \\ D_{B|C}(\psi | c) &= \frac{1}{2} \psi(0) + \frac{1}{2} \psi(1), \end{aligned}$$

but $D_{A|BC}(\psi | 1, 1) = \psi(0)$. So, A, B, C are *not* mutually independent, though each pair is independent.

A weaker notion than independence is *exchangeability*. If X is an m -dimensional random variable, we say that X has an **exchangeable** distribution if $D_{\varphi(X)} = D_X$, for any *permutation statistic* φ , which simply rearranges the components.⁴ This says that rearranging the components does not change the Distribution/Kernel/Kind. As we often do, we apply a term to the random variable that describes its distribution: we say here that X is an exchangeable random variable.

⁴See Sections 15.3 and 18.10 and page 674.

Example 24.10.

A box contains one blue ball and two red balls. We draw balls at random one at a time from the box, without replacing the drawn balls. Assume that all balls in the box at any draw are equally likely to be chosen.

Let $B = \langle B_1, B_2, B_3 \rangle$ be the random variable that represents the chosen balls, where B_i is the event that the i th ball chosen is blue.

Then, B is exchangeable but they are *not* independent.

To see the latter claim, suppose I tell you that $B_1 = 1$ and $B_2 = 1$, then you *know* that $B_3 = 0$. That change in our predictions is enough to disprove independence.

To show exchangeability, we need to establish that B and $\rho(B)$ have the same distribution for all $6 = 3!$ permutations ρ of the components of B . There are several ways to do this, let's start by finding K_B .

We use the join equation, expressing B through a chain of joins $B = B_1 \triangleright (B_2 | B_1) \triangleright (B_3 | B_1, B_2)$, just as we would in `frplib`. The corresponding Kind has

Kernel

$$K_B(m, n, r) = K_{B_1}(m) K_{B_1|B_2}(n | m) K_{B_3|B_1, B_2}(r | m, n).$$

We can deduce each of these conditional Kernels from the simple binary Kind:

$$K_{B_1}(m) = \frac{1}{3} \{m = 0\} + \frac{2}{3} \{m = 1\}$$

$$K_{B_1|B_2}(n | m) = \{m = 0\} \{n = 1\} + \{m = 1\} \left[\frac{1}{2} \{n \in \{0, 1\}\} \right]$$

$$K_{B_3|B_1, B_2}(r | m, n) = \{m = 0 \vee n = 0\} \{r = 1\} + \{m = 1 \wedge n = 1\} \{r = 0\}.$$

Tabulating these, we have

$\frac{m}{0}$	$\frac{n}{0}$	$\frac{r}{0}$	$\frac{K_B(m, n, r)}{0}$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	$\frac{1}{3} \cdot 1 \cdot 1 = \frac{1}{3}$
1	0	0	0
1	0	1	$\frac{2}{3} \cdot \frac{1}{2} \cdot 1 = \frac{1}{3}$
1	1	0	$\frac{2}{3} \cdot \frac{1}{2} \cdot 1 = \frac{1}{3}$
1	1	1	0

Hence,

$$D_B(\psi) = \frac{1}{3} \psi(0, 1, 1) + \frac{1}{3} \psi(1, 0, 1) + \frac{1}{3} \psi(1, 1, 0)$$

and so

$$\begin{aligned} D_{\rho(B)}(\psi) &= D_B(\psi \circ \rho) \\ &= \frac{1}{3} \psi(\rho(0, 1, 1)) + \frac{1}{3} \psi(\rho(1, 0, 1)) + \frac{1}{3} \psi(\rho(1, 1, 0)) \\ &= \frac{1}{3} \psi(0, 1, 1) + \frac{1}{3} \psi(1, 0, 1) + \frac{1}{3} \psi(1, 1, 0). \end{aligned}$$

The last equality follows because we are averaging ψ over all permutations of $\langle 0, 1, 1 \rangle$, and it does not matter in which order we add up the terms. Thus, $D_{\rho(B)} = D_B$, and because ρ was an arbitrary permutation, the components of B are exchangeable (but not independent).

An important special case of independent joins that we have used frequently are *independent join powers*, where we take an independent join of the *same Kind* repeatedly. If $X_1, X_2, \dots, X_m$ are random variables that are (i) (mutually) independent,

and (ii) have the same Distribution, we say they are **independent and identically distributed**, or **IID** for short. In this case, the join $\langle X_1, X_2, \dots, X_m \rangle$ is just the independent join power $X_1 \star m$, consisting of m clones of X_1 . When referring to the common Distribution or Kernel of IID random variables, we often just use the Distribution or Kernel of the first variable arbitrarily in this way. The assumption that a collection of random variables is IID is a powerful one. We are decomposing a system (or part thereof) into multiple probabilistically identical units – clones. The IID assumption is also common in practice, especially with data-analysis methods, because it simplifies the analysis. But care should usually be taken to ensure that the assumption is reasonable and that the analysis is robust to slight deviations from this assumed model.

Example 24.11 CDF of the Maximum

In Example 19.9, we defined the CDF, or Cumulative Distribution Function, associated with a scalar random variable X . This is the function

$$F_X(t) = D_X(\{\blacksquare \leq t\}) = \mathbb{E}(\{X \leq t\})$$

that gives for each t the probability that X is less than or equal to t .

Three nice features of the CDF is that (i) two random variables with the same CDF have the same Distribution, (ii) we can find the Kernel from the CDF easily, and (iii) in some situations the CDF is often relatively simple to compute.

Let Y be an n -dimensional random variable with IID scalar components $Y_1, Y_2, \dots, Y_n$ and define

$$M = \text{Max}(Y) = \max(Y_1, Y_2, \dots, Y_n).$$

(In `frplib` terms, we are transforming Y by the `Max` statistic.)

We want to find the Distribution or Kernel of M , and we will use the CDF and the logic of events.

First, for any real t , then $M \leq t$ if and only if *all* the $Y_i \leq t$ because M is their maximum. This logical equivalence tells us that these events are equal:

$$\{M \leq t\} = \{Y_1 \leq t \wedge \dots \wedge Y_n \leq t\} = \{Y_1 \leq t\} \cdots \{Y_n \leq t\}.$$

Taking expectations and applying independence we get

$$\begin{aligned}
 F_M(t) &= \mathbb{E}(\{M \leq t\}) \\
 &= \mathbb{E}(\{Y_1 \leq t\}\{Y_2 \leq t\} \cdots \{Y_n \leq t\}) \\
 &= \mathbb{E}(\{Y_1 \leq t\}) \mathbb{E}(\{Y_2 \leq t\}) \cdots \mathbb{E}(\{Y_n \leq t\}) \\
 &= \mathbb{E}(\{Y_1 \leq t\})^n \\
 &= F_{Y_1}^n(t).
 \end{aligned}$$

The third equality follows by independence as the product of indicators factors. The fourth equality follows because all the Y_i 's have the same Distribution.

As we will see in the next Chapter, we can obtain the Kernel of M directly from F_M . As a preview, assume that the Y_i 's are all continuous. Then,

$$K_M(t) = F'_M(t) = n F_{Y_1}^{n-1}(t) K_{Y_1}(t).$$

Puzzle 171. Let Y be an n -dimensional random variable with IID scalar components $Y_1, Y_2, \dots, Y_n$ and define

$$M = \text{Min}(Y) = \min(Y_1, Y_2, \dots, Y_n).$$

By analogy with the last example, use the SDFs $S_M(t) = \mathbb{E}(\{M > t\})$ and $S_{Y_i}(t) = \mathbb{E}(\{Y_i > t\})$ to find the Distribution of M in terms of the common Distribution of the Y_i 's.

24.2 Conditional Independence

Independence of random variables tells us that the value of one does not change our predictions about the other. This relationship is affected by what we know. *conditional independence* is *exactly the same idea* but expanded in scope to make the relationship contingent on a specific knowledge state. Two random variables are conditionally independent given some knowledge (i.e., observation) if learning the value of one of the variables – **in addition to the given knowledge** – does not change our predictions of the other **with the given knowledge**.

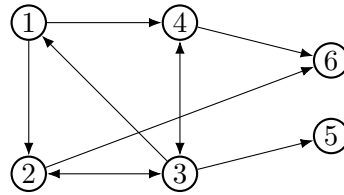
A common knowledge state to account for is knowledge of the values of one or more random variables. We say that random variables X and Y , of arbitrary

dimensions, are **conditionally independent given another random variable Z** (of arbitrary dimension) if *when we know the value of Z* , observing the value of X does not change our predictions about Y .⁵

⁵We delve even more deeply into this idea in Chapter 27.

Example 24.12 State Transitions

A robot moves randomly among locations 1–6 as described by the graph below. The robot starts at location 1. When at a location with $k \geq 1$ outgoing arrows, the robot moves at random to each corresponding neighboring location (the target of an arrow) with probability $1/k$. If the location has no outgoing arrows, the robot stops.



The random variables $R_0, R_1, R_2, \dots$ describe the robot's location after successive moves, where R_n 's value gives the robot's location after n moves.

Notice that when a robot is in a location ℓ , its next move only depends on the current location, *not on how it got there*. We can express this in terms of conditional independence: R_0 and R_2 are conditionally independent given R_1 , R_3 and $\langle R_0, R_1 \rangle$ are conditionally independent given R_2 , R_4 and $\langle R_0, R_1, R_2 \rangle$ are conditionally independent given R_3 , and so forth. This means, for instance, that the following hold for every valid setting of r_0, r_1, r_2, r_3 :

$$D_{R_2|R_1, R_0}(\psi \mid r_1, r_0) = D_{R_2|R_1}(\psi \mid r_1) \quad (24.22)$$

$$D_{R_3|R_2, R_1, R_0}(\psi \mid r_2, r_1, r_0) = D_{R_3|R_2}(\psi \mid r_2) \quad (24.23)$$

$$D_{R_4|R_3, R_2, R_1, R_0}(\psi \mid r_3, r_2, r_1, r_0) = D_{R_4|R_3}(\psi \mid r_3). \quad (24.24)$$

In fact, we can make a stronger statement: *the robot's future path is conditionally independent of its history given its current position!* This is the **Markov property** that we used in Chapter 7: the future is conditionally independent of the past given the present.

Example 24.13. Alice is flying from Pittsburgh to San Francisco, and at the same time, Bob is driving from Pittsburgh to Atlanta.

Alice's and Bob's travel times are not independent because if we learned that Alice's flight was substantially delayed than usual, we might predict a longer travel time for Bob as well. Alice's delay might, for instance, reflect weather conditions near Pittsburgh that would also slow Bob's drive.

It might be reasonable to say, however, that Alice's and Bob's travel times are conditionally independent given the weather in Pittsburgh on that day.

The mathematical meaning of conditional independence directly generalizes our conditions for independence.

Let X , Y , and Z be random variables of arbitrary dimension. Then, X and Y are **conditionally independent given Z** if and only if any of the following hold:

$$1. D_{Y|XZ}(\psi | x, z) = D_{Y|Z}(\psi | z) \quad (24.25)$$

$$2. K_{Y|XZ}(y | x, z) = K_{Y|Z}(y | z) \quad (24.26)$$

$$3. D_{X|YZ}(\varphi | y, z) = D_{X|Z}(\varphi | z) \quad (24.27)$$

$$4. K_{X|YZ}(x | y, z) = K_{X|Z}(x | z) \quad (24.28)$$

Recall: $(\varphi \otimes \psi)(x :: y) = \varphi(x) \cdot \psi(y)$

$$5. D_{XY|Z}(\varphi \otimes \psi | z) = D_{X|Z}(\varphi) D_{Y|Z}(\psi | z) \quad (24.29)$$

$$6. K_{XY|Z}(x, y | z) = K_{X|Z}(x | z) K_{Y|Z}(y | z) \quad (24.30)$$

Notice that these conditions exactly parallel those of independence with the inclusion of Z . And as with independence, we can extend this definition to more than two random variables, giving us *mutual* conditional independence. For instance, if U , V , and W are (mutually) conditionally independent given Z , then

$$K_{UVW|Z}(u, v, w | z) = K_{U|Z}(u | z) K_{V|Z}(v | z) K_{W|Z}(w | z).$$

Notice that the Z is given in *every term*.

It's the same idea with any number of random variables. If $X_1, X_2, \dots, X_n$ and Z are random variables of arbitrary dimension then $X_1, X_2, \dots, X_n$ are conditionally

mutually independent given Z when

$$D_{X_1 X_2 \dots X_n | Z}(\psi_1 \otimes \dots \otimes \psi_n | z) = \prod_{i=1}^n D_{X_i | Z}(\psi_i | z) \quad (24.31)$$

$$K_{X_1 X_2 \dots X_n | Z}(x | z) = \prod_{i=1}^n K_{X_i | Z}(x_i | z). \quad (24.32)$$

Again, by default, when we say conditionally independent, we mean conditionally mutually independent. In general, $X_1, X_2, \dots, X_n$ are conditionally independent given Z if **knowing the values of any subset of the X_i 's and Z does not change our predictions (given Z) of the other X_i 's.**

Puzzle 172. Suppose Z is a constant random variable. What does it mean for X and Y to be conditionally independent given Z ?

Because Z is a constant, its value is known, *a priori*.

Convince yourself that X and Y are conditionally independent given (constant) Z if and only if X and Y are *independent*. Thus, the definition of independence follows as a special case of conditional independence.

Example 24.14. Let random variables A, U, W have a $\text{DiscreteUniform}\langle 1, 2, 3, 4 \rangle$ distribution be independent.

Define $B = A + U$ and $C = A + W$. Then B and C are *not* independent. (Why?)

But B and C are *conditionally independent* given A . In other words, once we know the value of A , learning the value of B does not change our predictions about C , and vice versa.

That is: $D_{B|CA} = D_{B|A}$ and $D_{C|BA} = D_{C|A}$. Note that A is given throughout these equations.

Example 24.15. Let $\langle X, Y \rangle$ be Uniform inside the unit square, and define $Z := X + Y$.

Without knowledge of Z , X and Y are independent, by definition.

But given Z , X and Y are no longer independent. The reason is that when you know $Z = z$, learning the value of Y tells you the value of X exactly. Our predictions thus change, so $D_{X|YZ} \neq D_{X|Z}$. Y has a $\text{Uniform}\langle \max(0, z -$

$1), \min(1, z)\rangle$ distribution.

But if you know that $Z = z$ and $X = x$, then Y must equal $z - x$.

That is, our predictions have changed with knowledge of X .

24.3 The Updating Equation

Our analysis of a random process gives us predictions that can support our making good decisions. But to make good decisions in practice, we need to base our predictions on the latest information available. We thus need away to update our predictions for the information we observe *as the process evolves*. In fact, we already know how to do this by constraining a Kind with a conditional. We erase all the branches that are inconsistent with our observations, and all the remaining branches retain the same relative weight that they had. With this operation, we can update our predictions and adjust our decision rules for any new information. This section introduces the *updating equation*, a distillation of an observation into a form that is often useful.

The join equation (24.3) tells us how to get the Kind of a join from a Kind and conditional Kind: $K_{XY} = K_X \triangleright K_{Y|X}$. But we can rearrange this to *solve* for the conditional Kind in terms of the join and marginal Kinds. This gives us $K_{Y|X} = K_{XY}/K_X$.

We can get the same result by constraining with an observation. Suppose we observe that X has the value x . The join Kind of X and Y given that condition erases all branches for which $X \neq x$. The branches that are left have weights proportional to $K_{XY}(x, y)$. To normalize these weights, we simply add up all the weights for which X has the value x ; this is the same as the Kind of $\text{proj}_1(\langle X, Y \rangle)$, which is $K_X(x)$. Hence, again, $K_{Y|X}(y | x) = K_{XY}(x, y)/K_X(x)$.

Whichever path we take to derive it, the result is the **Updating Equation**:

$$K_{Y|X}(y | x) = \frac{K_{XY}(x, y)}{K_X(x)}. \quad (24.33)$$

This is well defined only for values x that are *possible*, i.e., $K_X(x) > 0$. It works for random variables of any dimension and for random variables that are discrete, continuous, or otherwise.

Example 24.16. Suppose that $\langle X, Y \rangle$ is a two-dimensional random variable

with kernel

$$K_{XY}(a, b) = \frac{1}{9} \{b \in [0 \dots 2 - |a|]\} \{a \in [-2 \dots 2]\}.$$

(What are the possible values of $\langle X, Y \rangle$? Draw a picture.)

If we observe that X equals 1, what do we know about Y ? Apply the Updating Equation:

$$\begin{aligned} K_{Y|X}(b | 1) &= \frac{K_{XY}(1, b)}{K_X(1)} \\ &= \frac{K_{XY}(1, b)}{K_{XY}(1, 0) + K_{XY}(1, 1)} \\ &= \frac{1}{2} \{b = 0 \vee b = 1\}. \end{aligned}$$

We started with equal weights, so after constraining with an observation, the remaining weights continue to be equal. But only $\langle 1, 0 \rangle$ and $\langle 1, 1 \rangle$ are possible values of $\langle X, Y \rangle$.

Notice that $K_X(1) = K_{XY}(1, 0) + K_{XY}(1, 1)$, which is the sum of all weights in the Kind for X and Y for which X equals 1.

Example 24.17. Let $\langle U, V \rangle$ be the two-dimensional continuous random variable with kernel

$$K_{UV}(s, t) = 2 \{0 \leq 0 \leq s^2\} \{0 \leq s \leq 1\}.$$

The point $\langle U, V \rangle$ is a random point chosen uniformly from the region beneath the parabola $\langle s \rangle \mapsto s^2$ and above the horizontal axis, from 0 to 1. Suppose we observe that V has the value $1/4$. What do we know about U ?

Notice that the kernel has height 2 because it measures a density. The area of the region of possible values of $\langle U, V \rangle$ is $1/2$, and the weights are equal. So the density value is 2, which gives a total mass of 1.

Applying the Updating Equation,

$$\begin{aligned} K_{U|V}(s | 1/4) &= \frac{K_{UV}(s, 1/4)}{K_V(1/4)} \\ &= \frac{K_{UV}(s, 1/4)}{\int_{s=1/2}^1 ds K_{UV}(s, 1/4)} \\ &= \frac{\{1/2 \leq s \leq 1\}}{1/2} \end{aligned}$$

$$= 2 \{1/2 \leq s \leq 1\},$$

which is again uniform on the interval $[1/2, 1]$. Again, notice that $K_{V(1/4)} = \int_{s=1/2}^1 ds K_{UV}(s, 1/4)$, adding up all weights in the join where V equals $1/4$.

In the previous section, we used the Expectation Updating equation (23.10):

$$\mathbb{E}(Y \mid \mathcal{C}) = \frac{\mathbb{E}(Y \{ \mathcal{C} \})}{\mathbb{E}(\{ \mathcal{C} \})},$$

when $\mathbb{E}(\{ \mathcal{C} \}) > 0$. This follows directly from the Updating equation, as we will see here. The observational constraint eliminates all branches inconsistent with that condition, and we take the weighted average of the remaining values and weights in the resulting Kind.

Let $\mathcal{C}$ be a Boolean expression that defines a condition, with associated event $\{ \mathcal{C} \}$, and assume that $\mathbb{E}(\{ \mathcal{C} \}) > 0$. As a convenience for labeling, define $V = \{ \mathcal{C} \}$. The Updating equation tells us that

$$K_{Y|V}(y \mid 1) = \frac{K_{YV}(y, 1)}{K_V(1)}.$$

This is just the Expectation Updating equation, but it will be nice to see why.

To start, let's see what happens when Y is discrete.

$$\begin{aligned} \mathbb{E}(Y \mid \mathcal{C}) &= \sum_y y K_{Y|V}(y \mid 1) \\ &= \sum_y y \frac{K_{YV}(y, 1)}{K_V(1)} \\ &= \sum_y y \frac{K_{YV}(y, 1)}{\mathbb{E}(V)} \\ &= \frac{\sum_y y K_{YV}(y, 1)}{\mathbb{E}(\{ \mathcal{C} \})} \\ &= \frac{\sum_{v=0}^1 \sum_y y v K_{YV}(y, v)}{\mathbb{E}(\{ \mathcal{C} \})} \\ &= \frac{\mathbb{E}(Y \{ \mathcal{C} \})}{\mathbb{E}(\{ \mathcal{C} \})}. \end{aligned}$$

The penultimate equality holds because the extra terms are all 0 (corresponding to $v = 0$). Notice how the last equation captures the process of constraining with an observation. We multiply by $\{ \mathcal{C} \}$ to eliminate branches that are inconsistent with the

condition. We average the result and then re-normalize by the portion of branches that are retained.

Now let's suppose that Y has a simple Kernel with

$$D_Y(\psi) = \sum_{a \in \text{range}_d(Y)} K_Y(a) \psi(a) + \int_{a \in \text{range}_c(Y)} da \in \text{range}_c(Y) f_Y(a) \psi(a).$$

Then, we get the following which consists of the same basic steps as in the discrete case. For clarity, we will drop the $\text{range}_d(Y)$ and $\text{range}_c(Y)$ below; assume the sums are over discrete values of Y and the integrals are over the continuous values of Y .

$$\begin{aligned} \mathbb{E}(Y \mid \mathcal{C}) &= \mathbb{E}(Y \mid V = 1) \\ &= D_{Y|V}(\text{id} \mid 1) \\ &= \sum_a K_{Y|V}(a \mid 1) \text{id}(a) + \int_a da K_{Y|V}(a \mid 1) \text{id}(a) \\ &= \sum_a \frac{K_{YV}(a, 1)}{K_V(1)} a + \int_a da \frac{K_{YV}(a, 1)}{K_V(1)} a \\ &= \sum_a \frac{K_{YV}(a, 1)}{\mathbb{E}(V)} a + \int_a da \frac{K_{YV}(a, 1)}{\mathbb{E}(V)} a \\ &= \frac{\sum_a K_{YV}(a, 1) a + \int_a da K_{YV}(a, 1) a}{\mathbb{E}(V)} \\ &= \frac{\sum_j \sum_a K_{YV}(a, j) a \{j = 1\} + \sum_j \int_a da K_{YV}(a, j) a \{j = 1\}}{\mathbb{E}(V)} \\ &= \frac{\mathbb{E}(YV)}{\mathbb{E}(V)} \\ &= \frac{\mathbb{E}(Y \{ \mathcal{C} \})}{\mathbb{E}(\{ \mathcal{C} \})}. \end{aligned}$$

Same technique, same result. Recall that $V = \{V = 1\}$ because it is an event.

Again, the Expectation Updating equation requires $\mathbb{E}(\{ \mathcal{C} \}) > 0$. There is an important case where this does not hold. Suppose X is a continuous random variable and we are interested in the condition that $X = x$ for some *possible* value x . Then, $K_X(x) > 0$, which tells us that the *density* of probability is positive. But a continuous smear of mass leaves 0 at x . That is, $\mathbb{E}(\{X = x\}) = 0$. It seems counter-intuitive that the probability of being any value be zero when the probability that it equals *some* value is one. This is the nature of the continuum and the reason that we use densities rather than masses for the weights at continuous values.

One implication is that we cannot use equation (23.10) to compute $\mathbb{E}(Y \mid X = x)$

at a continuous value of X . The good news is that we do not have to. This conditional expectation is well defined and can be computed directly from the Updating Equation. For a possible value x ,

$$\begin{aligned}\mathbb{E}(Y \mid X = x) &= \int_y y K_{Y|X}(y \mid x) dy \\ &= \frac{1}{K_X(x)} \int_y y K_{XY}(x, y) dy,\end{aligned}$$

which is well defined because $K_X(x) > 0$, i.e., x is a *possible value* of X . Note that in the continuous case $K_X(x)$ is *not* the probability that $X = x$ but the density of probability *near* x .

Example 24.18.

Let D_1 and D_2 represent the rolls of two balanced, six-sided dice, with $\langle D_1, D_2 \rangle$ having Kind `uniform(1, 2, ..., 6) ** 2`.

What is $\mathbb{E}(D_2 \mid D_1 + D_2 = 9)$? We can compute this directly from the Kinds: the condition that $D_1 + D_2 = 9$ leaves only the branches $\langle 3, 6 \rangle, \langle 4, 5 \rangle, \langle 5, 4 \rangle, \langle 6, 3 \rangle$, all equally weighted. So the expectation of D_2 must be 4.5.

Now let's apply equation (23.10):

$$\begin{aligned}\mathbb{E}(D_2 \mid D_1 + D_2 = 9) &= \frac{\mathbb{E}(D_2 \{D_1 + D_2 = 9\})}{\mathbb{E}(\{D_1 + D_2 = 9\})} \\ &= \frac{\frac{1}{36}(6 + 5 + 4 + 3)}{\frac{4}{36}} \\ &= \frac{18}{4} = 4.5,\end{aligned}$$

which shows how equation (23.10) captures the operation of constraining with an observation then taking expectations.

24.4 The Multiplication Rule

The Updating and Expectation Updating equations lead to a useful and related pair of identities that we dub the **multiplication rule**. The first identity gives a way to compute the expectation for a *product of events*.

If $A_1, A_2, \dots, A_n$ are arbitrary *events*, their product $A_1 A_2 \cdots A_n$ is also an event. It is the event that occurs if and only if *all* of the A_i 's occur. The expectation of the product is the probability that all the events occur. We have the following.

Multiplication Rule (Expectation Version). If $A_1, A_2, \dots, A_n$ are *events*, then

$$\begin{aligned} \mathbb{E}(A_1 A_2 \cdots A_n) &= \mathbb{E}(A_1) \\ &\quad \cdot \mathbb{E}(A_2 \mid A_1 = 1) \\ &\quad \cdot \mathbb{E}(A_3 \mid A_2 = 1 \wedge A_1 = 1) \\ &\quad \cdots \\ &\quad \cdot \mathbb{E}(A_{n-1} \mid A_{n-2} = 1 \wedge \cdots \wedge A_1 = 1) \\ &\quad \cdot \mathbb{E}(A_n \mid A_{n-1} = 1 \wedge \cdots \wedge A_1 = 1). \end{aligned} \quad (24.34)$$

This decomposes the probability of all the events occurring into a product of probabilities of each event occurring *given* that all the “earlier” events occurred.

If we expand all the terms on the right-hand side of (24.34) via the Expectation Updating equation we get a “miraculous” cancellation that yields the left-hand side of the equation.

$$\mathbb{E}(A_1) \cdot \frac{\mathbb{E}(A_2 A_1)}{\mathbb{E}(A_1)} \cdot \frac{\mathbb{E}(A_3 A_2 A_1)}{\mathbb{E}(A_2 A_1)} \cdots \frac{\mathbb{E}(A_{n-1} A_{n-2} \cdots A_1)}{\mathbb{E}(A_{n-2} A_{n-3} \cdots A_1)} \cdot \frac{\mathbb{E}(A_n A_{n-1} \cdots A_1)}{\mathbb{E}(A_{n-1} A_{n-2} \cdots A_1)}.$$

Here, we use the fact that for events $A_i = \{A_i = 1\}$. Note that the product of the events is commutative, so we can list them in any order. Thus, the designation of which events are “earlier” than the others is up to us. Note also that if the events are *independent*, then equation (24.34) reduces to the product of expectations $\mathbb{E}(A_1)\mathbb{E}(A_2)\cdots\mathbb{E}(A_n)$ because the observational constraints do not change our predictions.

The same logic applies with the Updating equation for Kernels. Consider for example the random variable $\langle A, B, C, D, E \rangle$. We can write the join Kernel K_{ABCDE} as

$$K_{ABCDE} = K_A K_{B|A} K_{C|AB} K_{D|ABC} K_{E|ABCD},$$

where we exclude the arguments for the moment to show the structure of the rule. We decompose the Kernel to be the conditional Kernel of one variable at a time given all the “earlier” variables. In general, this looks like the following.

Multiplication Rule (Kernel Version). If $X_1, X_2, \dots, X_n$ are random variables,

then the join Kernel $K_{X_1, X_2, \dots, X_n}(x_1, x_2, \dots, x_n)$ can be decomposed as a product:

$$\begin{aligned} K_{X_1 X_2 \dots X_n}(x_1, x_2, \dots, x_n) &= K_{X_1}(x_1) \\ &\quad \cdot K_{X_2|X_1}(x_2 | x_1) \\ &\quad \cdot K_{X_3|X_1, X_2}(x_3 | x_1, x_2) \\ &\quad \dots \\ &\quad \cdot K_{X_n|X_1, X_2, \dots, X_{n-1}}(x_n | x_1, x_2, \dots, x_{n-1}). \end{aligned} \quad (24.35)$$

Each term in this decomposition is the Kernel of one X_i . Typically these are scalars, but the equation works for random variables of any dimensions. If all the X_i 's are independent, the observational constraints are irrelevant, and this reduces to the independent join of Kernels.

The reason equation (24.35) holds is basically the same as for the expectation version but using the Updating equation instead. Expand each term on the right-hand side via the Updating equation and cancel successive denominators and numerators, yielding the left-hand side.

$$\begin{aligned} &\frac{K_{X_1}(x_1) \frac{K_{X_1 X_2}(x_1, x_2)}{K_{X_1}(x_1)} \frac{K_{X_1 X_2 X_3}(x_1, x_2, x_3)}{K_{X_1 X_2}(x_1, x_2)}}{\dots} \\ &\quad \dots \frac{K_{X_1 \dots X_{n-1}}(x_1, \dots, x_{n-1})}{K_{X_1 \dots X_{n-2}}(x_1, \dots, x_{n-2})} \frac{K_{X_1 \dots X_{n-1}}(x_1, \dots, x_n)}{K_{X_1 \dots X_{n-1}}(x_1, \dots, x_{n-1})} \end{aligned}$$

Again, the ordering of the variables is arbitrary, so we can choose whichever ordering is most useful.

The Multiplication Rule (24.35) should look familiar: it is just repeated joins! We have the Kind of X_1 , the conditional Kind of X_2 given X_1 , the conditional Kind of X_3 given X_1 and X_2 and so forth, and we form their join, evolving the system and collecting the entire history. In terms of the Random Variables the join is

$$X_1 \triangleright (X_2 | X_1) \triangleright (X_3 | X_1, X_2) \triangleright \dots$$

passing the value at each stage to the next stage. We used this approach regularly in Part I, whether it was Theseus wandering in the labyrinth or Gollum trying to avoid the lava.

In these examples, we often made an assumption that the future evolution of a system depends only the system's current state and not how the system arrived at that state. We called this the *Markov property* in Chapter 7. Suppose Theseus is

wandering in the labyrinth by making a decision in each room about where to go next. If his decision on where to go next depends only on where he is, his random paths have the Markov property. But if his decision depends on his path to the current room – for instance, if he has recorded all the doors he has passed through and only goes through new doors – then the Markov property does not hold.

Definition 60. The Markov Property. A sequence of random variables $X_0, X_1, X_2, \dots$, is said to have the **Markov property** if for every $n > 0$,

$$X_{n+1}, X_{n+2}, \dots \text{ is conditionally independent of } X_0, \dots, X_{n-1} \text{ given } X_n.$$

If we view the X_i 's as the state of an evolving random system where i indexes “time” (e.g., ticks of a clock), then the Markov property says that the future evolution of the system is conditionally independent of its history given the current state.

Or more pithily: **the future is conditionally independent of the past given the present.**

The Markov property is a powerful simplifying assumption that is often realistic. With the Markov property, we can describe a system fully with the Distribution of the initial state X_0 and the conditional Distributions of $X_{n+1} | X_n$ that describe the state *transitions*. In many useful models, these transition conditional Distributions are the same for all n .

Let X be a random variable where for each $n \in [0..)$, the component random variable $X_n = \text{proj}_n(X)$ is a discrete random variable. We think of n as indexing “discrete time” with X_n representing the state of a system at time n . X is an infinite-dimensional random variable that describes the evolution of a system at distinct ticks of the clock over an infinite time horizon. It is called a discrete-time, discrete-state **stochastic process**. We will discuss stochastic processes in detail in Part IV, but here we look briefly at the implications of the Markov property.

If X is a discrete-time stochastic process, we call the sequence of states the system is in over a particular time interval a **path segment**. Specifically, for $m \leq n$ with $m, n \in [0..)$ the path segment $X_{m:n}$ is the $((n - m + 1)$ -dimensional random variable:

$$X_{m:n} = \langle X_m, \dots, X_n \rangle = \text{proj}_{m:n}(X). \quad (24.36)$$

This describes the states of the system at times m through n , which we visualize as a “path” through the set of possible states. It is convenient to use the same notation

to index states themselves at the corresponding times, with $s_{m:n} = \langle s_m, \dots, s_n \rangle$ a possible value of $X_{m:n}$, s_m a possible value of X_m , and so forth.

Similarly, for each $n \in [0..)$, define

$$\text{past}_n(X) = \langle X_0, X_1, \dots, X_{n-1} \rangle \quad (24.37)$$

$$\text{present}_n(X) = X_n \quad (24.38)$$

$$\text{future}_n(X) = \langle X_{n+1}, X_{n+2}, X_{n+3}, \dots \rangle. \quad (24.39)$$

The first random variable gives, the history of the system's evolution before time n , which is just $X_{0:(n-1)}$. (When $n = 0$, this is the 0-dimensional "empty" random variable.) The second gives the current state of the system at time n . The third gives a stochastic process that represents the future evolution of the system after time n . The Markov property for such a process posits that for every $n \in [0..)$, $\text{future}_n(X)$ is conditionally independent of $\text{past}_n(X)$ given $\text{present}_n(X)$. Once you know the current state of the system, none of your predictions about the future from that point on depend on how you got to that state.

Example 24.19 Theseus with Chalk

Theseus wanders about the labyrinth, but rather than choosing among every door out of his current room, he marks the door from which he entered with chalk and then chooses at random from among the rest. Unfortunately, the moisture in the room obscures the chalk within a few minutes after he leaves the room.

If random variables $R_0, R_1, \dots$, represent the rooms he visits, then this sequence does *not* have the Markov property because if Theseus is in room r , his next room depends not only on r but also on the previous room he was in.

But! If we change the state from just his current room to his current and previous room, we can recover the Markov property. For $n > 0$, define $S_n = \langle X_n, X_{n-1} \rangle$ which has values $\langle r', r \rangle$ where r and r' are consecutive rooms he visits. We take $S_0 = \langle X_0, Z \rangle$, where Z is a constant random variable whose value is any constant that is not a valid room.

Now, if we know $S_n = \langle r', r \rangle$, we know Theseus's current room and his previous room, so his future wanderings are determined, no matter how we got to the state $\langle r', r \rangle$. That is, the random variables $S_0, S_1, S_2, \dots$ have the Markov property.

Example 24.20 Markovian Paths

Let X be a discrete-time, discrete-state stochastic process that satisfies the Markov property. What can we predict about how the system evolves, and in particular about the paths that the system takes?

Let s_0 be a possible starting state, and consider possible paths $s_{0:n}$ through time n from that initial state. The Multiplication Rule and the Markov property tell us that the probability of any particular path is

$$\begin{aligned} K_{X_{0:n}}(s_{0:n}) &= K_{X_0}(s_0) \prod_{i=1}^n K_{X_i|X_{0:(i-1)}}(s_i | s_{0:(i-1)}) \\ &= K_{X_0}(s_0) \prod_{i=1}^n K_{X_i|X_{i-1}}(s_i | s_{i-1}), \end{aligned} \quad (24.40)$$

The first equation follows from the Multiplication Rule, and the second by applying the Markov property. Applying the Updating equation, we get

$$\begin{aligned} K_{X_{1:n}|X_0}(s_{1:n} | s_0) &= \frac{K_{X_{0:n}}(s_{0:n})}{K_{X_0}(s_0)} \\ &= \prod_{i=1}^n K_{X_i|X_{i-1}}(s_i | s_{i-1}). \end{aligned} \quad (24.41)$$

Notice that the same logic works for any “time” $m \in [0..)$ to get the probability of a path segment $s_{(m+1):(m+n)}$ starting at state s_0 :

$$K_{X_{(m+1):(m+n)}|X_m}(s_{(m+1):(m+n)} | s_m) = \prod_{i=1}^n K_{X_{m+i}|X_{m+i-1}}(s_{m+i} | s_{m+i-1}). \quad (24.42)$$

Let ψ be a n -dimensional statistic that is compatible with the path segments $X_{1:n}$. Then, we can make predictions about $\psi(X_{1:n})$ given a particular starting state $X_0 = s_0$:

$$\begin{aligned} D_{X_{1:n}|X_0}(\psi | s_0) &= \sum_{s_{1:n}} K_{X_{1:n}|X_0}(s_{1:n} | s_0) \psi(s_{1:n}) \\ &= \sum_{s_{1:n}} \prod_{i=1}^n K_{X_i|X_{i-1}}(s_i | s_{i-1}) \psi(s_{1:n}), \end{aligned} \quad (24.43)$$

We can view the statistic ψ as *scoring* the path, so this gives the predicted score of the path.

By picking an appropriate ψ , we can make predictions about various interesting features of the path. For example:

1. For a fixed state a , if

$$\psi(s_1, \dots, s_n) = \sum_{i=1}^n \{s_i = a\}, \quad (24.44)$$

then, random variable $\psi(X_{1:n})$ represents the number of times the system *visits* state a .

2. If r is a scalar function defined on the states that gives an associated “reward” for each state and

$$\psi(s_1, \dots, s_n) = \sum_{i=1}^n r(s_i), \quad (24.45)$$

then $\psi(X_{1:n})$ represents the accumulated reward obtained up to time n .

3. If r is as in the previous item, we can take

$$\psi(s_1, \dots, s_n) = \sum_{i=1}^n \alpha^n r(s_i), \quad (24.46)$$

for some $0 < \alpha \leq 1$. This gives the *discounted reward*, recognizing that reward in the present may be worth more to us than the same absolute reward in the future. (For instance, \$100 now is worth more than \$100 in 10 years because of inflation and the possibility of investing over that time horizon.)

4. If $\psi(s_1, \dots, s_n) = \{s_n = a\}$ for a fixed state a , then $\psi(X_{1:n})$ is the event that the system is in state a at time n .

There are many possibilities that make $D_{X_{1:n}|X_0}(\psi \mid s_0)$ useful to compute.

The sum in equation (24.43) looks a bit nasty, but it has nicer structure than it might seem at first. To see this, we will simplify in two ways.

First, assume in addition that the conditional Distributions describing the transitions, e.g., $K_{X_{n+1}|X_n}$, are the *same* for all n . Put another way, the transition

mechanism does not depend on the absolute time. In this case, we can write

$$K_{X_n|X_{n-1}}(s' | s) = K(s' | s) \quad (24.47)$$

for all $n \in [1..)$ and for all possible states s, s' . Then, our predicted score for any path segment is

$$D_{X_{1:n}|X_0}(\psi | s_0) = \sum_{s_{1:n}} \prod_{i=1}^n K(s_i | s_{i-1}) \psi(s_{1:n}). \quad (24.48)$$

Second, keeping the previous assumption, assume as well that the statistic ψ only depends on the final component, s_n . This holds, for instance, in #4 above, when $\psi(s_{1:n}) = \{s_n = a\}$. Define a matrix P , with rows and columns labeled by possible states for a fixed but arbitrary ordering of the states, where the element in row s and column s' is

$$P_s^{s'} = K(s' | s). \quad (24.49)$$

That is, matrix element $P_s^{s'}$ gives the probability of transitioning from state s to state s' in one step. We call P the **transition probability matrix**. (See Sections 17.2 and 17.3 in Interlude F.) Define a vector v with components $v_s = \psi(s)$.

We can now write equation (24.48) as

$$D_{X_{1:n}|X_0}(\psi | s_0) = (P)^n \cdot v, \quad (24.50)$$

the n th power of the matrix P times the vector v . If we define the covector w by $w^s = K_{X_0}(s)$, then by conditioning on X_0 , we have that

$$D_{X_{1:n}}(\psi) = w \cdot (P)^n \cdot v. \quad (24.51)$$

Our calculation becomes a simple matrix-vector calculation. We will exploit this in Part IV to solve interesting problems.

As described in interlude F, vectors are represented as columns and covectors as rows; we use subscripts for vector components and superscripts for covector components.

Example 24.21 Pattern Matching

Consider a binary pattern of length n , denoted as a string $c_1 c_2 \cdots c_n$ where each $c_i \in \{0, 1\}$. We use $\bar{c}_i = 1 - c_i$ to denote the opposite bit. Now suppose we have a sequence $B_1, B_2, B_3, \dots$ of random bits (e.g., random coin flips), and we let T be the number of bits we must examine in order until we *first* see the specified

pattern. For instance, in Example 8.14, we studied the number of flips required until first seeing two consecutive heads in which case the pattern has $n = 2$, $c_1 = 1$, and $c_2 = 1$.

We can describe this system with $n+1$ states (plus an a “start” state inhabited before the first transition) that correspond to seeing each of the following patterns:

$$\begin{aligned}
 0 : & \quad \overline{c_1} \\
 1 : & \quad c_1 \overline{c_2} \\
 2 : & \quad c_1 c_2 \overline{c_3} \\
 & \quad \vdots \\
 k : & \quad c_1 c_2 \cdots c_{k-1} \overline{c_k} \\
 & \quad \vdots \\
 n-1 : & \quad c_1 c_2 \cdots c_{n-1} \overline{c_n} \\
 n : & \quad c_1 c_2 \cdots c_{n-1} c_n
 \end{aligned}$$

These are mutually exclusive and collectively exhaustive possibilities. We reach state n for the first time after T transitions. After reaching state $k < n$, there is some $m \in [0..k)$ for which $c_{k-m+1}c_{k-m+2}\cdots c_{k-1}\overline{c_k} = c_1c_2\cdots c_m$. At that point, we start as if having seen $c_1\cdots c_m$ and proceed to the next transition. The future transitions from state k are conditionally independent of how we reached state k , so this process is a Markov Chain.

In the next Section, the Mighty Conditioning Identity will help us find $\mathbb{E}(T)$.

The Markov property is a powerful – and often quite realistic – modeling assumption that generalizes the assumption of independence in a useful and often manageable way. Sequences of random variables that satisfy the Markov property comprise an import class of stochastic processes.

Definition 61. A sequence of random variables $X_0, X_1, X_2, \dots$, finite or infinite, that satisfies the Markov property is called a **Markov Chain**.

We often use Markov chains to model the evolution of a system’s state over time and/or space. The Markov property allows us to derive strong predictions about the short-term and long-term behavior of this stochastic process. We have seen examples of such analysis in Part I, and we will develop the theory more completely in Part IV. The stochastic process described in Example 24.20 is a Markov Chain, and the

example gives a hint at some of the questions we can answer about the process's evolution. One important special case is the assumption in equation (24.47) that the transition mechanism is constant across time.

24.5 The Mighty Conditioning Identity

The conditioning operation ($//$ in `frplib`) starts with a conditional Kind of Y given X and *averages* this with the weighs in the Kind of X to compute the Kind of Y alone. This corresponds to taking a join and then projecting onto the values of Y . Figure 24.1 reproduces Figure 4.19 from Chapter 4. The conditional Kind produces two Kinds, one for input $\langle 0 \rangle$ and one for input $\langle 1 \rangle$. We condition on a Kind that produces $\langle 0 \rangle$ or $\langle 1 \rangle$ with some probabilities. The resulting Kind is the average of the two target Kinds using the weights from Kind we are conditioning on. To average the Kinds, we include all possible values in both trees in each tree, setting the weight of any missing branches to zero, and then we average the weights on each value with the weights from Kind to the right of the $//$ operator.

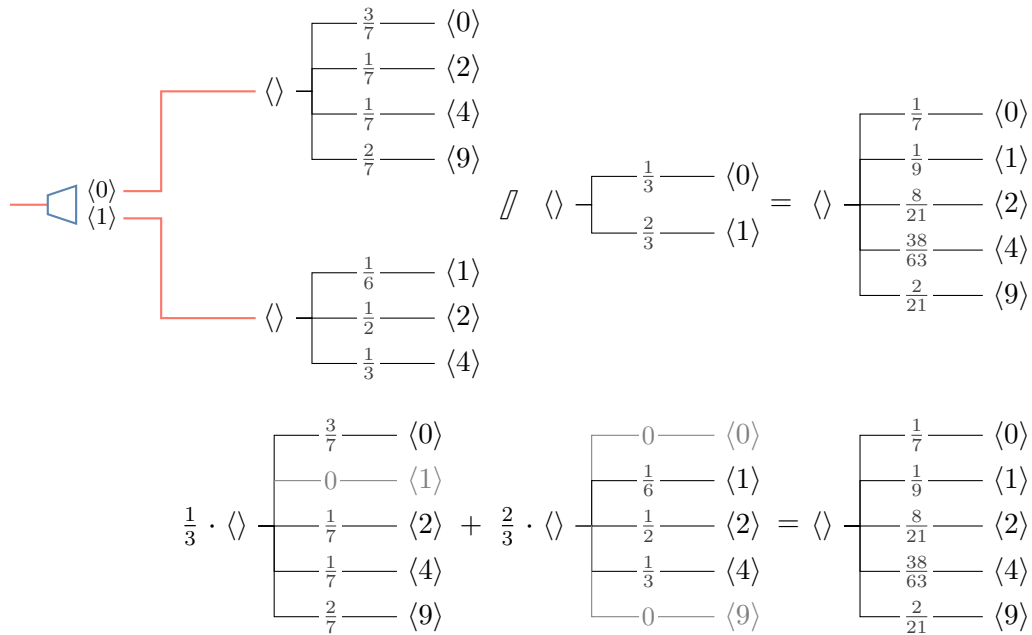


FIGURE 24.1. The conditioning operation as a weighted average over a conditional Kind by a Kind. The top panel combines a conditional Kind and a Kind with the conditioning operator $//$, producing the Kind at the right. The bottom shows how that Kind can be viewed as an average of Kinds.

View the conditional Kind as producing two *hypothetical* Kinds, two possibilities

depending on the input to the join. With each of these target Kinds, we can get a hypothetical answer to any question we might ask about the output Kind. But each hypothetical answer only applies with some probability, so averaging the trees – and averaging the hypothetical answers – gives us the predicted answer to our original question. This is the essence of the the *Mighty Conditioning Identity*, which is a general mathematical expression of the conditioning operation.

The Mighty Conditioning Identity embodies us a powerful method for solving problems that are otherwise very difficult to solve. This is the Method of Hypotheticals, which has the following steps:

1. We identify a *question* about Y that we want to answer.

As usual, we represent this question by a statistic ψ .

2. Our goal becomes to get a *predicted answer* to our question given whatever background knowledge we have.

That is, we want to find:

$$D_{Y|\mathcal{C}}(\psi) = \mathbb{E}(\psi(Y) \mid \mathcal{C}), \quad (24.52)$$

for some condition $\mathcal{C}$ which may just be $\top$, in which case we can drop the $\mid$ part.

3. We choose another random variable, call it X , to condition on. Call it the *conditioning variable*. There is some art to this choice as we will see.
4. The first step is to *pretend* that we know that X has the value x for some possible value. This gives us our original predicted answer enhanced with the extra knowledge that $X = x$:

$$\mathbb{E}(\psi(Y) \mid X = x \wedge \mathcal{C}).$$

Call this the *hypothetical answer* to our question.

5. But we do not really know the value of X , but we can find a hypothetical answer for any x that is a possible value of X . Imagine putting all of them in a table indexed by the value of x . That is just a *function*:

$$\langle x \rangle \mapsto \mathbb{E}(\psi(Y) \mid X = x \wedge \mathcal{C}).$$

This maps values of X to predicted answers, which are numbers. So it gives us a **statistic**.

Just for clarity let's give this statistic a name here $\varphi(x) = \mathbb{E}(\psi(Y) \mid X = x \wedge \mathcal{C})$.

6. The Mighty Conditioning Identity tells us that the answer to our original question *equals* the average value of this statistic weighted by the Distribution of $\mathcal{A}$. That is,

$$\mathbb{E}(\psi(Y) \mid \mathcal{C}) = \mathbb{E}(\varphi(X) \mid \mathcal{C}).$$

We have translated our problem to a simpler problem, at least with a good choice of X .

Taken together, these steps give us the Mighty Conditioning Identity.

The Mighty Conditioning Identity. If Y is a random variable, ψ a compatible statistic, and $\mathcal{C}$ an arbitrary condition, then

$$D_{Y|\mathcal{C}}(\psi) = D_{X|\mathcal{C}}(\langle x \rangle \mapsto D_{Y|X,\mathcal{C}}(\psi \mid x)). \quad (24.53)$$

Notice that the condition $\mathcal{C}$ plays the same role in every term; it is our background knowledge for the calculation. When $\mathcal{C} = \top$, we can drop it.

Applying this identity with conditioning variable X and condition $\mathcal{C}$ is called **conditioning on X** given $\mathcal{C}$.

Let's pull this apart from the inside out. The ψ is our original question, and $D_{Y|X,\mathcal{C}}(\psi \mid x) = \mathbb{E}(\psi(Y) \mid X = x \wedge \mathcal{C})$ is the hypothetical answer for one possible value x of X . Then the statistic $\langle x \rangle \mapsto D_{Y|X,\mathcal{C}}(\psi \mid x) = \mathbb{E}(\psi(Y) \mid X = x \wedge \mathcal{C})$ is the “lookup table” for hypothetical answers indexed by values of X . And

$$D_{X|\mathcal{C}}(\langle x \rangle \mapsto D_{Y|X,\mathcal{C}}(\psi \mid x))$$

is the expected hypothetical answer, the average of the hypothetical answers weighted by the Kernel of X . Finally, the Mighty Conditioning Identity tells us that this average equals the predicted answer to our original question.

Remember what we saw repeatedly in Chapter 4. Conditioning is just the operation of taking a join followed by a projection. The Mighty Conditioning Identity reflects that, including an expectation step, which interchanges with the join and projection. Indeed in `frplib`, we have for a conditional Kind `cK` and Kind `k`

```
(cK // k).expectation = cK.expectation // k.
```

This is just the Mighty Conditioning Identity.

The Mighty Conditioning Identity follows from the Updating and the Mass-Balancing equations. Let us show this for the case where X and Y are discrete. The general case is virtually the same except there is a sum and an integral (over the discrete and continuous parts) for both X and Y .

$$\begin{aligned}
 D_{X|C}(\langle x \rangle \mapsto D_{Y|X,C}(\varphi | x)) &= \sum_x \mathbb{E}(\varphi(Y) | X = x \wedge C) K_{X|C}(x) \\
 &= \sum_x \frac{\sum_y \varphi(y) K_{XY|C}(x, y)}{K_{X|C}(x)} K_{X|C}(x) \\
 &= \sum_x \sum_y \varphi(y) K_{XY|C}(x, y) \\
 &= \sum_x \sum_y \varphi(y) K_{XY}(x, y) \\
 &= \mathbb{E}(\varphi(Y) | C) \\
 &= D_{Y|C}(\varphi),
 \end{aligned}$$

as expected.

Conditioning is a powerful tool because we can *choose* the conditioning variable. We want to choose a conditioning variable X that balances two properties:

1. The ease of computing the hypothetical answers $D_{Y|X}(\varphi | x)$, and
2. The ease of averaging the hypothetical answers using the distribution of X .

There is a bit of art to this, and a good choice can make a complicated problem much easier.

The key step in applying the Mighty Conditioning Identity is writing the hypothetical solution in terms of the original problem we are trying to solve. We use the **same expression** for both with the addition of the extra information about the value of the conditioning variable in the observational constraint. The problem will usually be an expectation of some sort, so our solution is $\mathbb{E}(\varphi(Y) | C)$ and our hypothetical solution is $\mathbb{E}(\varphi(Y) | X = x \wedge C)$. (Though see Example 24.23 below where the problem is different while this same step works.) We then simply average the statistic of X that looks up the hypothetical solution for each value of X .

Example 24.22 **Example 24.22. Generating Random Numbers** You have three random number generators.

1. Generator 1 returns -1 or 1 with probability 1/2 each.

2. Generator 2 returns a uniform integer value on $[2 \dots 5]$
3. Generator 3 returns a *continuous* $\text{Uniform}(-1, 1)$ value.

You pick one of the generators at random (1 with probability 1/2, 2 with probability 1/3, and 3 with probability 1/6) and then generate a random number with the chosen generator.

What is the Distribution of the number you generate?

What is the expectation of the number you generate?

Define Relevant Random Variables

1. Let G be the generator chosen
2. Let N be the number generated

State What You Know. We know the distribution of G

$$D_G(\psi) = \frac{1}{2}\psi(1) + \frac{1}{3}\psi(2) + \frac{1}{6}\psi(3).$$

We know $D_{N|G}(\psi \mid k)$ for each $k \in [1 \dots 3]$.

State What You Want To Find We will *condition on* G to find the distribution of N .

Find it. The Mighty Conditioning Identity gives us that

$$\begin{aligned} D_N(\varphi) &= \sum_{k=1}^3 K_G(k) D_{N|G}(\varphi \mid k) \\ &= \frac{1}{2} \left(\frac{1}{2}\varphi(-1) + \frac{1}{2}\varphi(1) \right) + \frac{1}{3} \sum_{k=2}^5 \frac{1}{4}\varphi(k) + \frac{1}{6} \int_{t=-1}^1 dt \frac{1}{2}\varphi(t) \\ &= \frac{1}{4}\varphi(-1) + \frac{1}{4}\varphi(1) + \frac{1}{12} \sum_{k=2}^5 \varphi(k) + \frac{1}{12} \int_{t=-1}^1 dt \varphi(t). \end{aligned}$$

Notice that this Distribution has some discrete points and some continuous points. Hence,

$$\begin{aligned} \mathbb{E}(N) &= D_N(\text{id}) \\ &= -\frac{1}{4} + \frac{1}{4} + \frac{1}{12} \sum_{k=2}^5 k + \frac{1}{12} \int_{t=-1}^1 dt \varphi(t) \\ &= 0 + 0 + \frac{7}{6} + 0 = \frac{7}{6}. \end{aligned}$$

Example 24.23 Projections

If we have a join $\langle X, Y \rangle$ of dimension $d_1 + d_2$ (with X of dimension d_1), then $X = \text{proj}_{1..d_1}(\langle X, Y \rangle)$. We can find K_X by transforming K_{XY} with that projection statistic. To find $K_X(x)$, as we have seen, we add up the weights over all branches that project to the same value of x . That is, we add up the weights $K_X(x, v)$ over all possible values v of Y . This is the familiar projection operation, which yields

$$K_X(x) = \int_v dv \ K_{XY}(x, v).$$

But we can also derive this relationship by the Mighty Conditioning Identity, conditioning on Y . Here, the problem we are trying to solve is to find the Kernel $K_X(x)$. The hypothetical solution solves the *same problem* with just the extra information given about the value of Y . Where the original problem has solution $K_X(x)$, the hypothetical solution is $K_{X|Y}(x | v)$. Same problem with extra information.

Next, we average the hypothetical solutions against the Kernel of the conditioning variable Y . The Mighty Conditioning Identity becomes

$$K_X(x) = \int_v dv \ K_Y(v) K_{X|Y}(x | v).$$

By the join equation, the integrand is just $K_{XY}(x, y)$, recovering the projection equation.

Example 24.24 Phew!

You play a game of repeated trials, where each trial has one of three outcomes: “You win,” “You lose,” or “Try again.” A game is an independent join of trials up to the first one which you obtain “You win” or “You lose.” On any given trial, you get “You win,” “You lose,” and “Try again” with respective probabilities p , q , and r . What is the probability that you win the game?

Let W be the event that you win the game. Let S_n be the outcome of the n th trial, with values -1, 0, and 1 corresponding to “You lose,” “Try again,” and “You win.”

We will compute the distribution of W by conditioning on S_1 , the outcome of the first trial. Since W is an event, its Kind/Distribution is determined by its expectation, so it suffices to compute $D_W(\langle x \rangle \mapsto x) = \mathbb{E}(W)$.

The Mighty Conditioning Identity gives us:

$$\begin{aligned}\mathbb{E}(W) &= D_{S_1}(\langle s \rangle \mapsto \mathbb{E}(W \mid S_1 = s)) \\ &= \mathbb{E}(W \mid S_1 = -1)q + \mathbb{E}(W \mid S_1 = 0)r + \mathbb{E}(W \mid S_1 = 1)p \\ &= 0 + \mathbb{E}(W)r + p.\end{aligned}$$

Solving for $\mathbb{E}(W)$,

$$\mathbb{E}(W) = \frac{p}{1-r} = \frac{p}{p+q}.$$

The equality $\mathbb{E}(W \mid S_1 = 0) = \mathbb{E}(W)$ holds because when we get a “Try again,” we are back in the same state in which we started, for which our winning probability is therefore also the same.

Example 24.25 Monty Hall Revisited

The Monty Hall problem from Chapter 1 is a nice example for conditioning. Let C be the event that you chose the prize initially, and let W be the event that you win the prize with your selected strategy.

We know D_C , so we compute D_W by conditioning on C . Again, it is sufficient to compute the expectation of W .

$$\begin{aligned}\mathbb{E}(W) &= D_C(\langle c \rangle \mapsto \mathbb{E}(W \mid C = c)) \\ &= \frac{2}{3} \mathbb{E}(W \mid C = 0) + \frac{1}{3} \mathbb{E}(W \mid C = 1).\end{aligned}$$

The two expectations here depend on the strategy. Under the “Don’t Switch” strategy, the first is 0 and the second is 1, giving $\mathbb{E}(W) = 1/3$. Under the “Switch” strategy, the first is 1 and the second is 0, giving $\mathbb{E}(W) = 2/3$. This is the result we saw in Chapter 1.

Example 24.26 Craps

The game of craps is played with two 6-sided dice. The player throws the two dice the first time, with the following results:

- If the sum is 7 or 11, she wins.
- If the sum is 2, 3, or 12, she loses.

- Otherwise, she continues to throw the dice until the sum of the two dice is either equal to the sum for her first throw (she wins) or is either a 7 or 11 (she loses).

Find the probability that the player wins, assuming that the dice are balanced (uniform) and that all rolls are independent.

One way to think about the Mighty Conditioning Identity is that it allows us to (temporarily) hold fixed some part of the system that is making our analysis difficult. In the game of Craps, the later part of the game depends on what we get on the first roll. If we *knew* that roll, then the rest of the game is simple to analyze because what we need to win or lose is fixed. So we condition on the value of the first roll.

After the first roll, we have repeated, independent trials where we either win, lose, or try again, with the same probabilities for every roll. This just the *Phew!* game from the earlier example with different values of p , q , and r depending on what that first roll is.

Let S_1 be the sum of the dice on the first roll, and let W be the event that you win the game. We want to compute $\mathbb{E}(W)$, the probability of winning. To apply the Mighty Conditioning Identity, we consider all the possible first rolls to get hypothetical answers $\mathbb{E}(W \mid S_1 = r)$. Notice how we are just computing the predicted answer to the same question, enhanced with the extra hypothetical knowledge. The Mighty Conditioning Identity then gives us

$$\mathbb{E}(W) = \sum_r \mathbf{K}_{S_1}(r) \mathbb{E}(W \mid S_1 = r).$$

To calculate this, need to compute both terms.

First, the Distribution of S_1 . Let D_A and D_B be the values of the two individual dice on the first roll; these are independent random variables, each with a DiscreteUniform $\langle[1..6]\rangle$ Distribution. Because $S_1 = D_A + D_B$, for $r \in [2..12]$, we have the logical equivalence that $S_1 = r$ if and only for some j , $D_A = j$ and $D_B = r - j$. The “for some” here is a logical-or, so this gives us the equality of events:

$$\{S_1 = r\} = \sum_{j=\max(r-6,1)}^{\min(r-1,6)} \{D_A = j \wedge D_B = r - j\},$$

so

$$\begin{aligned}
 K_{S_1}(r) &= \mathbb{E}(\{S_1 = r\}) \\
 &= \left(\sum_{j=\max(r-6,1)}^{\min(r-1,6)} \{D_A = j \wedge D_B = r - j\} \right) \\
 &= \sum_{j=\max(r-6,1)}^{\min(r-1,6)} \mathbb{E}(\{D_A = j \wedge D_B = r - j\}) \\
 &= \sum_{j=\max(r-6,1)}^{\min(r-1,6)} \mathbb{E}(\{D_A = j\}) \cdot \mathbb{E}(\{D_B = r - j\}) \\
 &= \frac{1}{36} \sum_{j=\max(r-6,1)}^{\min(r-1,6)} 1 \\
 &= \frac{\min(r-1,6) - \max(r-6,1) + 1}{36} \\
 &= \frac{6 - |r-7|}{36}.
 \end{aligned}$$

Hence,

$$D_{S_1}(\psi) = \sum_r \frac{6 - |r-7|}{36} \{r \in [2 \dots 12]\} \psi(r).$$

Second, we want to compute $\mathbb{E}(W \mid S_1 = r)$. If the first roll is 7 or 11, you win the game, so $\mathbb{E}(W \mid S_1 = 7) = \mathbb{E}(W \mid S_1 = 11) = 1$. If the first roll is 2, 3, or 12, you lose the game, so $\mathbb{E}(W \mid S_1 = 2) = \mathbb{E}(W \mid S_1 = 3) = \mathbb{E}(W \mid S_1 = 12) = 0$. If the first roll is a 4, 5, 6, 8, 9, or 10, each subsequent roll is like a *Phew* game where matching the first roll is “You win,” 7 or 11 is “You lose,” and any other roll is “Try again.” Using the result of the *Phew!* Example 24.24, for $r \in \{4, 5, 6, 8, 9, 10\}$

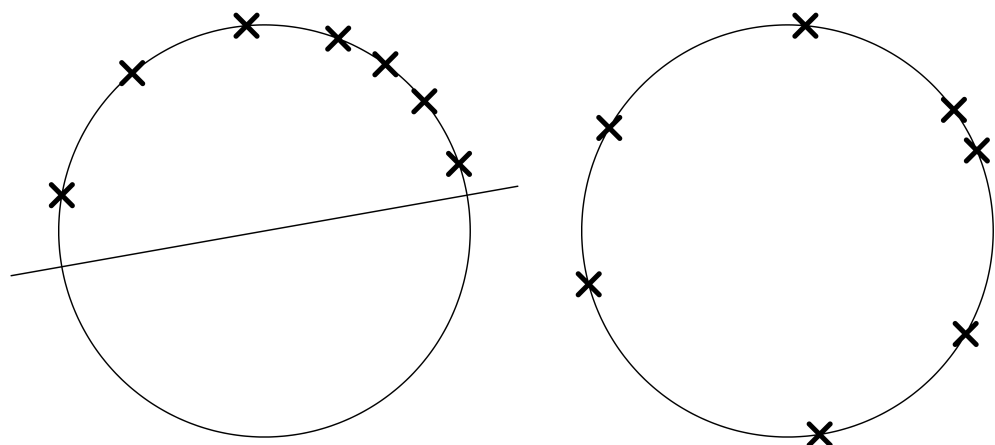
$$\mathbb{E}(W \mid S_1 = r) = \frac{K_{S_1}(r)}{K_{S_1}(r) + K_{S_1}(7) + K_{S_1}(11)}.$$

Putting this together, we have

$$\begin{aligned}
 \mathbb{E}(W) &= \sum_{k=2}^{12} \mathbb{E}(\{S_1 = k\}) \mathbb{E}(W \mid S_1 = k) \\
 &= K_{S_1}(7) + K_{S_1}(11) + \sum_{k \in \{4,5,6,8,9,10\}} \frac{K_{S_1}(k)^2}{K_{S_1}(k) + K_{S_1}(7) + K_{S_1}(11)}
 \end{aligned}$$

$$\approx 0.4486,$$

where you get the last step by plugging in the values of $K_{S_1}(k)$ above.



All points lie within a semi-circle

All points do not lie within a semi-circle

FIGURE 24.2. Two sample configurations from Example 24.27.

Example 24.27 Circle Points

You choose n random points in succession on the unit circle. (That is, on the perimeter not in the disk.) The position of each point is chosen uniformly over the circle, and the positions of all points are independent.

Find the probability that the n points *all* lie within a semi-circle, or in other words, that they all lie within a 180° arc on the circle. (See Figure 24.2 for examples.)

State the assumptions. The points are uniformly distributed over the circle and are all chosen independently. If we measure positions on the circle by an angle in the interval $[0, 2\pi)$, clockwise from east, say, then the points are independent $\text{Uniform}\langle 0, 2\pi \rangle$ random variables.

Define relevant random variables. For $i \in [1..n]$, let A_i be the *angular position* (from 0 to 2π), measured clockwise from east, of the i th chosen point.

For $i \in [1..n]$, let C_i be the *indicator that all n points lie within a semi-circle, clockwise from the i th chosen point*.

Let S be the *indicator that all n points lie within a semi-circle*.

State what you know. The A_i 's are independent $\text{Uniform}\langle 0, 2\pi \rangle$ rv's.

The key insight here is that whenever all the points lie in a semi-circle, we can take the semi-circle to be *anchored* at the counter-clockwise-most point. So we decompose the event S into the mutually exclusive events $C_1, \dots, C_n$ based on which point is the anchor. Consequently, we know that

$$S = C_1 + \dots + C_n.$$

State what you want to find. The desired probability is just $\mathbb{E}(S)$.

Find it. By symmetry among the points, all the points have the same distribution. By the Expectation Additivity Rule,

$$\mathbb{E}(S) = \mathbb{E}(C_1) + \dots + \mathbb{E}(C_n) = n \mathbb{E}(C_1).$$

We will compute the $\mathbb{E}(C_i)$ by conditioning.

Strategy: What would we like to know to make this question easier? In other words, for what random variable(s) X do we (a) know the Distribution of and (b) makes $\mathbb{E}(S \mid X = x)$ easy to compute?

Idea: Condition on one of the points, e.g., A_1 , and find the probability that the other points are in a 180 degree arc clockwise from A_1 . And indeed, to find $\mathbb{E}(C_i)$ it makes sense to condition on A_i .

Once we fix the point A_i , then C_i will be 1 when the other $n - 1$ points lie in the semi-circle clockwise from A_i . Each point independently lies in this semi-circle with probability $\frac{1}{2}$, so $\mathbb{E}(C_i \mid A_i = a) = \frac{1}{2^{n-1}}$. Notice that this does not depend on the value of A_i . So, because A_i has a Uniform $\langle 0, 2\pi \rangle$,

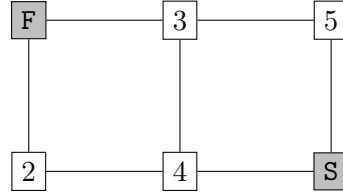
$$\begin{aligned} \mathbb{E}(C_i) &= D_{A_i}(\langle a \rangle \mapsto \mathbb{E}(C_i \mid A_i = a)) \\ &= \int_a da \frac{1}{2\pi} \{0 \leq a < 2\pi\} \mathbb{E}(C_i \mid A_i = a) \\ &= \int_{a=0}^{2\pi} da \frac{1}{2\pi} \frac{1}{2^{n-1}} \\ &= \frac{1}{2^{n-1}}. \end{aligned}$$

Hence,

$$\mathbb{E}(S) = \frac{n}{2^{n-1}}.$$

Example 24.28 Rat in a Maze

This graph represents a maze. Each node is a room, and each edge represents a door between two rooms.



A rat is placed into this maze. If the rat is in one of the other four rooms, it chooses one of the doors at random with equal probability. The rat stays put once it reaches either the food (F) or the shock (S). Given that the rat starts in state s , what is the probability that the rat gets food.

Let $R_0, R_1, R_2, \dots$ be the rooms that the rat visits with R_n the room after n steps. We label F as state 1 and S as state 6, and assume that once the rat reaches 1 or 6 it stays there.

Let F be the event that the rat reaches food. We want to find $f(r) = \mathbb{E}(F \mid R_0 = r)$, the probability of reaching food given that it starts in room r . Note that this is a Dirichlet problem as described in Section 7.3, and that dialogue is worth reviewing.

We know that $f(1) = 1$ and $f(6) = 0$ because if the rat starts in state 6 it stays there and never gets the food, but if it starts in 1 it stays there and gets the food. So we need to find $f(2), f(3), f(4)$, and $f(5)$.

Apply the Mighty Conditioning Identity by *conditioning on the rats first move, R_1* .

$$\begin{aligned} \mathbb{E}(F \mid R_0 = r) &= \sum_{r'} K_{R_1|R_0}(r' \mid r) \mathbb{E}(F \mid R_1 = r' \wedge R_0 = r) \\ &= \sum_{r'} K_{R_1|R_0}(r' \mid r) \mathbb{E}(F \mid R_0 = r'), \end{aligned}$$

so

$$f(r) = \sum_{r'} K_{R_1|R_0}(r' \mid r) f(r').$$

We know from the graph that

$$\begin{aligned} K_{R_1|R_0}(r' | 2) &= \frac{1}{2} \{r' = 1\} + \frac{1}{2} \{r' = 4\} \\ K_{R_1|R_0}(r' | 3) &= \frac{1}{3} \{r' = 1\} + \frac{1}{3} \{r' = 4\} + \frac{1}{3} \{r' = 5\} \\ K_{R_1|R_0}(r' | 4) &= \frac{1}{3} \{r' = 2\} + \frac{1}{3} \{r' = 3\} + \frac{1}{3} \{r' = 6\} \\ K_{R_1|R_0}(r' | 5) &= \frac{1}{2} \{r' = 3\} + \frac{1}{2} \{r' = 6\}. \end{aligned}$$

Putting these together, we get the system of four equations in four unknowns:

$$f(2) = \frac{1}{2} 1 + \frac{1}{2} f(4) \tag{24.54}$$

$$f(3) = \frac{1}{3} 1 + \frac{1}{3} f(4) + \frac{1}{3} f(5) \tag{24.55}$$

$$f(4) = \frac{1}{3} f(2) + \frac{1}{3} f(3) \tag{24.56}$$

$$f(5) = \frac{1}{2} f(3), \tag{24.57}$$

because $f(1) = 1$ and $f(6) = 0$. Solving this system of equations gives us $f(1) = 1, f(2) = 5/7, f(3) = 4/7, f(4) = 3/7, f(5) = 2/7, f(6) = 0$.

Transforming with Statistics

25

Chapter

Contents

25.1 Special Case: Projections	1015
25.2 Technique: Composition	1018
25.3 Technique: Determining Functions	1024
25.4 Technique: Change of Variables	1033

Key Take Aways

If X is a random variable and ψ is a compatible statistic, then the transform of X by ψ is the random variable $Y = \varphi(X)$.

To find the Distribution of Y , we consider all values of X and transform them by ψ , which yields all possible values of Y . For each possible value of Y , we get its weight by **adding up the weights of all values of X that return the same output from the statistic**.

For example, when Y is discrete and X has a simple Kernel:

$$\begin{aligned} D_Y(\psi) &= \sum_u p_X(u) \psi(\varphi(u)) + \int_u du f_X(u) \psi(\varphi(u)) \\ &= \sum_v \left(\sum_u p_X(u) \{\varphi(u) = v\} + \sum_v \int_u du f_X(u) \{\varphi(u) = v\} \right) \psi(v) \\ K_Y(v) &= \sum_u p_X(u) \{\varphi(u) = v\} + \sum_v \int_u du f_X(u) \{\varphi(u) = v\}. \end{aligned}$$

This adds up the probability masses (finite or infinitesimal) of the values u that produce each value v of Y . Analogous expressions work for general Y .

We consider three techniques for finding the Kind/Kernel/Distribution of Y :

1. **Composition Method:** $D_Y(\varphi) = D_X(\varphi \circ \psi)$.

We *translate* questions about Y to questions about X .

2. **Determining Functions Method:** For a function derived from a Distribution that uniquely determines the Distribution, we find the derived function for Y from the derived function for X .

The prototype for this is the CDF, F_Y , from which we can get D_Y and K_Y .

3. **Change of Variables Method:** When X and Y are continuous, we can find K_Y by a direct change of variables.

In addition, we can often find the Distribution of Y by direct reasoning.

Transforming an FRP with a statistic creates a new, related FRP. The two are connected by a wire – through the statistic’s adapter – so that when the original FRP is activated, its value is transformed by the statistic and becomes the activated value of the new FRP. The same mechanism applies for general random variables.

Let X be a random variable and ψ a compatible statistic, and define $Y = \psi(X)$. The two random variables are related by this statistic. When X is activated with value x , Y is activated as well with value $\psi(x)$.

We transform a Kind with a statistic in two steps: we apply the statistic to the value at each branch and then we combine branches that map to the same output, adding their weights. In `frplib`, this looks like:

```
original = kind(X)
transformed = psi(original)
```

With the Kernel representation of an FRP, which is necessarily discrete, this combining of branches and adding of weights becomes

$$K_Y(v) = \sum_u K_X(u) \{\psi(u) = v\}. \quad (*)$$

The indicator here is 1 for every value u that ψ maps to the value v from the left-hand side and is 0 otherwise. So the sum, simply adds up the corresponding weights.

The same steps apply for general random variables. When X is discrete, equation (*) holds as is. When X has continuous values, we use the same procedure and get an analogous picture

$$K_Y(v) = \sum_u K_X(u) \{\psi(u) = v\} + \int_u du K_X(u) \{\psi(u) \text{ near } v\}, \quad (**)$$

where the indicator “ $\{\psi(u) \text{ near } v\}$ ” captures an adjustment we need to make near continuous values, as we will see.

In this section, we look closely at the operation of transforming a Distribution with a statistic. We develop three basic techniques for calculating these transformations in practice, and in the process, we get further insight into how our procedure works even with continuous values.

25.1 Special Case: Projections

To get a feel for the general techniques, we start with a familiar and commonly used class of statistics – projections. A projection statistic maps a tuple of dimension d to a tuple consisting of $d' < d$ components of its input. This includes extracting single components, as with proj_1 or proj_2 , or extracting sub-tuples, as with $\text{proj}_{1:4}$ and $\text{proj}_{1,3,5}$.

For thinking about this transformation, it helps to think about Kind trees explicitly. For example, if we have Kind with values of the form $\langle a, b, c \rangle$ that we are transforming with proj_3 , then which branches do we combine? All branches that have an equal *third component*. Therefore, the weight in the transformed Kind on a value like $\langle 4 \rangle$ is the sum of the weights over all values $\langle a, b, 4 \rangle$. This works in general.

The easiest way to see this is to start with the Distribution. Let X be a random variable with dimension $d > 1$ and let proj_ι stand for a specific projection statistic, where $\iota = i_1, i_2, \dots, i_{d'}$ where $1 \leq i_1 < i_2 < \dots < i_{d'} \leq d$. This could be proj_1 , $\text{proj}_{1,3}$, $\text{proj}_3 : d$, or any other projection for suitable choice of ι . Let $Y = \text{proj}_\iota(X)$.

So we have

$$\begin{aligned} D_Y(\psi) &= \mathbb{E}(\psi(Y)) \\ &= \mathbb{E}(\psi(\text{proj}_\iota(X))) \\ &= \mathbb{E}((\psi \circ \text{proj}_\iota)(X)) \\ &= D_X(\psi \circ \text{proj}_\iota). \end{aligned} \tag{25.1}$$

This *translates* a question about Y , ψ , into a question about X , $\psi \circ \text{proj}_\iota$. Specifically, any question we have about components $X_{i_1}, X_{i_2}, \dots, X_{i_{d'}}$ can be expressed as a question about X that merely focuses on those components. For example, if proj_ι is proj_1 , then the question “Is Y bigger than 10?” with corresponding statistic $\blacksquare > 10$ translates into the question “Is the first component of X bigger than 10?” with corresponding statistic $\blacksquare_1 > 10$.

To understand the meaning of equation (25.1), suppose that X has a simple Kernel K_X and consider the case where $\text{proj}_\iota = \text{proj}_i$ (i.e., $\iota = i$) for some $i \in [1 \dots d]$. For a d -tuple u , let $u_{-i} = \langle u_1, \dots, u_{i-1}, u_{i+1}, \dots, u_d \rangle$ be the $(d-1)$ -tuple excluding the i th component of u . (If $i = 1$ or $i = d$, the segments on the left or right ends of this tuple are empty.) For clarity, we will exclude the range_d and range_c limits, treating K_X as its discrete part p_X in sums and as its continuous part f_X in integrals.

Then, equation (25.1) becomes

$$\begin{aligned} D_Y(\psi) &= D_X(\psi \circ \text{proj}_\iota) \\ &= \sum_u K_X(u) \psi(\text{proj}_i(u)) + \int_u du K_X(u) \psi(\text{proj}_i(u)) \\ &= \sum_u K_X(u) \psi(u_i) + \int_u du K_X(u) \psi(u_i) \\ &= \sum_{u_i} \left(\sum_{u_{-i}} K_X(u) \right) \psi(u_i) + \int_{u_i} du_i \left(\int_{u_{-i}} du_{-i} K_X(u) \right) \psi(u_i). \end{aligned}$$

And here we see that we are just adding up weights over all values of X whose i th component is u_i , and then using those weights to average ψ . It might help to write this explicitly in terms of the components:

$$\begin{aligned} D_Y(\psi) &= \sum_{u_i} \left(\sum_{u_1} \cdots \sum_{u_{i-1}} \sum_{u_{i+1}} \cdots \sum_{u_d} K_X(u_1, \dots, u_d) \right) \psi(u_i) \\ &\quad + \int_{u_i} du_i \left(\int_{u_1} du_1 \cdots \int_{u_{i-1}} du_{i-1} \int_{u_{i+1}} du_{i+1} \cdots \int_{u_d} du_d K_X(u_1, \dots, u_d) \right) \psi(u_i) \end{aligned}$$

Do not let the many sums and integrals intimidate you here; within the parentheses, we are simply looping over all components except the i th. From this, we can extract the Kernel K_Y :

$$\begin{aligned} K_Y(w) &= \sum_{u_1} \cdots \sum_{u_{i-1}} \sum_{u_{i+1}} \cdots \sum_{u_d} K_X(u_1, \dots, u_{i-1}, w, u_{i+1}, \dots, u_d) \\ &\quad + \int_{u_1} du_1 \cdots \int_{u_{i-1}} du_{i-1} \int_{u_{i+1}} du_{i+1} \cdots \int_{u_d} du_d K_X(u_1, \dots, u_{i-1}, w, u_{i+1}, \dots, u_d). \end{aligned}$$

We get the weight at w by adding up the weights over all values for which the i th component equals w .

For the general case, for a d -tuple u , let $u_\iota = \text{proj}_\iota(u)$ and $u_{-\iota} = \overline{\text{proj}_\iota}$ and

for d' -tuple w , let $u[w]$ be the tuple u with the $i_1, \dots, i_{d'}$ components replaced by $w_1, \dots, w_{d'}$. We get the same basic result

If X is a d -dimensional random variable with a simple Kernel and $\iota = i_1, i_2, \dots, i_{d'}$ where $1 \leq i_1 < i_2 < \dots < i_{d'} \leq d$, then the transformation of X by the projection statistic proj_ι is the random variable $Y = \text{proj}_\iota(X)$ with Distribution and Kernel:

$$D_Y(\psi) = \sum_{u_\iota} \left(\sum_{u_{-\iota}} K_X(u) \right) \psi(u_\iota) + \int_{u_\iota} du_\iota \left(\int_{u_{-\iota}} du_{-\iota} K_X(u) \right) \psi(u_\iota) \quad (25.2)$$

$$K_Y(w) = \sum_{u_{-\iota}} K_X(u[w]) + \int_{u_{-\iota}} du_{-\iota} K_X(u[w]). \quad (25.3)$$

Example 25.1. Random variable $\langle X, Y \rangle$ represents a point chosen uniformly from inside the unit circle $\mathcal{C} = \{ \langle u, v \rangle \mid u^2 + v^2 \leq 1 \}$. Find D_Y and K_Y .

We can see directly that $Y = \text{proj}_2(\langle X, Y \rangle)$, and both random variables are continuous. So by equations (25.2) and (25.3):

$$\begin{aligned} D_Y(\psi) &= \int_y dy \left(\int_x dx K_{XY}(x, y) \right) \psi(y) \\ &= \int_{y=-1}^1 dy \left(\int_{x=-\sqrt{1-y^2}}^{\sqrt{1-y^2}} dx \frac{1}{2\pi} \right) \psi(y) \\ &= \int_{y=-1}^1 dy \frac{1}{\pi} \sqrt{1-y^2} \psi(y) \\ &= \int_y dy \frac{1}{\pi} \sqrt{1-y^2} \{ -1 \leq y \leq 1 \} \psi(y), \end{aligned}$$

so

$$K_Y(y) = \frac{1}{\pi} \sqrt{1-y^2} \{ -1 \leq y \leq 1 \}.$$

The limits of the integrals because if we fix a value of y , the only way $x \in \mathcal{C}$ is if $x^2 + y^2 \leq 1$ which implies $|x| \leq \sqrt{1-y^2}$.

Example 25.2. We have scalar random variables L , W , and H , representing

the length, width, and height of a random region. We have

$$\begin{aligned} K_L(\ell) &= \frac{1}{4} \{\ell = 10\} + \frac{3}{4} \{\ell = 20\} \\ K_{W|L}(w | \ell) &= \sum_{d=-1}^1 \frac{1}{3} \{w = \ell + 5d\} \\ K_{H|L,W}(h | \ell, w) &= \frac{1}{2} \{h = \ell \wedge \ell = w\} + \frac{1}{2} \{h = 50\}. \end{aligned}$$

Find the Distribution or Kernel of A , the width and height of the vertical face of the rectangular region at the right-hand end.

Let $B = \langle L, W, H \rangle$ be the join random variable that describes the region. Then, $A = \text{proj}_{23}(B)$. By equation (25.3), we have

$$\begin{aligned} K_A(w, h) &= \sum_{\ell} K_B(\ell, w, h) \\ &= \sum_{\ell} K_L(\ell) K_{W|L}(w | \ell) K_{H|L,W}(h | \ell, w), \end{aligned}$$

because $K_B = K_L \triangleright K_{W|L} \triangleright K_{H|L,W} = K_{LWH}$. For instance,

$$\begin{aligned} K_A(10, 10) &= K_{LWH}(10, 10, 10) + K_{LWH}(20, 10, 10) \\ &= K_{LWH}(10, 10, 10) + 0 \\ &= \frac{1}{4} \cdot \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{24} \\ K_A(15, 50) &= K_{LWH}(10, 15, 50) + K_{LWH}(20, 15, 50) \\ &= \frac{1}{4} \cdot \frac{1}{3} \cdot \frac{1}{2} + \frac{3}{4} \cdot \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{6} \\ K_A(25, 50) &= K_{LWH}(10, 25, 50) + K_{LWH}(20, 25, 50) \\ &= 0 + \frac{3}{4} \cdot \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{8}, \end{aligned}$$

and similarly for the other possible values $\langle 5, 50 \rangle$, $\langle 10, 50 \rangle$, $\langle 20, 20 \rangle$, and $\langle 20, 50 \rangle$.

25.2 Technique: Composition

Again, we let X be an arbitrary random variable (of any dimension) and ψ an arbitrary compatible statistic. Define $Y = \psi(X)$ to be the transform of X , with dimension equal to the dimension of ψ .

The **composition method** computes the distribution of Y by **translating**

questions about Y into questions about X and using D_X to answer those questions. A statistic φ that is compatible with Y represents a question we might ask about Y , with answer $\varphi(Y)$. But by definition, $Y = \psi(X)$, so

$$\varphi(Y) = \varphi(\psi(X)) = (\varphi \circ \psi)(X).$$

The composition $\varphi \circ \psi$ (read “ φ after ψ ”) is a statistic that represents a question we can ask about X . We have translated the question φ about Y to a question $\varphi \circ \psi$ about X . Taking expectations gives us

$$\mathbb{E}(\varphi(Y)) = \mathbb{E}(\varphi(\psi(X))),$$

so we can use the Distribution of X to find what we need.

The Composition Method. If $Y = \psi(X)$, then

$$D_Y(\varphi) = D_X(\varphi \circ \psi). \quad (25.4)$$

If we know the Distribution of X , we can find the Distribution of Y . This method works with any distribution regardless of the dimension and even if we have updated by constraining with a conditional.

To express the Distribution operator itself, equation (25.4) gives us everything we need. But we can also use this equation to find the Kernel K_Y . The trick is to do some rearrangement or change-of-variables in the resulting sum or integral so that the argument to φ is a single variable. We can then view $D_Y(\varphi)$ as a weighted average over the values of φ , and the weights give K_Y . Even if we do not need the Kernel per se, this trick can sometimes also make the calculations of the expectations cleaner.

Example 25.3 Discrete Shift Let B be a random variable with Distribution

$$D_B(\psi) = \sum_k \binom{n}{k} p^k (1-p)^{n-k} \{k \in [0 \dots n]\} \psi(k)$$

for constants: even integer $n > 0$ and $0 \leq p \leq 1$. Define $S = \zeta(B)$, where $\zeta(x) = x - n/2$.

$$D_S(\varphi) = D_B(\varphi \circ \zeta)$$

$$\begin{aligned}
&= \sum_k \binom{n}{k} p^k (1-p)^{n-k} \{k \in [0 \dots n]\} \varphi(\zeta(k)) \\
&= \sum_{k=0}^n \binom{n}{k} p^k (1-p)^{n-k} \varphi(k - n/2) \\
&= \sum_{j=-n/2}^{n/2} \binom{n}{j+n/2} p^{j+n/2} (1-p)^{n/2-j} \\
&= \sum_j \binom{n}{j+n/2} p^{j+n/2} (1-p)^{n/2-j} \{j \in [-n/2 \dots n/2]\} \varphi(j).
\end{aligned}$$

To see the transition from the third to the fourth equalities here, notice that the if you expand the sums, you get the same terms in both. Thus,

$$\mathsf{K}_S(j) = \binom{n}{j+n/2} p^{j+n/2} (1-p)^{n/2-j} \{j \in [-n/2 \dots n/2]\}.$$

Example 25.4 Discrete Sum Let A be a random variable with Kind `uniform(1, 2, ..., 6) ** 2`, a $\text{Uniform}([1 \dots 6]^2)$ Distribution. Define $S = \text{Sum}(A)$. Find the distribution of S by the composition method.

$$\begin{aligned}
\mathsf{D}_S(\varphi) &= \mathsf{D}_A(\varphi \circ \text{Sum}) \\
&= \sum_i \sum_j \mathsf{K}_A(i, j) \varphi(\text{Sum}(i, j)) \\
&= \sum_{i=1}^6 \sum_{j=1}^6 \frac{1}{36} \varphi(i+j) \\
&= \sum_{i=1}^6 \sum_{k=i+1}^{i+6} \frac{1}{36} \varphi(k) \\
&= \sum_{k=2}^{12} \sum_{i=\max(k-6, 1)}^{\min(k-1, 6)} \frac{1}{36} \varphi(k) \\
&= \sum_{k=2}^{12} \frac{\min(k-1, 13-k)}{36} \varphi(k).
\end{aligned}$$

From the second to the third equation, we simply sum over values $\langle i, k \rangle$ that are in direct correspondence with the original $\langle i, j \rangle$ pairs. From the third to the fourth equations, we sum over the same $\langle i, k \rangle$ pairs but first over the k 's. As a

byproduct, we get that

$$\mathsf{K}_S(k) = \frac{\min(k-1, 13-k)}{36} \{k \in [2 \dots 12]\}.$$

Example 25.5 As Simple As Needed

Let B be the random variable in Example 25.3 and consider a different statistic:

$M = \xi(B)$ where $\xi(y) = y(y-1)$.

$$\begin{aligned} \mathsf{D}_M(\varphi) &= \mathsf{D}_B(\varphi \circ \xi) \\ &= \sum_k \mathsf{K}_B(k) \varphi(\xi(k)) \\ &= \sum_k \binom{n}{k} p^k (1-p)^{n-k} \{k \in [0 \dots n]\} \varphi(\xi(k)) \\ &= \sum_k \binom{n}{k} p^k (1-p)^{n-k} \{k \in [0 \dots n]\} \varphi(k(k-1)). \end{aligned}$$

While we could rearrange this to put a single variable argument to φ , there is not much benefit from doing so since the possible values are a discrete set that is most easily described as $\{k(k-1) : k \in [0 \dots n]\}$. In most cases, some change of variables is useful, but in this case the Distribution is clear as is.

Example 25.6 Scaled Uniform

Let U be a random variable with kind $\mathsf{K}_U = [0\_1]$, and equivalently $\mathsf{D}_U(\psi) = \int_{t=0}^1 \psi(t) dt$. We will call this a Uniform $\langle 0, 1 \rangle$ distribution.

Define $W := \psi(U)$, where $\psi(u) = a + (b-a)u$ for constants $b > a$. We want to find the distribution of W using the composition method.

$$\begin{aligned} \mathsf{D}_W(\varphi) &= \mathsf{D}_U(\varphi \circ \psi) \\ &= \int_u du \mathsf{K}_U(u) \varphi(\psi(u)) \\ &= \int_u du \{0 \leq u \leq 1\} \varphi(a + (b-a)u) \\ &= \int_{u=0}^1 du \varphi(a + (b-a)u) \\ &= \int_{u=0}^1 du \frac{b-a}{b-a} \varphi(a + (b-a)u) \end{aligned}$$

$$\begin{aligned}
&= \int_{w=a}^b dw \frac{1}{b-a} \varphi(w) && \text{(see note below)} \\
&= \int_w dw \frac{1}{b-a} \{a \leq w \leq b\} \varphi(w).
\end{aligned}$$

In the penultimate line, we do a change of variables $w = a + (b-a)u$. This gives us a term $\varphi(w)$, making the rest of the integrand the kernel. To do the change of variables, we write $dw = (b-a)du$. Having multiplied by $1 = (b-a)/(b-a)$, we have a factor $(b-a)du$ which becomes dw . When $u = 0$, $w = a$ and when $u = 1$, $w = b$; this gives us the limits of integration after the change of variables.

So we have expressed $D_w(\varphi)$ in terms of φ with a single variable argument, allowing us to read off the kernel K_w directly.

$$K_w(t) = \frac{1}{b-a} \{a \leq w \leq b\}.$$

Notice that the Kernel of W (here a probability density function as W is continuous) gives equal weight to every value in the interval $[a, b]$. The density is uniform and is in canonical form. We say that W has a Uniform $\langle a, b \rangle$ distribution.

It's worth asking what happens if $b < a$. The analysis carries over as before, but in the final result, the limits of integration go from the larger value a to the smaller value b , and $1/(b-a)$ is negative. Switching the order of integration multiplies the integral by -1, which we can apply to the $1/(b-a)$ factor. This yields

$$D_w(\varphi) = \int_w dw \frac{1}{a-b} \{b \leq w \leq a\} \varphi(w).$$

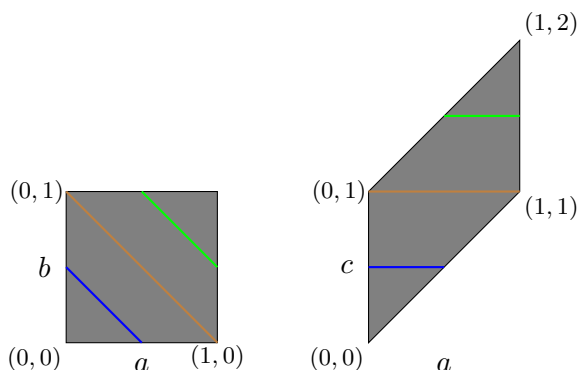
Example 25.7 Uniform Sum

Let W be a random variable with kind Uniform $\langle 0, 1 \rangle \star \star 2$, which has kernel $K_w(u, v) = [0, 1](u) [0, 1](v)$. Define $Z := \psi(W)$ where $\psi(a, b) = a + b$.

We want to find the distribution of Z using the composition method. First, we translate a question about Z to a question about W :

$$\begin{aligned}
D_Z(\varphi) &= D_w(\varphi \circ \psi) \\
&= \int_a da \int_b db K_w(a, b) \varphi(\psi(a, b)) \\
&= \int_{a=0}^1 da \int_{b=0}^1 db \varphi(a + b).
\end{aligned}$$

Now, we will change variables so that the argument to φ is a single variable. Specifically, we want a to remain unchanged and $c = a + b$. Then, $db\,da = d(c - a)\,da = dc\,da$.



So,

$$\begin{aligned}
 D_Z(\varphi) &= \int_{a=0}^1 da \int_{c=a}^{1+a} dc \, \varphi(c) \\
 &= \int_{c=0}^2 dc \int_{a=\max(c-1,0)}^{\min(c+1,1)} da \, \varphi(c) \\
 &= \int_{c=0}^2 dc \left(\int_{a=\max(c-1,0)}^{\min(c+1,1)} da \right) \varphi(c) \\
 &= \int_{c=0}^2 dc [\min(c+1,1) - \max(c-1,0)] \varphi(c) \\
 &= \int_c dc [c\{0 \leq c < 1\} + (2-c)\{1 \leq c \leq 2\}] \varphi(c),
 \end{aligned}$$

from which we can see $K_Z(c) = c\{0 \leq c < 1\} + (2-c)\{1 \leq c \leq 2\}$.

The second equation here is the trickiest part. What is happening is that we simply switch the order in which we sum over the region of integration. In the first equation, we sum up first along values of the $a + b$ for a fixed a , and then sum up these sums across a . Reversing this in the second equation, we first sum up across horizontal slices (a) and then sum those sums across values of the sum.

The figure above shows the change of variables and illustrates what a line of constant c looks like in both the original coordinates and the new coordinates. The limits of integration in the second equation above are evident in the right hand panel.

Example 25.8 Radial Distance

$\langle X, Y \rangle$ is a random point chosen uniformly in the unit disk $\mathcal{D} = \{ \langle x, y \rangle : x^2 + y^2 \leq 1 \}$. Define $R := \psi(X, Y)$ where $\psi(x, y) = \sqrt{x^2 + y^2}$. We want to find the distribution of R using the composition method.

Since R represents the radial distance from the random point to the origin, it will be useful to change from Cartesian to polar coordinates. See Example 20.5.

We first translate a question about R to a question about $\langle X, Y \rangle$, and then we change variables to express the φ term with a single variable argument $\varphi(r)$.

$$\begin{aligned}
 D_R(\varphi) &= D_{XY}(\varphi \circ \psi) \\
 &= \iint dx \, dy \, K_{XY}(x, y) \, \varphi(\psi(x, y)) \\
 &= \iint dx \, dy \, \frac{1}{\pi} \mathcal{D}(x, y) \, \varphi(\psi(x, y)) \\
 &= \iint_{\mathcal{D}} dx \, dy \, \frac{1}{\pi} \varphi(\sqrt{x^2 + y^2}) \\
 &= \int_{r=0}^1 dr \, r \int_{\theta=0}^{2\pi} d\theta \, \frac{1}{\pi} \varphi(r) \quad (\text{change to polar coordinates}) \\
 &= \int_{r=0}^1 dr \, \left[\int_{\theta=0}^{2\pi} d\theta \, \frac{1}{\pi} \right] r \varphi(r) \\
 &= \int_{r=0}^1 dr \, \frac{2\pi}{\pi} r \varphi(r) \\
 &= \int_r dr \, 2r \, \{0 \leq r \leq 1\} \varphi(r).
 \end{aligned}$$

The composition method is a general and powerful approach for analyzing transformed random variables in any dimension. With a little practice, it also becomes fast and efficient.

25.3 Technique: Determining Functions

Computing the distribution by finding the expectation for every compatible statistic would be a daunting task. Fortunately, we do not have to do that. As we saw in Chapter 6 (see Definition 39 and surrounding text), the Distribution of a random variable Y is *determined* by a sufficiently rich class of statistics. We call such a collection of statistics a **determining class**. If $\mathcal{S}$ is a determining class, $D_Y(\psi) = D_X(\psi)$ for *all* $\psi \in \mathcal{S}$, then $D_Y = D_X$. To determine the distribution of Y uniquely, we thus need to find $D_Y(\psi)$ for ψ in a determining class.

For scalar random variables, there are two commonly used determining classes

1. $\{\varphi_t \mid -\infty < t < \infty\}$ where $\varphi_t = (\{\blacksquare \leq t\})$, and
2. $\{\bar{\varphi}_t \mid -\infty < t < \infty\}$ where $\bar{\varphi}_t = (\{\blacksquare > t\})$.

We have

$$D_Y(\varphi_t) = \mathbb{E}(\{Y \leq t\})$$

$$D_Y(\bar{\varphi}_t) = \mathbb{E}(\{Y > t\}).$$

So if we know the probability that $Y \leq t$ for *every* t , we can find the Distribution of Y . And similarly if we know the probability that $Y > t$ for every t .

Viewing these expectations *as a function of t* gives us

$$F_Y(t) = \mathbb{E}(\{Y \leq t\}) \tag{25.5}$$

$$S_Y(t) = \mathbb{E}(\{Y > t\}). \tag{25.6}$$

We call a function derived from a Distribution whose argument indexes a determining class, and thus determines the distribution, a **determining function**. Both F_Y and S_Y are determining functions.

Observe that $\{Y \leq t\}$ and $\{Y > t\}$ are complements, so

$$\{Y > t\} = 1 - \{Y \leq t\},$$

so

$$S_Y(t) = 1 - F_Y(t).$$

The function F_Y was introduced in Example 19.9 and is called the **Cumulative Distribution Function**, or **CDF** for short. The complementary function S_Y is called the **Survival Distribution Function**, or **SDF** for short.

So we have

$$F_Y(t) = D_Y(\blacksquare \leq t) \tag{25.7}$$

$$S_Y(t) = D_Y(\blacksquare > t). \tag{25.8}$$

And if Y has a simple Kernel, we can express these functions as

$$F_Y(t) = \sum_v p_Y(v) \{v \leq t\} + \int_v dv f_Y(v) \{v \leq t\} \quad (25.9)$$

$$S_Y(t) = \sum_v p_Y(v) \{v > t\} + \int_v dv f_Y(v) \{v > t\}, \quad (25.10)$$

where p_Y and f_Y are the discrete (mass) and continuous (density) parts of K_Y .

Puzzle 173. If T is a continuous random variable with density function $f_T(u) = K_T(u)$, sketch a picture of f_T and fill in the area under the curve corresponding to the probability $F_T(u)$.

See Figure 25.1 below *after* you give this a try.

The CDF is a monotone non-decreasing function that goes from 0 at $-\infty$ to 1 at ∞ ; the SDF is a monotone non-increasing function that goes from 1 at $-\infty$ to 0 at ∞ . If Y is continuous, then both functions are differentiable. If Y is discrete, then both functions are constant between values of Y and *jump* at values of Y .

Example 25.9. Let Y have Kernel

$$K_Y(t) = \frac{1}{6} \{t = -4\} + \frac{1}{2} \{t = 1\} + \frac{1}{3} \{t = 10\}.$$

Find the CDF and SDF of Y .

We have

$$\begin{aligned} F_Y(t) &= \mathbb{E}(\{Y \leq t\}) \\ &= \sum_s K_Y(s) \{s \leq t\} \\ &= \sum_s \left(\frac{1}{6} \{s = -4\} + \frac{1}{2} \{s = 1\} + \frac{1}{3} \{s = 10\} \right) \{s \leq t\} \\ &= \frac{1}{6} \{-4 \leq t\} + \frac{1}{2} \{1 \leq t\} + \frac{1}{3} \{10 \leq t\}, \end{aligned}$$

and $S_Y(t) = 1 - F_Y(t)$.

Example 25.10. Let T have Kernel

$$K_T(u) = \frac{1}{b-a} \{a \leq u \leq b\},$$

for $a < b$. Find the CDF of T .

We have

$$\begin{aligned} F_T(t) &= \int_{-\infty}^t du \, K_T(u) \{u \leq t\} \\ &= \int_{-\infty}^t du \, K_T(u) \\ &= \int_{-\infty}^t du \, \frac{1}{b-a} \{a \leq u \leq b\} \\ &= \int_{u=a}^{\min(t,b)} du \, \frac{1}{b-a} \\ &= \frac{t-a}{b-a} \{a \leq t \leq b\} + \{t > b\}. \end{aligned}$$

Here, the penultimate integral is 0 if $t \leq a$ and is 1 if $t \geq b$.

As we saw in Example 19.9, we can actually recover the Distribution and Kernel of Y from its CDF, and the same goes for its SDF because we can get the CDF from it. If v is a discrete value of Y ,

$$K_Y(v) = F_Y(v) - F_Y(v-), \quad (25.11)$$

where $F_Y(v-)$ denotes the biggest value of $F_Y(t)$ for all t *strictly less than* v . We can write $\Delta F_Y(v) = F_Y(v) - F_Y(v-)$ to denote this difference. Then $\Delta F_Y(v)$ is the amount by which F_Y *jumps* at v . Similarly, for a continuous value v of Y :

$$K_Y(v) = F'_Y(v), \quad (25.12)$$

the *derivative* of the CDF at v , the rate at which probability mass is accumulating near v .

Puzzle 174. Using the previous two equations, express the Kernel K_Y in terms of the SDF at discrete and continuous points.

The name *Cumulative* in CDF reflects these equations. $F_Y(t)$ *accumulates* all the probability up to and including the value t . The name *Survival* in CDF reflects the

event $\{Y > t\}$ if Y were measuring the lifetime of some object; it is the event that Y *survives* at least to time t .

While the CDF and SDF are the most commonly used *determining functions*, they are not the only useful ones as we will see. The **Determining Functions Method** for finding the distribution of a transform is to use the relationship between the original and transformed random variables to compute a determining function for the transformed variable. Sometimes rather than computing the Kernel or Distribution directly, it is easier to compute an aggregate quantity like the CDF or SDF and use that to find the distribution.

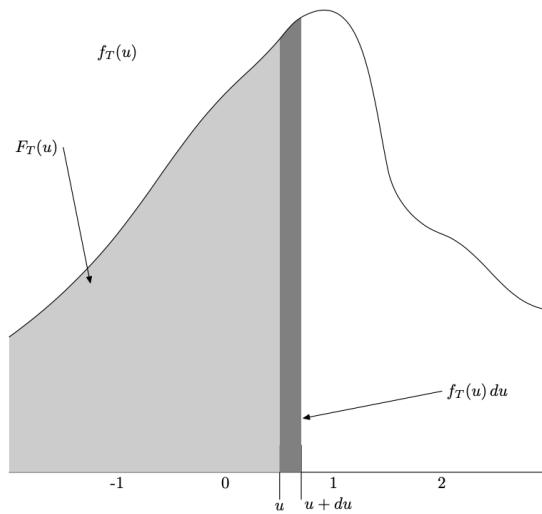


FIGURE 25.1. The Kernel (density function) of a continuous random variable T . The accumulated probability $F_T(u)$ is shaded light gray under the curve. The incremental, infinitesimal probability mass near u is shown in the dark gray slice between u and $u + du$.

Example 25.11 Smallest Waiting Time

Alice and Bob each flip a balanced coin until they first get a heads. Find the distribution of the number of flips it takes until the both of them have gotten heads.

Let A and B be random variables whose respective values are the number of flips Alice and Bob make up to and including the first heads. We assume that the flips that Alice and Bob make are independent of each other, i.e., $\langle A, B \rangle$ is an independent join.

We have already seen that

$$K_{AB}(m, n) = 2^{-n} m \in [1 \dots] 2^{-m} \{n \in [1 \dots]\}.$$

We are interested in the transformed random variable $M := \max(A, B)$. Finding the CDF of M is relatively easy because we have the logical equivalence: $\{M \leq t\} = \{A \leq t \wedge B \leq t\}$. So,

$$\begin{aligned} F_M(t) &= \mathbb{E}(\{M \leq t\}) \\ &= \mathbb{E}(\{A \leq t \wedge B \leq t\}) \\ &= \sum_{m,n} K_{AB}(m, n) \{m \leq t\} \{n \leq t\} \\ &= \sum_{m=1}^{\lfloor t \rfloor} \sum_{n=1}^{\lfloor t \rfloor} 2^{-m} 2^{-n} \\ &= \left(\sum_{m=1}^{\lfloor t \rfloor} 2^{-m} \right)^2 \\ &= \left(1 - 2^{-\lfloor t \rfloor} \right)^2. \end{aligned}$$

Notice that since we have an independent join, $\mathbb{E}(\{A \leq t \wedge B \leq t\}) = \mathbb{E}(\{A \leq t\}) \cdot \mathbb{E}(\{B \leq t\})$.

Example 25.12 Scaled Uniform cont'd

Let U and W be the random variables from Example 25.6. We want to find the distribution of W , and we start by finding its CDF, $F_W(t) = \mathbb{E}(\{W \leq t\})$.

To do this, let's start with the logic and move from what we want toward what we want to find. Remember that logically equivalent conditions give equal events.

$$\begin{aligned} \{W \leq t\} &= \{a + (b - a)U \leq t\} \\ &= \{(b - a)U \leq t - a\} \\ &= \left\{ U \leq \frac{t - a}{b - a} \right\}. \end{aligned}$$

(Be careful here if $b < a$.) Now we can take expectations.

$$\begin{aligned} F_w(t) &= \mathbb{E}(\{W \leq t\}) = \mathbb{E}\left(\left\{U \leq \frac{t-a}{b-a}\right\}\right) \\ &= \frac{t-a}{b-a} \{a \leq t \leq b\} + \{t > b\}. \end{aligned}$$

To get the kernel, we need the density around each t :

$$f_w(t) = K_w(t) = F'_w(t) = \frac{1}{b-a} \{a \leq t \leq b\},$$

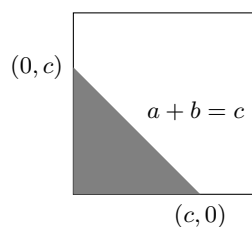
which is the same result we got earlier.

Example 25.13 Uniform Sum cont'd

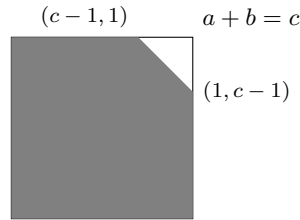
Continuing Example 25.7, we have W and Z , where W is a uniform random point in the unit square and $Z = \text{Sum}(W)$. We will find the distribution of Z using the CDF method.

$$\begin{aligned} F_Z(c) &= \mathbb{E}(\{Z \leq c\}) \\ &= \mathbb{E}(\{W_1 + W_2 \leq c\}) \\ &= \text{Area of the set } \{a + b \leq c \mid a, b \in [0, 1]\}, \end{aligned}$$

because W is uniform and the unit square has area 1. If $0 \leq c \leq 1$, the set of interest has area $\frac{c^2}{2}$.



If $1 \leq c \leq 2$, the area is $1 - \frac{(2-c)^2}{2}$.



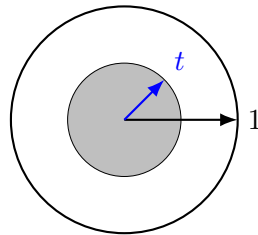
Taking derivatives $F'_z(c) = f_c(z) = K_c(z)$, and we get

$$D_z(\varphi) = \int_c dc [c \{0 \leq c < 1\} + (2-c)\{1 \leq c \leq 2\}] \varphi(c),$$

as before.

Example 25.14 Radial Distance cont'd

Following up on Example 25.8, we have $\langle X, Y \rangle$ a random point in disk $\mathcal{D}$, $R = \sqrt{X^2 + Y^2}$. We will use the CDF method to find the distribution of R .



Because $\langle X, Y \rangle$ is uniform in the unit disk, the probability that the point lies in the inner circle in the above picture is just the relative area of that circle inside the disk. Hence, we have

$$\begin{aligned} F_R(t) &= \mathbb{E}(\{R \leq t\}) \\ &= D_{XY} \left(\langle x, y \rangle \mapsto \left\{ 0 \leq \sqrt{x^2 + y^2} \leq t \right\} \right) \\ &= \frac{\pi t^2}{\pi} \{0 \leq t \leq 1\} + \{t > 1\} \\ &= t^2 \{0 \leq t \leq 1\} + \{t > 1\}. \end{aligned}$$

Taking derivatives

$$\begin{aligned} K_R(t) &= f_R(t) = F'_R(t) = 2t \{0 \leq t \leq 1\} \\ D_R(\psi) &= \int_r dr \ 2r \{0 \leq r \leq 1\} \psi(r), \end{aligned}$$

as before.

Example 25.15 A Different Determining Function

Let N, M be *independent* random variables with common Distribution

$$D(\psi) = \sum_k \binom{n}{k} p^k (1-p)^{n-k} \{k \in [0 \dots n]\} \psi(x),$$

where $n \in [1 \dots \infty)$ and $0 < p < 1$. We call this a Binomial $\langle n, p \rangle$ distribution; see Interlude C.

Let $S = N + M$. We want to find the Distribution of S , and we will use the determining function

$$G_S(z) = \mathbb{E}(z^S),$$

and similarly $G_N(z)$ and $G_M(z)$.

Note that

$$\begin{aligned} G_S(z) &= \mathbb{E}(z^S) \\ &= \mathbb{E}(z^{N+M}) \\ &= \mathbb{E}(z^N z^M) \\ &= \mathbb{E}(z^N) \mathbb{E}(z^M) \\ &= G_N(z) G_M(z) \\ &= G_N^2(z). \end{aligned}$$

The third equality follows because N and M are independent, and the last equality because they have the same Distribution.

See Interlude G for a demonstration that this is a determining function.

Now,

$$\begin{aligned}
 G_N(z) &= D_N(z) \\
 &= \sum_k \binom{n}{k} p^k (1-p)^{n-k} \{k \in [0 \dots n]\} z^k \\
 &= \sum_{k=0}^n \binom{n}{k} (pz)^k (1-p)^{n-k} \\
 &= (1-p+pz)^n,
 \end{aligned}$$

by the Binomial Theorem. (See Chapter 18 and Example 17.30 in Interlude F.)

Hence,

$$G_S(z) = (1-p+pz)^{2n}.$$

But this has the same form as G_N but with a different value of n . We can thus conclude that S has a Binomial($2n, p$) Distribution.

25.4 Technique: Change of Variables

When transforming *continuous* random variables, we have to account for the local stretch-and-shrink induced by the transformation. For continuous random variables (or for a region of continuous values for a general random variable), we can use the change of variables equation (20.7) or (20.6) to directly express the probability density function (kernel) of the transformed random variable in terms of the density function of the original random variable.

We consider transformations over a region $\mathcal{R}$ of *continuous* values for the original random variable X , with ψ a compatible statistic. This method requires two additional constraints on the transforming statistic ψ . First, we require that ψ be *invertible* on $\mathcal{R}$ with inverse¹ ψ^{-1} . Second, ψ must be *differentiable* on $\mathcal{R}$, meaning that the derivative (matrix) ψ' exists. It follows from these constraints that ψ' must be *non-singular* on $\mathcal{R}$, meaning that its Jacobian is non-zero at every point in $\mathcal{R}$. For scalar statistics, this means that $|\psi'| \neq 0$.

When ψ is *not* invertible, we might be able to solve an extended problem by introducing an auxiliary random variable Z and an invertible statistic $\tilde{\psi}$ such that $\langle Y, Z \rangle = \tilde{\psi}(X, Z)$. Then, if $\tilde{\psi}$ satisfies the three constraints, we apply the method to $\langle X, Z \rangle$ and statistic $\tilde{\psi}$.

The **Jacobian Method** has three steps:

¹See Chapter 13 and Section 14.1 in Interlude F. A function ψ is invertible with inverse ψ^{-1} if $\psi(\psi^{-1}(u)) = u$ and $\psi^{-1}(\psi(v)) = v$.

1. Find the inverse function ψ^{-1} .
2. Compute the Jacobian J of ψ^{-1} .² The Jacobian is a function that at each point gives the absolute *determinant* of the derivative matrix, $J(y) = |\det(\psi^{-1})'(y)|$.
For scalar statistics, the Jacobian is just the absolute derivative.
3. $K_Y(y) = K_X(\psi^{-1}(y)) J(y)$.

²Note that
 $|\det \psi'| \cdot |\det(\psi^{-1})'| = 1$.

This is easiest to see with examples.

Example 25.16 Scaled Uniform cont'd

Let U and W be the random variables from Example 25.6. We want to find the distribution of W using the Jacobian method.

$\psi(u) = a + (b - a)u$. This is invertible, and $\psi^{-1}(t) = \frac{t-a}{b-a}$. This means that $\psi(\psi^{-1}(t)) = t$ and $\psi^{-1}(\psi(u)) = u$.

The logic is

$$W \text{ near } t \iff \psi(U) \text{ near } t \iff U \text{ near } \psi^{-1}(t)$$

where “near” means infinitesimally close, or

$$W \in [t, t + dt] \iff U \in [\psi^{-1}(t), \psi^{-1}(t + dt)],$$

but $\psi^{-1}(t + dt) = \psi^{-1}(t) + \frac{d}{dt}\psi^{-1}(t) dt$. Hence,

$$\begin{aligned} K_W(t) &= K_U(\psi^{-1}(t)) \left| \frac{d}{dt}\psi^{-1}(t) \right| \\ &= K_U\left(\frac{t-a}{b-a}\right) \left| \frac{1}{b-a} \right| \\ &= \left\{ 0 \leq \frac{t-a}{b-a} \leq 1 \right\} \left| \frac{1}{b-a} \right| \\ &= \{a \leq t \leq b\} \left| \frac{1}{b-a} \right| \\ &= \frac{1}{|b-a|} \{a \leq t \leq b\}. \end{aligned}$$

Example 25.17 Uniform Sum cont'd

Continuing Example 25.7, we have W and Z , where W is a uniform random point in the unit square and $Z = \text{Sum}(W)$. We will find the distribution of Z

using the Jacobian method.

Z is 1-dimensional, but W is 2-dimensional, so ψ is not invertible. But we can define a new statistic that *extends* ψ and is invertible and ψ is the first component:

$$\zeta(a, b) = \langle \psi(a, b), a - b \rangle = \langle a + b, a - b \rangle.$$

Then, $Z := (\text{proj}_1 \circ \zeta)(W)$, and

$$\zeta^{-1}(u, v) = \left\langle \frac{u+v}{2}, \frac{u-v}{2} \right\rangle.$$

Convince yourself that $\zeta(\zeta^{-1}(u, v)) = \langle u, v \rangle$ and $\zeta^{-1}(\zeta(a, b)) = \langle a, b \rangle$.

We take derivatives of every component against each argument giving a matrix

$$D\zeta^{-1}(u, v) = \begin{bmatrix} \frac{d\zeta_1^{-1}}{du} & \frac{d\zeta_1^{-1}}{dv} \\ \frac{d\zeta_2^{-1}}{du} & \frac{d\zeta_2^{-1}}{dv} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

$$J(u, v) = |\det(D\zeta^{-1}(u, v))| = \frac{1}{2}.$$

So, if $Y = \zeta(W)$,

$$\begin{aligned} K_Y(u, v) &= K_W\left(\frac{u+v}{2}, \frac{u-v}{2}\right) \frac{1}{2} \\ &= \frac{1}{2} \left\{ 0 \leq \frac{u+v}{2} \leq 1 \right\} \left\{ 0 \leq \frac{u-v}{2} \leq 1 \right\} \end{aligned}$$

and transforming by proj_1 gives

$$\begin{aligned} K_Z(u) &= \int_v dv \frac{1}{2} \left\{ 0 \leq \frac{u+v}{2} < 1 \right\} \left\{ 0 \leq \frac{u-v}{2} < 1 \right\} \\ &= \frac{1}{2} \int_v dv \left\{ 0 \leq \frac{u+v}{2} < 1 \right\} \left\{ 0 \leq \frac{u-v}{2} < 1 \right\} \\ &= u\{0 \leq u < 1\} + (2-u)\{1 \leq u \leq 2\}, \end{aligned}$$

as before.

Example 25.18 Radial Distance cont'd

Following up on Example 25.8, We use the same strategy, transforming $\langle X, Y \rangle$ to $\langle R, A \rangle$, where A is the angle around the circle. We can write the inverse

transformation ζ^{-1} directly

$$x = r \cos a$$

$$y = r \sin a$$

with $\zeta(x, y) = \langle \sqrt{x^2 + y^2}, \arctan(y/x) \rangle$. The derivative of ζ^{-1} is

$$\begin{bmatrix} \cos a & -r \sin a \\ \sin a & r \cos a \end{bmatrix}$$

which has absolute determinant $r \cos^2 a + r \sin^2 a = r$. Hence,

$$\begin{aligned} K_{RA}(r, a) &= K_{XY}(r \cos a, r \sin a) r \{0 \leq r \leq 1\} \{0 \leq a \leq 2\pi\} \\ &= \frac{1}{\pi} r \{0 \leq r \leq 1\} \{0 \leq a \leq 2\pi\}. \end{aligned}$$

Transforming by proj_1 yields

$$\begin{aligned} f_R(r) &= \int_a da \frac{1}{\pi} r \{0 \leq r \leq 1\} \{0 \leq a \leq 2\pi\} \\ &= 2r \{0 \leq r \leq 1\}, \end{aligned}$$

as before.

Variance and Correlation

26

Chapter

Contents

26.1 Properties of Variance	1041
26.2 Covariance	1043
26.3 The Variance of a Sum	1044
26.4 Higher Dimensional Preliminaries	1046
26.5 Variance in Higher Dimensions	1048

Key Take Aways

The **variance** of a scalar random variable X is defined by

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}(X))^2], \quad (26.1)$$

which gives a non-negative number (which can be infinite).

Variance satisfies several properties, most of which derive directly from the Expectation Rules:

- $\text{Var}(X) \geq 0$. *Variance is always non-negative.*
- $\text{Var}(1) = 0$. *Constants have variance 0.*
- $\text{Var}(cX) = c^2 \text{Var}(X)$ for constant c . *Variance scales quadratically.*
- $\text{Var}(X + c) = \text{Var}(X)$ for constant c . *Variance is translation invariant.*
- $\text{Var}(X) = \mathbb{E}(X^2) - [\mathbb{E}(X)]^2$. *This is the **variance shortcut**.*
- If X and Y have the same distribution, $\text{Var}(X) = \text{Var}(Y)$.

The variance shortcut is the most common way to compute variances.

If we want to convert variance to have the same units as X , we take the square root. This is called the **standard deviation**:

$$\text{SD}(X) = \sqrt{\text{Var}(X)}. \quad (26.2)$$

The **covariance** of X and Y is a measure of association between the two variables, defined by

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}(X))(Y - \mathbb{E}(Y))] \quad (26.3)$$

Covariance satisfies the following properties:

1. If X and Y are independent, $\text{Cov}(X, Y) = 0$.
2. In general, $\text{Cov}(X, Y)$ can be positive or negative.
3. $\text{Cov}(X, X) = \text{Var}(X)$
4. $\text{Cov}(aX + b, cY + d) = ac \text{Cov}(X, Y)$.
5. $\text{Cov}(X + U, Y + W) = \text{Cov}(X, Y) + \text{Cov}(X, W) + \text{Cov}(U, Y) + \text{Cov}(U, W)$.
6. $|\text{Cov}(X, Y)|^2 \leq \text{Var}(X)\text{Var}(Y)$.

We define the **correlation** of two random to be the standardized covariance:

$$\text{Cor}(X, Y) = \frac{\text{Cov}(X, Y)}{\text{SD}(X) \text{SD}(Y)}. \quad (26.4)$$

Correlation is always between -1 and 1 (inclusive); it reaches ± 1 when Y is a linear function of X . In that sense, we say that correlation and covariance measure linear association.

If $X = \sum_{i=1}^n X_i$, then

$$\text{Var}(X) = \sum_{i=1}^n \text{Var}(X_i) + 2 \sum_{i=1}^n \sum_{\substack{j=1 \\ j < i}}^n \text{Cov}(X_i, X_j) = \sum_{i=1}^n \sum_{j=1}^n \text{Cov}(X_i, X_j). \quad (26.5)$$

When the X_i 's are *independent*, $\text{Var}(X) = \sum_i \text{Var}(X_i)$. When the X_i 's are IID, $\text{Var}(X) = n\text{Var}(X_1)$. In the latter case, $\text{Var}(X/n) = \text{Var}(X_1)/n$, so averaging reduces variance.

The definition of variance generalizes directly so that it applies to both vector or scalar random variables:

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}(X))(X - \mathbb{E}(X))^T]. \quad (26.6)$$

If X is an m -vector, then $\text{Var}(X)$ is an $m \times m$ matrix. (When $m = 1$, it's just a number as before.) This matrix has entries $\text{Var}(X)_{ij} = \text{Cov}(X_i, X_j)$; note that on the diagonal $\text{Cov}(X_i, X_i) = \text{Var}(X_i)$.

We saw in the last section that expectation optimizes a notion of *squared prediction*

error. While the expectation gives the optimal prediction, notice that the optimal prediction error that was achieved had the form $\mathbb{E}[(X - \mathbb{E}(X))^2]$.^{*} We think of a distribution as encapsulating our knowledge about a random variable, then this minimum prediction error reflects the intrinsic difficulty of predicting a random variable with that knowledge in hand.

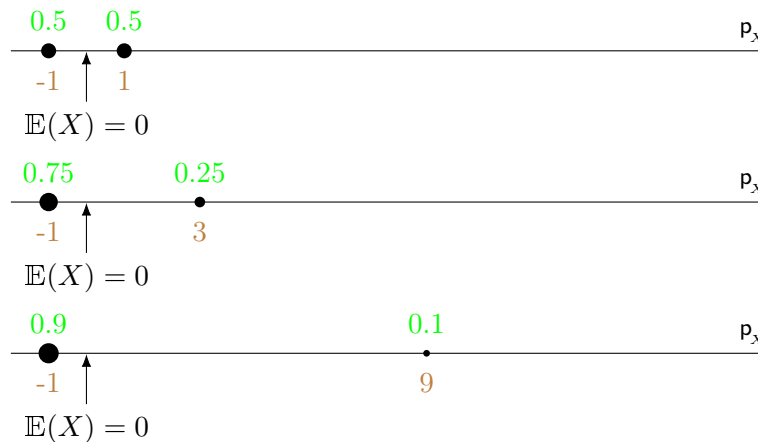
^{*}More precisely, this is our best prediction of our squared prediction error.

If X were a constant random variable, then we *know* its value, and the minimum prediction error is zero. That's the best we can do. In general, it will be positive, and the bigger it is, the less certain we are about X 's value.

We call this expected (squared) prediction error the **variance**, and in this section, we will develop some of its properties. The variance is a useful summary measure of uncertainty, and it (or its square root, the **standard deviation**) are often used descriptively.

The variance can also be seen as a measure of how much a distribution *spreads out* its probability mass. The more spread out, the farther our prediction is likely to be from the true value of the random variable.

Consider the following three distributions. All have the same expectation (0), but their variances (1, 3, and 9, respectively) increase as the probability mass is spread out.



If X is a scalar random variable, its **variance**, $\text{Var}(X)$, is defined by

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}(X))^2]. \quad (26.7)$$

This is a non-negative number.

The transpose of an $m \times n$ matrix A is the $n \times m$ matrix A^T with $(A^T)_{ij} = A_{ji}$. This also works for vectors (which have dimension 1) and scalars (which have both dimensions 1). Thus the

In general, for scalar or vector X , we can write

$$\text{Var}(X) = \mathbb{E} [(X - \mathbb{E}(X))(X - \mathbb{E}(X))^T], \quad (26.8)$$

where $\cdot^T$ denotes transpose. If X has dimension m , then $\text{Var}(X)$ is an $m \times m$ matrix, which we will discuss below.

Let's pull this apart in the scalar case; we will return to the vector case below.

1. X is the (unobserved) value of our measurement.
2. $\mathbb{E}(X)$ is our best prediction of that value.
3. $X - \mathbb{E}(X)$ is the signed difference between the actual value of X and our prediction of it.
4. $(X - \mathbb{E}(X))^2$ is the squared difference between the actual value and our prediction. This is now in “squared units” and is non-negative.
5. $\mathbb{E}[(X - \mathbb{E}(X))^2]$ is our *prediction* of the “squared error” of our prediction of X 's value.

The variance quantifies our prediction accuracy and thus our uncertainty about the value of X . Higher variance means more uncertainty, lower variance less. The smallest possible variance is 0, which only occurs when X is a constant random variable.

Constant random variables have variance zero. Conversely, if a random variable has variance zero, it is a constant with probability 1.

The variance is a measure of prediction accuracy. It also measures the *spread* in a distribution. Low variance means that the distribution concentrates most of the mass in a smaller range; high variance means that the distribution spreads the mass more widely.

You might wonder why we define variance with $(X - \mathbb{E}(X))^2$, which gives units that are the square of X 's units, and not, say, $|X - \mathbb{E}(X)|$ which has the same units as X and is easier to interpret. Good question. The answer is a practical one: it is easier to work with squares in many ways, including deriving properties that allow us to compute variances. We live with the awkwardness of the square, and when we want to describe it in the same units as X , we simply take the square root.

The **standard deviation** of X is the square root of the variance. That is,

$$\text{SD}(X) = \sqrt{\text{Var}(X)}. \quad (26.9)$$

Standard deviation is in the same units as X .

Aside. On Parentheses. In defining variance, we use brackets around the expression inside the expectation because the inner term is squared. This is fine, but there is a convention that you can use to reduce the number of delimiters. The convention is to treat $\mathbb{E}$ as an operator with precedence *higher than addition* and *lower than multiplication*. Under this convention: $\mathbb{E}(X - \mathbb{E}(X))^2$ is the same as $\mathbb{E}[(X - \mathbb{E}(X))^2]$, but not the same as $[\mathbb{E}(X - \mathbb{E}(X))]^2$. Similarly, $\mathbb{E}(XY)$ and $\mathbb{E}XY$ are the same.

26.1 Properties of Variance

- $\text{Var}(X) \geq 0$. *Variance is always non-negative.*

fact $((X - \mathbb{E}(X))^2 \geq 0)$ implies this by the Expectation Ordering Rule.

- $\text{Var}(1) = 0$. *Constants have variance 0.*

Our predictions about constants are perfect, i.e., $\mathbb{E}(1) = 1$. This is the Expectation Constancy Rule.

- $\text{Var}(cX) = c^2\text{Var}(X)$ for constant c . *Variance scales quadratically.*

$\text{Var}(cX) = \mathbb{E}(cX - \mathbb{E}(cX))^2 = \mathbb{E}(cX - c\mathbb{E}X)^2 = c^2\text{Var}(X)$. This comes from the Expectation Scaling Rule.

- $\text{Var}(X + c) = \text{Var}(X)$ for constant c . *Variance is translation invariant.*

$\text{Var}(X + a) = \mathbb{E}(X + a - \mathbb{E}(X + a))^2 = \mathbb{E}(X + a - \mathbb{E}(X) - a)^2 = \text{Var}(X)$. This comes from the Expectation Additivity Rule.

- $\text{Var}(X) = \mathbb{E}(X^2) - [\mathbb{E}(X)]^2$. This is the *variance shortcut*.

$$\begin{aligned}
 \text{Var}(X) &= \mathbb{E}(X - \mathbb{E}(X))^2 \\
 &= \mathbb{E}(X^2 - 2X\mathbb{E}(X) + (\mathbb{E}(X))^2) \\
 &= \mathbb{E}(X^2) - 2\mathbb{E}(X)\mathbb{E}(X) + (\mathbb{E}(X))^2 \\
 &= \mathbb{E}(X^2) - (\mathbb{E}(X))^2.
 \end{aligned}$$

- If X and Y have the same distribution, $\text{Var}(X) = \text{Var}(Y)$.

$\text{Var}(X) = D_X(\psi)$ for a particular statistic ψ . (Which?) So, if $D_X = D_Y$, we get the same prediction for the same algorithm for both rv's.

The *variance is a property of the distribution* because it is a prediction (about our predictions).

Example 26.1 Suppose D has a DiscreteUniform $\langle[1 \dots 11]\rangle$ distribution. Find $\text{Var}(D)$.

We can find $\mathbb{E}(D)$ without calculation because the balancing point is _____. Then, using the variance shortcut

$$\begin{aligned}
 \text{Var}(D) &= \mathbb{E}(D^2) - (\mathbb{E}(D))^2 = \sum_{k=1}^{11} \frac{1}{11} k^2 - 36 \\
 &= \frac{11 \cdot 11.5 \cdot 12}{3} - 36 = 506 - 36 = 470.
 \end{aligned}$$

In units of D : $\text{SD}(D) = \sqrt{470} \approx 21.68$.

Suppose Z has a Uniform $\langle[a, b]\rangle$ distribution. Find $\text{Var}(Z)$.

Again, we know $\mathbb{E}(Z) = (b + a)/2$ as a balancing point. Then

$$\begin{aligned}
 \text{Var}(Z) &= \mathbb{E}(Z^2) - (\mathbb{E}(Z))^2 = \int_{t=a}^b dt \frac{1}{b-a} t^2 - \frac{(b+a)^2}{4} \\
 &= \frac{b^3 - a^3}{3(b-a)} - \frac{a^2 + 2ab + b^2}{4} \\
 &= \frac{a^2 + ab + b^2}{3} - \frac{a^2 + 2ab + b^2}{4} \\
 &= \frac{(b-a)^2}{12}.
 \end{aligned}$$

In units of Z : $\text{SD}(Z) = \sqrt{\text{Var}(Z)} = |b - a|/\sqrt{12}$.

Example 26.2 **Example 26.2. An Exponential Distribution** Suppose L has distribution $D_L(\psi) = \int_{t=0}^{\infty} dt e^{-t} \psi(t)$. Find $\text{Var}(L)$.

$$\begin{aligned}\mathbb{E}(L) &= \int_{t=0}^{\infty} dt e^{-t} t = 1 \\ \mathbb{E}(L^2) &= \int_{t=0}^{\infty} dt e^{-t} t^2 = 2 \\ \text{Var}(L) &= \mathbb{E}(L^2) - (\mathbb{E}(L))^2 = 2 - 1 = 1.\end{aligned}$$

For the second integral, you can look it up or find it using integration by parts:

$$t^2 e^{-t} - 2t e^{-t} = \frac{dt}{d - t^2 e^{-t}}.$$

26.2 Covariance

Related to the variance is a quantity that measures the *linear association* between two random variables. We call it the **covariance**.

The **covariance** of random variables X and Y is defined by

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}(X))(Y - \mathbb{E}(Y))]$$

It measures the association between X and Y : the tendency of X and Y to deviate from their expectations in the same (or opposite) directions.

Covariance is a measure of *linear association* between random variables. Here are a few useful properties we can use:

1. If X and Y are independent, $\text{Cov}(X, Y) = 0$.

By independence, $\mathbb{E}[(X - \mathbb{E}(X))(Y - \mathbb{E}(Y))] = \mathbb{E}(X - \mathbb{E}(X))\mathbb{E}(Y - \mathbb{E}(Y)) = 0$.

Note, however, that $\text{Cov}(X, Y) = 0$ *does not imply independence*.

2. In general, $\text{Cov}(X, Y)$ can be positive or negative.

3. $\text{Cov}(X, X) = \text{Var}(X)$.

Follows directly from the definition.

$$4. \text{Cov}(aX + b, cY + d) = ac \text{Cov}(X, Y).$$

$$(aX + b - \mathbb{E}(aX + b) = a(X - \mathbb{E}(X)) \text{ and similarly for } cY + d.$$

$$5. \text{Cov}(X + U, Y + W) = \text{Cov}(X, Y) + \text{Cov}(X, W) + \text{Cov}(U, Y) + \text{Cov}(U, W).$$

$$(X + U - \mathbb{E}(X + U) = (X - \mathbb{E}(X)) + (U - \mathbb{E}(U)) \text{ and similarly for } Y + W.$$

$$6. |\text{Cov}(X, Y)|^2 \leq \text{Var}(X)\text{Var}(Y).$$

This follows from the a famous inequality (the Cauchy-Schwartz) inequality that we will derive later.

An equivalent formulation is $|\text{Cov}(X, Y)| \leq \text{SD}(X) \text{SD}(Y)$.

Related to covariance is a standardized quantity that expresses the strength of association, the **correlation**.

The **correlation** between random variables X and Y is defined by

$$\text{Cor}(X, Y) = \frac{\text{Cov}(X, Y)}{\text{SD}(X) \text{SD}(Y)}. \quad (26.10)$$

The correlation satisfies $-1 \leq \text{Cor}(X, Y) \leq 1$. If $\text{Cor}(X, Y) = 0$, we say that X and Y are *uncorrelated*.

If $Y := aX + b$, that is if Y is a linear function of X , then $\text{Cor}(X, Y) = \pm 1$, where the sign is determined by the sign of a . It is in this sense that we say that Covariance and Correlation are measures of *linear association*.

26.3 The Variance of a Sum

While it is always true that the expectation of a sum of random variables is the sum of the expectations, the same is *not* true of variance. If random variables are independent, then intuitively, our prediction error for all the random variables will be unaffected by the other variables' values. So it makes sense that if the variables are independent, their variances will sum, and this is true. More generally, the variance in a sum of terms will be affected by any association between the values of the random variables.

Here, we compute in general the variance of a sum.

Consider scalar random variables $X_1, \dots, X_n$, and define

$$X = \sum_{i=1}^n X_i.$$

Our goal here is to compute $\text{Var}(X)$.

Notice that

$$\sum_{i=1}^n X_i - \mathbb{E} \left(\sum_{i=1}^n X_i \right) = \sum_{i=1}^n X_i - \sum_{i=1}^n \mathbb{E}(X_i) = \sum_{i=1}^n (X_i - \mathbb{E}(X_i))$$

so

$$\begin{aligned} \left[\sum_{i=1}^n X_i - \mathbb{E} \left(\sum_{i=1}^n X_i \right) \right]^2 &= \left[\sum_{i=1}^n (X_i - \mathbb{E}(X_i)) \right]^2 \\ &= \sum_{i=1}^n \sum_{j=1}^n (X_i - \mathbb{E}(X_i))(X_j - \mathbb{E}(X_j)) \\ &= \sum_{i=1}^n (X_i - \mathbb{E}(X_i))^2 + \sum_{i=1}^n \sum_{\substack{j=1 \\ j \neq i}}^n (X_i - \mathbb{E}(X_i))(X_j - \mathbb{E}(X_j)). \end{aligned}$$

Taking expectations and applying the Expectation Rules, we get that

$$\begin{aligned} \text{Var}(X) &= \sum_{i=1}^n \text{Var}(X_i) + \sum_{i=1}^n \sum_{\substack{j=1 \\ j \neq i}}^n \text{Cov}(X_i, X_j) \\ &= \sum_{i=1}^n \text{Var}(X_i) + 2 \sum_{i=1}^n \sum_{\substack{j=1 \\ j < i}}^n \text{Cov}(X_i, X_j). \end{aligned}$$

If the X_i s are independent (or even just uncorrelated), then the covariance terms are all zero, so for *independent random variables*

$$\text{Var} \left(\sum_{i=1}^n X_i \right) = \sum_{i=1}^n \text{Var}(X_i). \quad (26.11)$$

If the X_i s are also identically distributed, then

$$\text{Var} \left(\sum_{i=1}^n X_i \right) = n \text{Var}(X_1). \quad (26.12)$$

Hence in this case, averaging *reduces variance*:

$$\text{Var} \left(\frac{1}{n} \sum_{i=1}^n X_i \right) = \frac{1}{n} \text{Var}(X_1), \quad (26.13)$$

by the quadratic scaling of variances.

26.4 Higher Dimensional Preliminaries

We have been working with random variables of dimension > 1 from the beginning and the equations we have developed generally work for any dimension. We would like to continue that trend, and to handle variance for arbitrary dimensions we need to deal (very gently) with matrices and vectors.

When we work with an n -tuple, we can add tuples of like dimension and scale the tuples by a scalar constant:

$$\begin{aligned} \langle a_1, a_2, \dots, a_n \rangle + \langle b_1, b_2, \dots, b_n \rangle &= \langle a_1 + b_1, a_2 + b_2, \dots, a_n + b_n \rangle \\ s \langle a_1, a_2, \dots, a_n \rangle &= \langle sa_1, sa_2, \dots, sa_n \rangle. \end{aligned}$$

When apply linear operators to tuples and take products of tuples, however, we need to make a distinction.

An n -tuple represented as an $n \times 1$ matrix is called a **vector**; an n -tuple represented as an $1 \times n$ matrix is called a **covector**. We use 0 or $\mathbf{0}$ to denote either a vector or covector all of whose components are 0 . By default, if not otherwise specified, an n -tuple is taken to be a vector.

Notice that when $n = 1$, we can elide the distinction among vector, covector, and scalar as needed.

If w is a covector and v is a vector, both of the same dimension, then their product wv is a number given by

$$w \cdot v = \sum_i w_i v_i. \quad (26.14)$$

For compatible vectors u, v , the product $u^T v$ is called the **dot product** of u and v , often denoted by $u \cdot v$.

The **transpose** operation converts a vector to a covector and vice versa, while preserving the dimension. That is, if v is an n -vector, v^T is an n -covector, and if w is an n -covector, w^T is an n -vector.

A **matrix** represents a linear operator on vectors or covectors¹ as a table of

¹See Section 17.3 in Interlude F.

numbers. An $m \times n$ matrix contains mn entries arranged in m rows and n columns. If A is an $m \times n$ matrix, then A_i^j is the entry in row i and column j . For each i , the entries $A_i \cdot$ comprise an n -covector, and for each j , the entries $A \cdot_j$ comprise an m -vector. If v is an n -vector and w is an m -covector, then

$$\begin{aligned}
 Av &= \left\langle \begin{matrix} A_1 \cdot v \\ \vdots \\ A_m \cdot v \end{matrix} \right\rangle \\
 &= \left\langle \begin{matrix} \sum_{j=1}^n A_1^j v_j \\ \vdots \\ \sum_{j=1}^n A_m^j v_j \end{matrix} \right\rangle \\
 wA &= \langle wA^1, wA^2, \dots, wA^n \rangle \\
 &= \left\langle \sum_{i=1}^m w_i A_i^1, \dots, \sum_{i=1}^m w_i A_i^n \right\rangle \\
 wAv &= \sum_{i=1}^m \sum_{j=1}^n w_i A_i^j v_j
 \end{aligned}$$

are, respectively, an m -vector, an n -covector, and a scalar.

If A is an $m \times n$ matrix, then its **transpose**, A^T , is an $n \times m$ matrix with entries $(A^T)_i^j = A_j^i$.

An $m \times n$ matrix A is called **square** if $m = n$, in which case we say it has *dimension* n . A square matrix is **symmetric** if $A_i^j = A_j^i$ for all $i, j \in [1 \dots n]$; that is, $A = A^T$. The $n \times n$ **identity matrix** I (or I_n if the dimension needs to be made clear) is the matrix with 1s in the diagonal entries and 0 in all other entries. This satisfies $Iv = v$ and $wI = w$ for every compatible vector and covector and is symmetric.

If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then $C = AB$ is an $m \times p$ matrix. This matrix has entries

$$C_i^j = \sum_{k=1}^n A_i^k B_k^j,$$

for $i \in [1 \dots m]$ and $j \in [1 \dots p]$. This is called the **product** of A and B . Note that the dimensions of the matrices must be compatible as shown. Also, AB need not equal BA even if both are defined; matrix multiplication is not commutative. We also have that $(AB)^T = B^T A^T$. Notice that from the product formula, if w is an

n -covector and v is an m -vector, then vw is an $m \times n$ matrix with i, j th entry $v_i w_j$. In particular, for a vector v , vv^T is a square matrix with entries $v_i v_j$.

A square matrix A is **invertible** if there exists a matrix A^{-1} , called its **inverse matrix**, such that $AA^{-1} = I = A^{-1}A$.

An invertible matrix A of dimension n satisfies $wAv \neq 0$ for all n -covectors $w \neq 0$ and all n -vectors $v \neq 0$. If $wAv > 0$ in all these cases, A is **positive definite**.

If A is symmetric and positive definite, A is invertible and we can find a symmetric, invertible matrix $A^{1/2}$, called its symmetric square root, such that $A^{1/2}A^{1/2} = A$.

The **determinant** of a square matrix A , denoted $\det A$, is scalar associated with the matrix. An invertible matrix has non-zero determinant, $\det I = 1$ for the identity matrix, and $\det AB = \det A \cdot \det B$. For

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

we have $\det A = ad - bc$. The **trace** of a square matrix is the sum of its diagonal entries: $\text{tr } A = \sum_i A_i^i$.

Finally, if A is a *random matrix*, i.e., a matrix whose entries are scalar random variables, then $\mathbb{E}(A)$ is a matrix of the same shape with $\mathbb{E}(A)_i^j = \mathbb{E}(A_i^j)$.

If $wAv \geq 0$ for all non-zero vectors and covectors, the matrix is **non-negative definite**.

26.5 Variance in Higher Dimensions

As we saw earlier, the definition of variance generalizes directly so that it applies to both vector or scalar random variables:

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}(X))(X - \mathbb{E}(X))^T]. \quad (26.15)$$

Here, ^{T} denotes the transpose operation converting the vector $X - \mathbb{E}(X)$ to a covector of the same dimension. The product is thus a matrix with i, j th entry $\mathbb{E}[(X_i - \mathbb{E}(X_i))(X_j - \mathbb{E}(X_j))] = \text{Cov}(X_i, X_j)$. Note that we treat an n -dimensional random variable as an n -vector by default.

Definition 62. The Variance Matrix.

If the random variable X has dimension n , the **variance matrix** of X is an $n \times n$ matrix defined by:

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}(X))(X - \mathbb{E}(X))^T]. \quad (26.16)$$

The i, j entry of this matrix is

$$\text{Var}(X)_i^j = \text{Cov}(X_i, X_j). \quad (26.17)$$

On the diagonal, where $(i = j)$, we have $\text{Var}(X_i) = \text{Cov}(X_i, X_i)$. Hence, $\text{Var}(X)$ is a matrix with the variances of the components on the diagonal and the covariances of different components off the diagonal.

When $n = 1$, $\text{Var}(X)$ is a number just as before, and this matches our definition of variance for scalar random variables.

The variance of a sum of random variables has a nice expression in terms of the variance matrix. Suppose X is an n -vector random variable and v is an n -vector constant. Then, $v^T X = \sum_{i=1}^n v_i X_i$ and taking variances we have

$$\begin{aligned} \text{Var}(v^T X) &= \text{Var}\left(\sum_{i=1}^n v_i X_i\right) \\ &= \sum_{i=1}^n v_i^2 \text{Var}(X_i) + \sum_{\substack{i,j=1 \\ i \neq j}}^n v_i v_j \text{Cov}(X_i, X_j) \\ &= \sum_{i=1}^n \sum_{j=1}^n v_i \text{Cov}(X_i, X_j) v_j \\ &= v^T \text{Var}(X) v. \end{aligned} \quad (26.18)$$

The variance matrix of a random variable is square, symmetric, and non-negative definite (because $\text{Var}(v^T X) \geq 0$). If it is positive definite, then we can find its symmetric square root, giving a matrix $\text{SD}(X)$ of the same dimension.

If X and Y are random variables of dimensions m and n , we define $\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}(X))(Y - \mathbb{E}(Y))^T]$. This is an $m \times n$ matrix with i, j th entry $\text{Cov}(X_i, Y_j)$.

Conditional Independence and Graphical Models

27

Chapter

Contents

27.1 Directed Acyclic Graphs	1052
27.2 Kernel Decompositions	1052
27.3 Graphical Models	1056

Key Take Aways

For any collection of random variables, we can express the various conditional independence relationships among them via a **directed acyclic graph**, and we have an algorithm for identifying those relationships from the graph.

Via the Multiplication Rule, a DAG specifies a way to decompose the Kernel of a Distribution in a way that expresses the conditional independence relations among the components of the random variable.

We identify four core decompositions:

1. Independence
2. Markov Chain
3. Common Cause (Fork)
4. Common Effect (Collider)

By combining these four decompositions, we develop an algorithm for identifying which variables are conditionally independent given any subset of the remaining variables.

27.1 Directed Acyclic Graphs

A directed graph is a collection of nodes and edges where each edge goes in one direction from a *source* node to a *destination* node. A directed graph is simple when there is at most one edge between any two nodes and no edges from a node to itself.

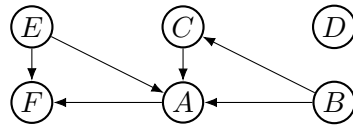


FIGURE 27.1. An example simple, directed acyclic graph.

Formally, a *directed (simple) graph* is a pair $\langle \mathcal{N}, \mathcal{E} \rangle$, where $\mathcal{N}$ is a set of *nodes* and $\mathcal{E} \subseteq \{ \langle s, d \rangle \in \mathcal{N} \times \mathcal{N} \mid s \neq d \}$ is a set of *edges*. Each $\langle s, d \rangle \in \mathcal{E}$ represents an edge from source node s to destination node d . That is, edges are directed, and $\langle s, d \rangle$ is not the same as $\langle d, s \rangle$ when $s \neq d$.

A graph is simple if there is at most one edge between any pair of nodes, and no edges from a node to itself.

A directed graph has a cycle if there is a path of length at least two from a node back to itself following edges from source to destination. A *cycle* is a sequence $s_0, s_1, \dots, s_n, s_0$ of nodes where the s_i are distinct and $\langle s_i, s_{i+1} \rangle \in \mathcal{E}$ for $i \in [0 \dots n]$ and $\langle s_n, s_0 \rangle \in \mathcal{E}$. This is a path that leads from a node back to itself following edges in the direction of their arrows (from source to destination). A directed graph with no cycles is called a *directed acyclic graph (DAG)*.

The nodes of any DAG can be ordered so that when there is an edge $s \rightarrow d$, it implies that s comes before d in the ordering. We call this the *generating order* of the nodes. Here, we consider DAGs in which each *node* is associated with a *random variable*.

27.2 Kernel Decompositions

Let $X = \langle X_1, \dots, X_n \rangle$ be a random variable, and suppose we have a DAG $(\mathcal{N}, \mathcal{E})$ whose nodes are the X_i 's. Define $\text{parent}(X_i)$ to be the set of nodes X_j for which there is an edge from X_j to X_i . Define P_i be the vector of parents of X_i ; that is, the vector with components $X_j \in \text{parent}(X_i)$ listed in generating order.

We can view this DAG as a recipe for generating values of X and for decomposing the kernel of D_X into simpler factors.

1. List the nodes in *generating order* $X_{i_1}, X_{i_2}, \dots, X_{i_n}$.

2. By definition, X_{i_1} has no parents, so we can generate it from $D_{X_{i_1}}$.

This distribution has kernel $K_{X_{i_1}}$.

3. Having generated $X_{i_1}, \dots, X_{i_{m-1}}$ for $1 < m \leq n$, we generate X_{i_m} from the conditional distribution of X_{i_m} given its parents. That is, we generate X_{i_m} from $D_{X_{i_m} | P_{i_m}}$. This distribution has kernel $K_{X_{i_m} | P_{i_m}}$.

This algorithm lets us generate a value of X and gives us a decomposition of K_X of the form:

$$K_X(x) = K_{X_{i_1}}(x_{i_1}) K_{X_{i_2} | P_{i_2}}(x_{i_2} | p_{i_2}) \cdots K_{X_{i_n} | P_{i_n}}(x_{i_n} | p_{i_n}) \quad (27.1)$$

This decomposition need not be unique.

For a random variable X with distribution D_X , a DAG $(\mathcal{N}, \mathcal{E})$ is a **valid representation** of D_X if and only if the decomposition in equation (27.1) is true.

We thus use the DAGs as a graphical language for describing the structure of a random variable's distribution, and in particular the conditional independence relationships among its components. Not any DAG is valid for a particular distribution D_X , but for any distribution D_X , we can find some valid DAG.

For a random variable $X = \langle X_1, X_2, \dots, X_n \rangle$, we have the following decomposition of D_X 's kernel by the *Multiplication Rule*:

$$K_X(x) = K_{X_1}(x_1) K_{X_2 | X_1}(x_2 | x_1) K_{X_3 | X_1 X_2}(x_3 | x_1, x_2) \cdots K_{X_n | X_1 \dots X_{n-1}}(x_n | x_1, \dots, x_{n-1}) \quad (27.2)$$

$$= \cancel{K_{X_1}(x_1)} \frac{K_{X_1 X_2}(x_1, x_2)}{\cancel{K_{X_1}(x_1)}} \frac{K_{X_1 X_2 X_3}(x_1, x_2, x_3)}{\cancel{K_{X_1 X_2}(x_1, x_2)}} \cdots \frac{K_{X_1 \dots X_n}(x_1, \dots, x_n)}{\cancel{K_{X_1 \dots X_{n-1}}(x_1, \dots, x_{n-1})}}. \quad (27.3)$$

This means that we can always find a valid DAG: with an edge from X_1 to each of $X_2, \dots, X_n$; an edge from X_2 to each of $X_3, \dots, X_n$; ...; and an edge from X_{n-1} to X_n . And indeed we can find $n!$ such DAGs, since a decomposition similar to (27.2) works for any permutation of $X_1, \dots, X_n$.

The value of these decompositions comes when the DAG reveals structure simpler than the decomposition in (27.2).

Example 27.1. Let $X = \langle A, B, C, D, E \rangle$ where all of the components are **independent**. This gives the DAG:

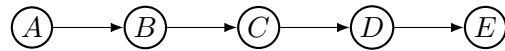


Following the approach above, we have the decomposition

$$K_{ABCDE}(a, b, c, d, e) = K_A(a) K_B(b) K_C(c) K_D(d) K_E(e), \quad (27.4)$$

as we would expect.

Example 27.2. Let $X = \langle A, B, C, D, E \rangle$ whose distribution is validly described by the following DAG:



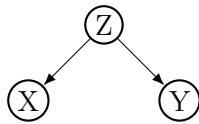
Following the approach above, we have the decomposition

$$K_{ABCDE}(a, b, c, d, e) = K_A(a) K_{B|A}(b | a) K_{C|B}(c | b) K_{D|C}(d | c) K_{E|D}(e | d). \quad (27.5)$$

This is more complicated than independence but only slightly. It shows (for reasons we will see in the next section) that A and C are conditionally independent given B , that B and D are conditionally independent given C , and so on.

This pattern is called a **Markov chain**.

Example 27.3. Suppose the random variable $\langle X, Y, Z \rangle$ has a valid representation:



This corresponds to the decomposition

$$K_{XYZ}(u, v, w) = K_Z(w) K_{X|Z}(u | w) K_{Y|Z}(v | w) \quad (27.6)$$

As a shorthand, we can write the decomposition without arguments make the pattern more salient: $K_{XYZ} = K_Z K_{X|Z} K_{Y|Z}$.

This pattern is called a **common cause** (aka. a **fork**). Here, Z is the node for which the pattern is named as we can generate $\langle X, Y, Z \rangle$ by first generating Z and then X and Y using the value of Z .

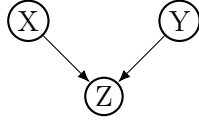
This pattern implies that X and Y are conditionally independent given Z :

$$K_{Y|XZ}(v | u, w) = K_{Y|Z}(v | w) \quad (27.7)$$

$$K_{X|YZ}(u | v, w) = K_{X|Z}(u | w) \quad (27.8)$$

$$K_{XY|Z}(u, v | w) = K_{X|Z}(u | w) K_{Y|Z}(v | w). \quad (27.9)$$

Example 27.4. Suppose the random variable $\langle X, Y, Z \rangle$ has a valid representation:



This gives the decomposition

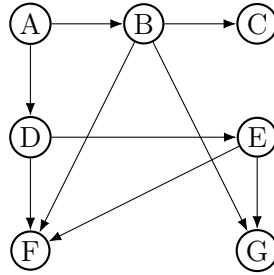
$$K_{XYZ}(u, v, w) = K_X(u) K_Y(v) K_{Z|XY}(w | u, v) \quad (27.10)$$

or for short,

$$K_{XYZ} = K_X K_Y K_{Z|XY}. \quad (27.11)$$

This pattern is called a **common effect** (aka. a **collider**). The latter name comes from the heads of the arrows meeting – or colliding – at the middle node.

Example 27.5. The following DAG



corresponds to the decomposition

$$K_{ABCDEFG} = K_A \cdot K_{B|A} \cdot K_{D|A} \cdot K_{C|B} \cdot K_{E|D} \cdot K_{F|BDE} \cdot K_{G|BE}.$$

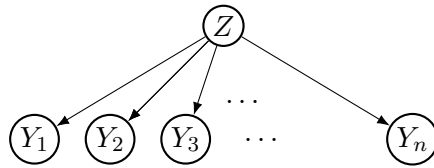
Example 27.6 **Example 27.6. Learning a Parameter from Data**

The distribution of continuous random variables $Y = \langle Y_1, \dots, Y_n \rangle$ and Z has a decomposition

$$f_{Y_1, \dots, Y_n, Z}(s_1, \dots, s_n, t) = f_Z(t) f_{Y_1, \dots, Y_n | Z}(s_1, \dots, s_n | t) \quad (27.12)$$

$$= f_Z(t) \prod_{i=1}^n f_{Y_i | Z}(s_i | t) \quad (27.13)$$

which corresponds to the DAG:

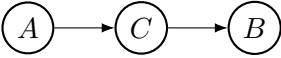
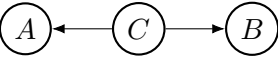
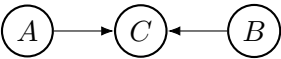


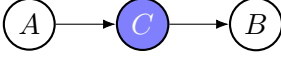
27.3 Graphical Models

In analyzing complex models, a useful way to simplify our work is to identify (conditional) independence relationships among the variables.

A DAG that gives a valid representation for the distribution of a random variable X gives us a decomposition for the kernel of that distribution. And in the process, it allows us to identify all the conditional independence relations among nodes in the graph.

We start with four core decompositions

1. Independence 
2. Markov Chain 
3. Common Cause (aka Fork) 
4. Common Effect (aka Collider) 

We indicate when the value of a random variable is known (in reality or hypothetically) by coloring in the node. For instance,  indicates that the value of C is given.

Thus, we start with a DAG that gives a valid representation of the distribution of a vector-valued random variable X , where the nodes of the DAG are the components

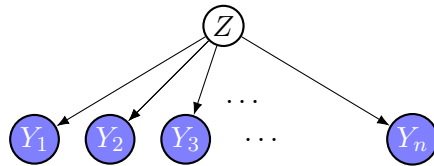
of X . We add to the information in a DAG by coloring in any nodes for which the corresponding random variable's value is given/known.

Algorithm 1 below describes how to determine whether random variables X and Y (of arbitrary dimension) are conditionally independent given Z . Recognizing this structure in a distribution allows us to simplify calculations of predictions from that distribution.

Example 27.7 Example 27.7. Learning a Parameter from Data (cont'd)

In Example 27.6, we showed a DAG where the distribution of data $Y_1, \dots, Y_n$ is determined by the value of an unknown parameter. Given that parameter, the different data points are conditionally independent. A common statistical problem is to use the data $\langle Y_1, \dots, Y_n \rangle$ to infer the value of the parameter Z .

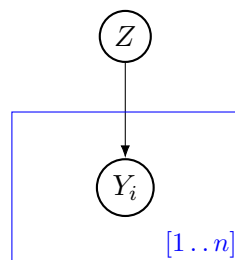
The DAG reflecting that situation shows that the data are *observed*:



One more bit of graphical language that is sometimes useful. When we have repeated structure in the graph, we can use a **plate** to represent the repeated structure. A plate is a rectangular region marked on the graph to indicate which structure in the graph is repeated; a range of indices annotates the plate to specify which random variables are included. This is most easily seen via an example.

Example 27.8 Plates

The DAG in Example 27.6 has the Y_i 's structure repeated, requiring unsatisfying and space-consuming “...” to implicitly describe the repetition. Here's how we would depict it with plates:



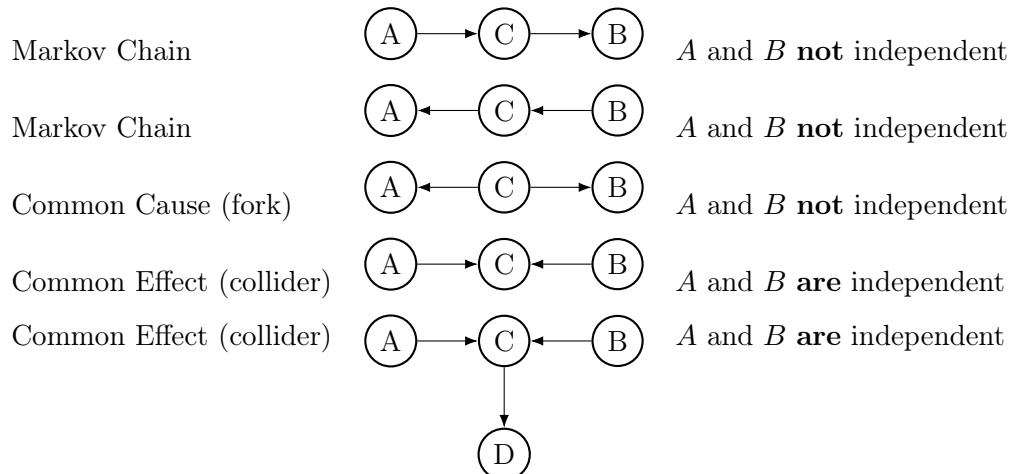
Or, if the index set is $[1..n]$, we annotate it more simply



The Four Core Decompositions

Four patterns of three-node subgraphs will be the main data in the Algorithm described in the next subsection. Here we identify those four patterns, both with and without the central node being given.

First, assume that none of the random variables has been observed.



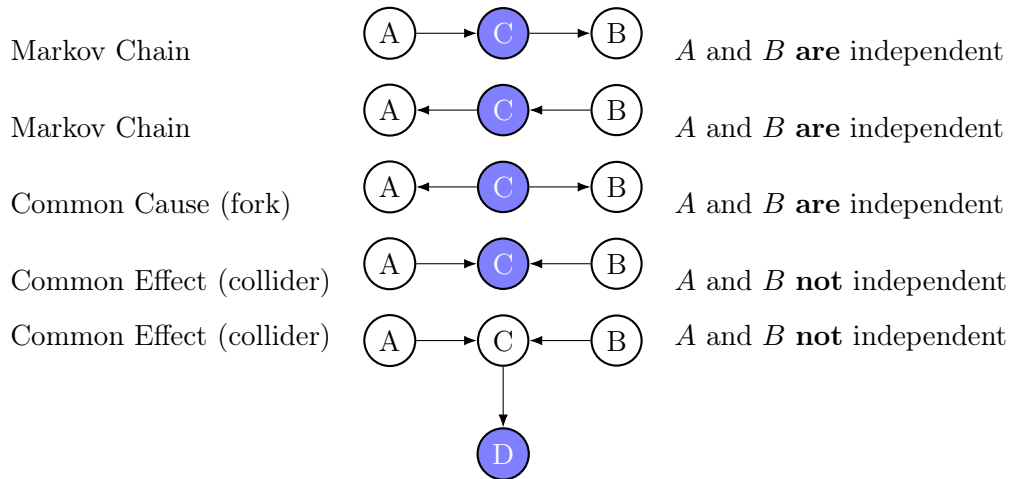
Without knowledge of the value of C (or D), we can assess the dependence or independence of A and B .

- In the independence pattern, all three random variables are mutually independent, so knowledge of C has no impact on the dependence of A and B .
- For the Markov chain, we can think of A, C, B as representing the states of a system as it evolves in time. In this role, A stands in for the “past”, C for the “present,” and B the “future.” Without any information, the past and future are dependent, but *given the present state*, the future state does not depend on how we got to the present state. The future and past are conditionally independent given the present.
- For the common cause, A and B are both “influenced” by C , so learning A and exploiting the connection with C gives us information about B . A and B are

not independent, but if we learn the value of C , there is no more to be gleaned from that connection. So A and B are conditionally independent given C .

- For the common effect, A and B are independent, but if we learn the value of C , then we can infer – from the connection between A and C and B and C – information about A from B and vice versa. The same happens if instead we see the value of D , a descendant of C in the graph.

We can see the impact of learning C (or D) in the next display. Assume that the values of C or D are given.



Recognizing these four decompositions is the key to identifying conditional independence in general.

Identifying Conditional Independence Relations

We will consider *subsets* of nodes in a DAG $(\mathcal{N}, \mathcal{E})$. Each such subset corresponds to a vector random variables whose components are the node variables in the subset. We thus identify a subset of nodes with the corresponding vector random variable.

If $\mathcal{V} \subseteq \mathcal{N}$ is a subset of nodes in the graph with $\mathcal{V} = \{V_1, \dots, V_m\}$, define the random variable

$$\text{vec}(\mathcal{V}) := \text{vec}(V_1, \dots, V_m),$$

where vec simply lists all the variables (by component) in $\mathcal{V}$. We identify the subset $\mathcal{V}$ with the vector random variable $V := \text{vec}(\mathcal{V})$.

The question we now turn to is how to determine if two subsets of random variables in a DAG are conditionally independent given a third subset. Specifically, identify *disjoint* node sets $\mathcal{A}, \mathcal{B}, \mathcal{G} \subseteq \mathcal{N}$ with associated random variables $A := \text{vec}(\mathcal{A}), B := \text{vec}(\mathcal{B}), G := \text{vec}(\mathcal{G})$, respectively. We ask whether A and B are

Statistic vec defined on
ATTNeq:tensorp-vec,
 $\text{vec}(u, v, w) = u \star v \star w$ et
cetera.

conditionally independent given Z ; that is,

$$D_{A|BG} = D_{A|G}. \quad (27.14)$$

Remember that A , B , and G are in general vector random variables.

Algorithm 1 describes how to answer this question.

Algorithm 1. Identifying Conditional Independence Relations

Input: Disjoint node sets $\mathcal{A}$, $\mathcal{B}$, and (given set) $\mathcal{G}$, with associated random variables A, B, G .

Procedure:

1. Color the nodes in the given set $\mathcal{G}$ blue.
2. For every *undirected path* between a node in $\mathcal{A}$ and a node in $\mathcal{B}$ (i.e., ignoring directions of arrows), check whether either of the following conditions holds for that path:

(N) There is a G node on the path where the arrows *on the path* meet either head-to-tail or tail-to-tail.

(C) There is a node on the path where the arrows *on the path* meet head-to-head such that neither the node nor any of its descendants are G nodes.

If either condition (N) or (C) holds for a node on the path, we say that the path is **blocked** at that node.

3. **If every path between $\mathcal{A}$ and $\mathcal{B}$ is blocked, then A is conditionally independent of B given G .**

A key point in this algorithm is that when checking condition N or C for a path in the graph, we view that path *in isolation* from the rest of the graph. For that subgraph only, we ask whether

(N) There is a G node that is a Markov chain or a common cause/fork.

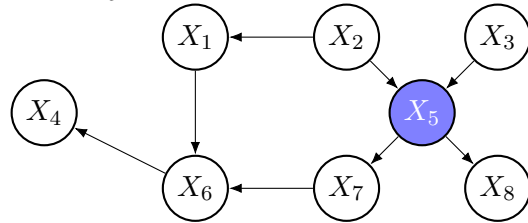
(C) There is a common effect/collider that is not a G node and has no descendants that are G nodes.

Notice that the endpoints of the path are not included in these checks. The only exception to this is that when identifying descendants of a collider in condition (C),

we look in the original graph.

Example 27.9.

$X_1, \dots, X_8$ $G = X_5$.



Is X_i conditionally independent of X_j given X_5 ?

i j $A = X_i, B = X_j$

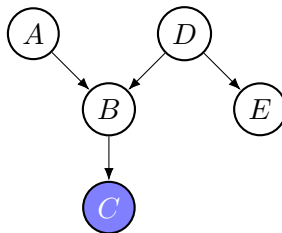
2 7 Yes

2 3 No

4 7 No

8 4 Yes

Example 27.10.

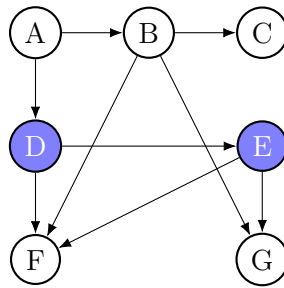


Is A conditionally independent of E given C ?

Is A conditionally independent of E given D ?

Applying Algorithm 1, the answers are No and Yes, respectively.

Example 27.11.



Are A and G conditionally independent given D, E ? No

Are A and F conditionally independent given D, E ? No

Are A and C conditionally independent given D, E ? No

Interlude C

Canonical Distributions

Canonical Distributions

28

Chapter

Key Take Aways

There are some Kinds/Distributions/Kernels that arise frequently in practice because the pattern of assumptions that give rise to them recur. We give these **canonical Distributions** names to make it easy to refer to them and to communicate those common assumptions.

The name of a canonical Distribution has two parts: the *family name* and the *parameters* that specify an individual member of the family. A canonical Distribution name has the form

`FamilyName(parameters1, ...)`

For example, `Uniform(a, b)`, `Binomial(n, p)`, or `Normal(μ, σ2)`.

We have already seen several families repeatedly without naming them. These include the `DiscreteUniform`, `Uniform`, `Exponential`, `Geometric`, `Bernoulli`, `Binomial`, `Poisson`, and `Normal`. We will see more.

In Part I, we encountered patterns of assumptions that again and again produced Kinds of a common form. For example, a Kind with equal weights on its values or a Kind in canonical form whose only values are 0 and 1, with weights $1 - p$ and p for some number $0 < p < 1$. Recognizing these patterns, `frplib` creates *factories* that produce such Kinds:

```
constant(a)      # The constant Kind with value a
binary(p)        # (1 - p) {__ == 0} + p {__ == 1}
uniform(u, v, w)  # u, v, w equally likely
...
```

Each of these factories names a *family* of Kinds, and individual Kinds in a particular family are specified by the values of their *parameters*. For example, `binary` gives

the family of all Kinds with only 0 and 1 as values, and the parameter $\mathbf{p}$ specifies such a Kind uniquely by the weight on the 1 branch.

We saw similar patterns in Chapter 21, where we derived the $\text{DiscreteUniform}\langle\mathcal{S}\rangle$ and $\text{Uniform}\langle\mathcal{R}\rangle$ Distributions from a single basic assumption – symmetry. Symmetry is quite often a reasonable assumption, and it can apply to a wide variety of systems. And indeed, we can define uniform Distributions for *any* discrete set $\mathcal{S}$ and for any n -dimensional region $\mathcal{R}$. Notice though that if $\mathcal{S} \neq \mathcal{S}'$, then $\text{DiscreteUniform}\langle\mathcal{S}\rangle \neq \text{DiscreteUniform}\langle\mathcal{S}'\rangle$. Similarly, if $\mathcal{R} \neq \mathcal{R}'$, then $\text{Uniform}\langle\mathcal{S}\rangle \neq \text{Uniform}\langle\mathcal{S}'\rangle$. For instance, a $\text{DiscreteUniform}\langle\{-1, 0, 1\}\rangle$ is not equal to a $\text{DiscreteUniform}\langle\{0, 1, 2\}\rangle$; and a $\text{Uniform}\langle[-1\_1]\rangle$ is not equal to a $\text{Uniform}\langle[0\_2]\rangle$. So uniform Distributions on different base sets are *different Distributions*; yet they are related through a common structure based on that common assumption of symmetry.

Again, we think of these as describing *families* of related Distributions, with the individuals within the families distinguished by the values of their parameters. DiscreteUniform is such a family; the parameter is the set of values. Similarly, the Uniform family describes continuous uniform Distributions with a parameter that is the region in space over which the probability is evenly spread. The special case $\text{Uniform}\langle a, b \rangle$, with two parameters $a < b$, is a synonym for a continuous $\text{Uniform}\langle [a\_b] \rangle$ Distribution over an interval. The Distributions in each family are related by a similar structure, but the details differ for the individual. When we write $\text{DiscreteUniform}\langle\mathcal{S}\rangle$ or $\text{Uniform}\langle\mathcal{R}\rangle$, we are specifying an individual Distribution by giving its *family name* and a tuple of *parameters* that identifies the individual within the family.

When we want to refer to a unique individual person, we say their family name and their given name (including modifiers like middle name and postnominals). When we want to refer to a unique Distribution, we can say the family name and specify the values of the parameters that determine the individual Distribution in the family.

Individual Person: Family Name, Given Name
Individual Distribution: Family Name, Parameter(s)

FIGURE 28.1. Analogy between individual names and canonical Distribution names

Canonical families of Distributions arise because of recurrent patterns of assumptions. For example, Uniform Distributions are derived from an assumption of symmetry over chosen items or points. Binomial Distributions are the sums of IID events – the count of independent occurrences that all have the same probability. Exponential Distributions are models for waiting times that are “memoryless” in a

certain sense. And Normal Distributions arise from aggregating many small random effects that are not too extreme individually.

When we recognize these patterns, we can specify the Distribution of a quantity accordingly. Canonical Distributions are thus conveniences that save us the work of re-deriving a Distribution again and again in isomorphic situations. They are also useful for succinctly communicating some basic models. When you are asked to find the Distribution of a quantity, if you recognize those assumptions and can identify the target Distribution as a member of a Canonical family, you merely need to name it to specify it fully. That's a nice feature. But otherwise, these are just examples of Distributions that happen to arise frequently.

A **canonical family of Distributions** is a *named, parameterized family* of Distributions arising from shared assumptions that differ only in the values of the parameters. These assumptions and Distributions occur frequently enough in practice to make it convenient to give the family a name.

A Distribution in a canonical family is called a **canonical Distribution**.

Instead of a mathematical expression, we can refer to a specific canonical Distribution by giving its name and a tuple of parameters. We specify these as follows:

$$\text{FamilyName}\langle \text{parameters}_1, \dots \rangle$$

where multiple parameters are separated by commas. For example: $\text{Uniform}\langle [0\_1] \rangle$ and $\text{DiscreteUniform}\langle \{1, 2, 3\} \rangle$, each of which takes a single parameter.

It is important to recognize that the name we use for a canonical Distribution is just a *shorthand for convenient communication*, not a new type of object. For instance, the $\text{Uniform}\langle [0\_1] \rangle$ Distribution, or $\text{Uniform}\langle 0, 1 \rangle$ Distribution for short, refers to the Distribution D given by

$$D(\psi) = \int_t dt \{0 \leq t \leq 1\} \psi(t),$$

but we can say the name rather than writing the integral if it makes sense to do so.

Once we have specified a family, then the tuple of parameters uniquely determines the Distribution within the family. We can typically determine the expectation (and variance and other properties) from the parameters, and sometimes these *are* the parameters.

Examples We Have Seen

We have already encountered and used several canonical Distributions without naming them. A bigger list with properties is given in the Help Sheet (Appendix H).

- Binary $\langle p \rangle$ or more commonly called Bernoulli $\langle p \rangle$, for $0 \leq p \leq 1$.

The Kind/Distribution of an event V with $\mathbb{E}(V) = p$.

Canonical Example: The flip of a balanced coin

- DiscreteUniform $\langle \mathcal{S} \rangle$.

A Distribution representing the choice among values in a discrete set with equal probabilities.

$$D(\psi) = \sum_s \frac{\mathcal{S}(s)}{\#\mathcal{S}} \psi(s).$$

Canonical Examples: Rolling a balanced die, choosing a value in a lottery

- Binomial $\langle n, p \rangle$, $n \in [1 \dots], 0 < p < 1$.

The count of “marks” in fixed number of trials, as we will see later.

$$D(\psi) = \sum_k \binom{n}{k} p^k (1-p)^{n-k} \{k \in [0 \dots n]\} \psi(k).$$

The sum of n IID events has a Binomial Distribution.

Canonical Example: Number of heads in a fixed number of coin flips.

The Hunter’s Success example in Chapter 8 where all hunters have equal hit probabilities produced this Kind. The Bernoulli $\langle p \rangle$ Distribution is the same as the Binomial $\langle 1, p \rangle$.

- Geometric $\langle p \rangle$, $0 < p < 1$.

Waiting time for the first “mark” in sequence of trials, as we will see later.

$$D(\psi) = \sum_k (1-p)^{k-1} p \{k \in [1 \dots]\} \psi(k).$$

The “Waiting for Heads” example is a prototype for this Kind.

Canonical example: Number of flips of a coin until the first heads.

- Poisson $\langle \lambda \rangle$, $\lambda > 0$.

A Distribution often used to model the counting a random scatter of points in time and space and to model the count of “rare” events.

$$D(\psi) = \sum_k e^{-\lambda} \frac{\lambda^k}{k!} \{k \in [0 \dots]\} \psi(k).$$

Canonical example: Number of radioactive decays from a sample in a time interval.

- [Uniform](#) $\langle \mathcal{R} \rangle$ and [Uniform](#) $\langle a, b \rangle$.

A Distribution representing the symmetric choice of a point in a continuous region.

$$D(\psi) = \int_s ds \frac{1}{\text{measure } \mathcal{R}} \mathcal{S}(s) \psi(s).$$

An important special case occurs when the region is just a real interval, $\mathcal{R} = [a\_b]$. This case is useful and common enough to merit its own name; we call it the [Uniform](#) $\langle a, b \rangle$ for an $-\infty < a < b < \infty$:

$$D(\psi) = \int_s ds \frac{1}{b-a} \{a \leq s \leq b\} \psi(s).$$

Note that it does not matter whether you consider the endpoints as belonging to the interval since the Distribution is continuous. [Uniform](#) $\langle a, b \rangle$ is equivalent to [Uniform](#) $\langle [a\_b] \rangle$, [Uniform](#) $\langle (a\_b) \rangle$, [Uniform](#) $\langle (a\_b] \rangle$, and [Uniform](#) $\langle [a\_b) \rangle$.

- [Exponential](#) $\langle \lambda \rangle$, $\lambda > 0$.

A “memoryless” waiting time.

$$D(\psi) = \int_t dt \lambda e^{-\lambda t} \{t > 0\} \psi(t).$$

The expectation is $1/\lambda$, and λ is often called the *rate parameter*.

Canonical examples: Lifetime of a component or organism, waiting time for arrivals in a queue.

- [Gamma](#) $\langle \alpha, \lambda \rangle$ with $\alpha > 0$ and $\lambda > 0$.

$$D(\psi) = \int_z dz \frac{\lambda^\alpha}{\Gamma(\alpha)} z^{\alpha-1} e^{-\lambda z} \{z > 0\} \psi(z).$$

This has expectation α/λ . The [Exponential](#) $\langle \lambda \rangle$ is the same as the [Gamma](#) $\langle 1, \lambda \rangle$.

Canonical example: waiting time for a fixed number of arrivals in a queue.

- Normal $\langle\mu, \sigma^2\rangle$

This critically important family of Distributions is discussed in Chapter 29.

- Beta $\langle a, b\rangle$ with $a, b > 0$.

$$D(\psi) = \int_u du \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} u^{a-1} (1-u)^{b-1} \{0 \leq u \leq 1\} \psi(u).$$

This has expectation $a/(a+b)$. The Beta $\langle 1, 1\rangle$ is the same as the Uniform $\langle 0, 1\rangle$.

The Normal Distributions

29

Chapter

Key Take Aways

The Normal (aka Gaussian) family of Distributions arises frequently in theory and applications. It is not an exaggeration to say that this is the most important family of Distributions in probability theory.

A random variable X of dimension n has a $\text{Normal}\langle\mu, \sigma^2\rangle$ Distribution for n -vector μ and positive definite $n \times n$ matrix σ^2 when

$$D_X(\psi) = \int \cdots \int_x \frac{1}{\sqrt{\det 2\pi\sigma^2}} e^{-\frac{1}{2}(x-\mu)^T \sigma^{-2}(x-\mu)} \psi(x).$$

Here, $\det \sigma^2$ is the determinant of the matrix σ^2 , and σ^{-2} is its inverse.

Then: $\mathbb{E}(X) = \mu$ and $\text{Var}(X) = \sigma^2$.

We often label the dimension in the name of the Distribution when it is not clear from context and call this the $\text{Normal}_n\langle\mu, \sigma^2\rangle$ Distribution. If the dimension is omitted and has not been specified elsewhere, we take the scalar case ($n = 1$) as the default.

In the scalar case, the Kernel reduces to the famous bell curve:

$$K_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.$$

The **standard Normal Distribution** is $\text{Normal}\langle 0, I \rangle$, where I is the identity matrix. This reduces to $\text{Normal}\langle 0, 1 \rangle$ in the scalar case. If Z is a standard Normal random variable, then its components are independent $\text{Normal}\langle 0, 1 \rangle$ random variables and

$$X := \mu + \sigma Z$$

has a $\text{Normal}\langle\mu, \sigma^2\rangle$ Distribution. Conversely, if X has a $\text{Normal}\langle\mu, \sigma^2\rangle$ Distribution, then

$$Z := \sigma^{-1}(X - \mu)$$

has a standard Normal Distribution.

The CDF of a Normal $\langle 0, 1 \rangle$ Distribution is a special function denoted Φ . It is a smooth function that increases from 0 at $-\infty$ to 1 at ∞ with $\Phi(0) = 1/2$. This function has an inverse Φ^{-1} that is also smooth and increasing; it maps a number in $(0, 1)$ to a real number. $\Phi^{-1}(\Phi(u)) = u$ and $\Phi(\Phi^{-1}(q)) = q$.

Random variables with Normal Distributions have many useful properties, including:

- Sums of Normals are Normal
- Linear functions of Normals are Normal
- Uncorrelated Normals are independent
- The conditional Distribution of a Normal given a Normal is Normal.

By far the most important and commonly used family of canonical Distributions is the family of **Normal Distributions**, also known as *Gaussian Distributions*. It arises commonly in problems across many fields including Statistics and Machine Learning, Physics and Astronomy, Engineering, Finance, Mathematics, and many more. We will make heavy use of the Normal Distributions, so it is worth introducing them here and learning about their properties.

To get a feel for the Distribution, we will start with the scalar case. The Kernel of the scalar Normal Distribution is the classic “bell curve,” as shown in Figure 29.1. We say that a scalar random variable has a (scalar) **Normal $\langle \mu, \sigma^2 \rangle$** Distribution, for a real number μ and a positive number σ^2 , when

$$D_X(\psi) = \int_{-\infty}^{\infty} dx \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \psi(x). \quad (29.1)$$

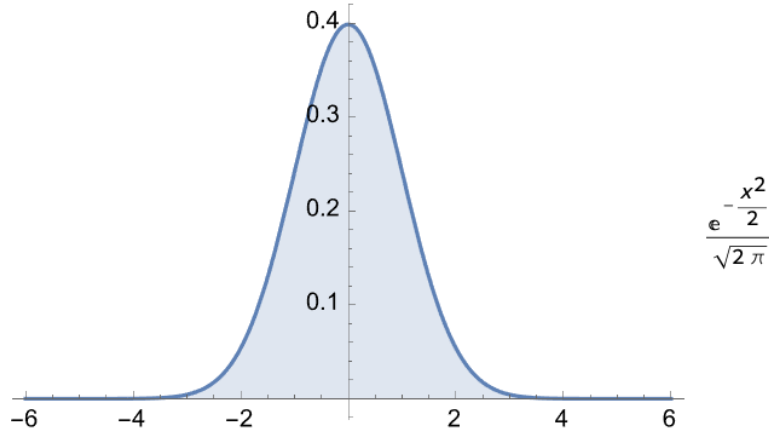
The Kernel of this Kind/Distribution K_X has several nice properties. It is symmetric around μ and thus has expectation μ . It is very smooth and decays towards zero very quickly for large $|x|$, remaining positive for all values.

The bigger σ^2 is, the more spread out the curve is. If X has a Normal $\langle 0, 1 \rangle$ Distribution and Y has a Normal $\langle \mu, \sigma^2 \rangle$, then the shapes are the same up to shifts and scaling:

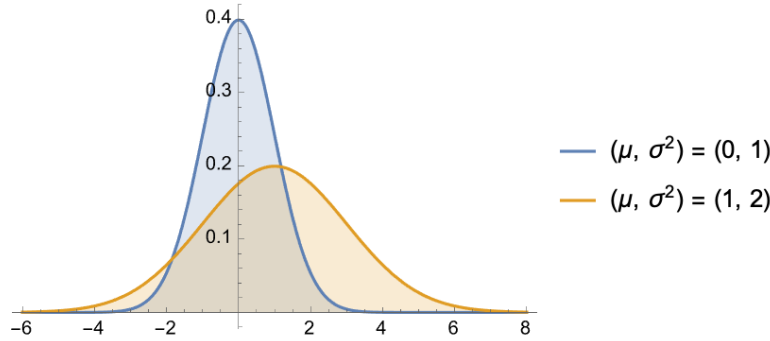
$$\frac{K_X(z)}{K_X(0)} = \frac{K_Y(\mu + \sigma z)}{K_Y(\mu)}.$$

See Figure 29.2. In fact, the σ^2 is the variance of a Normal $\langle \mu, \sigma^2 \rangle$ Distribution.

You might wonder why we use σ^2 rather than σ as the second parameter. Either convention could be used, but in practice it is more often useful to represent the

FIGURE 29.1. The Kernel of a Normal $\langle 0, 1 \rangle$ Distribution

variance than the standard deviation with its pesky square root. And this form generalize and this form will also generalize naturally to the vector Normal Distributions below.

FIGURE 29.2. Comparing the Kernels of a Normal $\langle 0, 1 \rangle$ and a Normal $\langle 1, 2 \rangle$ Distribution

Distributions in the Normal family are determined by two parameters, the expectation and variance. Once you specify those, the Distribution in the family is determined. So if you know that a random variable has a Normal Distribution, you can find it by finding the random variable's expectation and variance.

Suppose X has a scalar Normal $\langle \mu, \sigma^2 \rangle$ Distribution, and let $Y := a + bX$ with $b \neq 0$. We know $\mathbb{E}(Y) = \mathbb{E}(a + bX) = a + b\mathbb{E}(X) = a + b\mu$ and $\text{Var}(Y) = \text{Var}(a + bX) = b^2\text{Var}(X) = b^2\sigma^2$. We also know

$$\begin{aligned} D_Y(\varphi) &= D_X(\varphi \circ \langle x \rangle \mapsto a + bx) \\ &= \int_{x=-\infty}^{\infty} dx \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \varphi(a + bx) \end{aligned}$$

$$\begin{aligned}
&= \int_{x=-\infty}^{\infty} dx \frac{1}{|b|\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} b \varphi(a+bx) \\
&= \int_{y=-\infty}^{\infty} dy \frac{1}{|b|\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{(y-a)/b-\mu}{\sigma}\right)^2} \varphi(y) \\
&= \int_{y=-\infty}^{\infty} dy \frac{1}{|b|\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{(y-a-b\mu)}{|b|\sigma}\right)^2} \varphi(y),
\end{aligned}$$

via the change of variables $y \leftarrow a + bx$. This is the form of a Normal Distribution and therefore Y has a $\text{Normal}\langle a + b\mu, b^2\sigma^2 \rangle$ Distribution.

If $b < 0$, we do an additional sign change to get $|b|$ in the denominator.

If X has a $\text{Normal}\langle \mu, \sigma^2 \rangle$ Distribution, then $Y := a + bX$ with $b \neq 0$ has a Normal Distribution with expectation $a + b\mu$ and variance $b^2\sigma^2$.

Now if Z has a scalar $\text{Normal}\langle 0, 1 \rangle$ Distribution, this tells us that $X := \mu + \sigma Z$ has a $\text{Normal}\langle \mu, \sigma^2 \rangle$ Distribution. And conversely, if X has a $\text{Normal}\langle \mu, \sigma^2 \rangle$ Distribution, then $Z := \frac{X-\mu}{\sigma}$ has a $\text{Normal}\langle 0, 1 \rangle$ Distribution.

The $\text{Normal}\langle 0, 1 \rangle$ Distribution is by convention called the (scalar) **standard Normal Distribution**. It is often convenient to express our calculations with Normals in terms of the standard Normal. The CDF of any random variable with a $\text{Normal}\langle 0, 1 \rangle$ Distribution is a special function traditionally denoted by Φ (capital “phi”). That is, $\Phi(t) = F_Z(t) = \mathbb{E}(\{Z \leq t\})$ for Z with $\text{Normal}\langle 0, 1 \rangle$. There is no simple formula to compute Φ , so this is generally computed numerically in practice. Modern calculators will often have a special button for Φ or for a related function called erf (the “error function”) where $\Phi(t) = \frac{1}{2} + \frac{1}{2}\text{erf}(t/\sqrt{2})$.

If X has a scalar $\text{Normal}\langle \mu, \sigma^2 \rangle$ Distribution, then its CDF is given by

$$\begin{aligned}
F_X(t) &= \mathbb{E}(\{X \leq t\}) \\
&= \mathbb{E}(\{X - \mu \leq t - \mu\}) \\
&= \mathbb{E}\left(\left\{\frac{X - \mu}{\sigma} \leq \frac{t - \mu}{\sigma}\right\}\right) \\
&= \Phi\left(\frac{t - \mu}{\sigma}\right).
\end{aligned} \tag{29.2}$$

Here, we use the principle that **equivalent conditions mean equal events**. Because $\sigma > 0$, the inequality stays in the same direction in the third line. Equation (29.2) gives us the CDF of an scalar Normal random variable in terms of the scalar standard Normal.

The Normal family is actually well defined for any dimension. If X is an n -

dimensional $\text{Normal}\langle\mu, \sigma^2\rangle$, then μ is an n -dimensional *vector* and σ^2 is an $n \times n$ *matrix*.

Definition 63. A random variable X of dimension n has a $\text{Normal}\langle\mu, \sigma^2\rangle$ Distribution for n -vector μ and positive definite $n \times n$ matrix σ^2 when

$$D_X(\psi) = \int \cdots \int_x dx \frac{1}{\sqrt{\det 2\pi\sigma^2}} e^{-\frac{1}{2}(x-\mu)^T \sigma^{-2}(x-\mu)} \psi(x).$$

Here, $\det 2\pi\sigma^2$ is the determinant of the matrix $2\pi\sigma^2$, and σ^{-2} is the inverse matrix of σ^2 .

Then: $\mathbb{E}(X) = \mu$ and $\text{Var}(X) = \sigma^2$.

We often label the dimension in the name of the Distribution when it is not clear from context and call this the $\text{Normal}_n\langle\mu, \sigma^2\rangle$ Distribution. If the dimension is omitted and has not been specified elsewhere, we take the scalar case ($n = 1$) as the default.

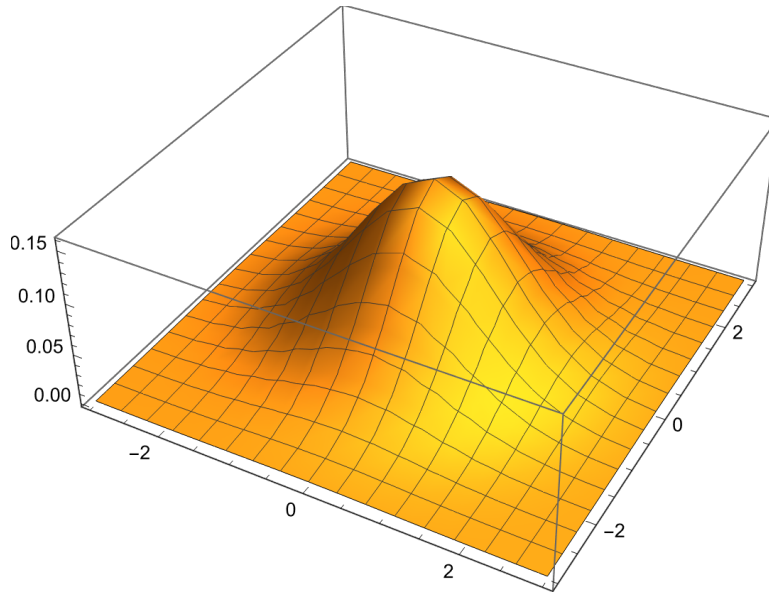
When $n = 1$, this reduces to the scalar definition in equation (29.1), with σ^2 a positive number.

When $n > 1$, $\sigma^2 = \text{Var}(X)$ is the **variance matrix** of the random variable X . The i, j th entry is the covariance of the random variables X_i and X_j , so $(\sigma^2)_i^j = \text{Cov}(X_i, X_j)$. This means that the diagonal entries $(\sigma^2)_i^i = \text{Var}(X_i)$ are just the variances of the corresponding component of X .

The matrix σ^2 is symmetric and positive definite, so it has both an inverse matrix σ^{-2} and a symmetric square-root σ this is also invertible. (When $n = 1$, this just means that σ^2 is a positive number, as we assumed before.)

The Kernel of two 2-dimensional Normal Distributions are shown in Figures 29.3 and 29.4 This has the same “bell curve” shape with potentially different spreads in different directions. The Kernel in Figure 29.4 exhibits non-zero covariance between the components, which we can see by the concentration of the density along a linear ridge.

The **standard Normal Distribution** $\text{Normal}_n\langle 0, I \rangle$ in higher dimensions mimics that in dimension one. The expectation is the zero vector, and the variance matrix is the identity matrix. Each component is thus a $\text{Normal}\langle 0, 1 \rangle$ scalar random variable. The different components have zero correlation and are in fact independent.

FIGURE 29.3. Standard $\text{Normal}_2\langle 0, I \rangle$ Kernel.

The components of the standard Normal Distribution are independent $\text{Normal}\langle 0, 1 \rangle$ random variables.

If X has a $\text{Normal}\langle \mu, \sigma^2 \rangle$ Distribution of any dimension, then

$$Z := \sigma^{-1}(X - \mu)$$

has a standard Normal Distribution (of the same dimension).

And if Z has a standard Normal Distribution of any dimension, then

$$X := \mu + \sigma Z$$

has a $\text{Normal}\langle \mu, \sigma^2 \rangle$ Distribution (of the same dimension).

Normal Distributions have many useful properties. These include the following:

- If X and Y are Normal random variables of the same dimension, then $X + Y$ is also Normal.
- If X is a Normal random variable of dimension n , then $a + bX$ is Normal, for n -vector a and scalar b .
- If X and Y are Normal random variables and $\text{Cov}(X, Y) = 0$, then X and Y are independent.
- If X and Y are Normal random variables, then the conditional Distribution of

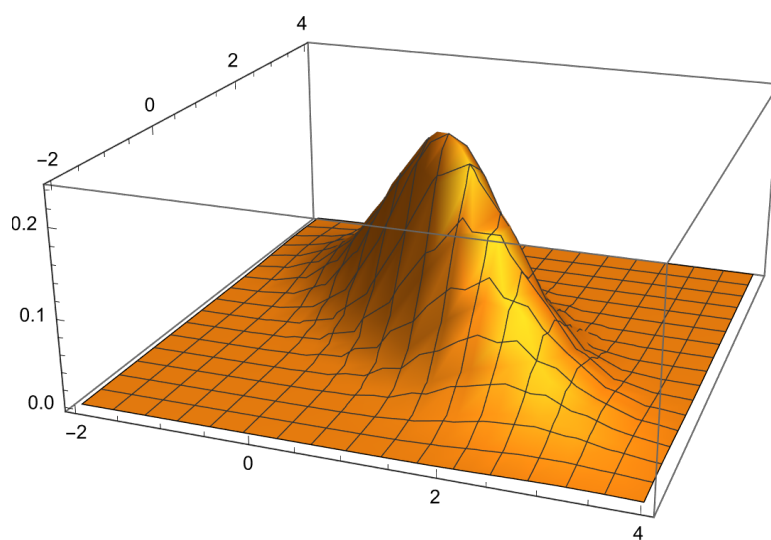


FIGURE 29.4. $\text{Normal}_2\langle\mu, \sigma^2\rangle$ Kernel where $\mu = \langle 1, 1 \rangle$ and $\sigma^2 = \begin{bmatrix} 1 & -\frac{3}{4} \\ -\frac{3}{4} & 1 \end{bmatrix}$.

X given $Y = y$ is Normal.

The Bernoulli Trials Process

30

Chapter

Contents

30.1 Counting the Number of Marks in a Fixed Number of Trials . .	1081
30.2 Waiting for Marks	1082
30.3 Counting Rare Marks	1086
30.4 Random Uniforms	1087
30.5 Random Walks	1088
30.6 Normals Everywhere	1088

Here we will study in detail a simple but important stochastic process called the **Bernoulli Trials Process**.

A **Bernoulli Trial** is a part of a random experiment that can turn out in one of two ways. Any random result with two states qualifies. Examples include: heads or tails, works or malfunctions, win or lose, succeed or fail, true or false, on or off, yes or no. To every Bernoulli trial, we associate an **indicator random variable**:

A trial for which the indicator is 1 is called a **mark**.

A trial for which the indicator is 0 is called a **space**.

The nomenclature comes from imagining an observer recording the outcome of several trials in a notebook, either making a mark or leaving a space in the notebook based on the outcome of each trial. The outcome associated with a mark is up to us to decide; we can use 1 for yes or no, up or down, heads or tails depending on what is most appropriate for the problem at hand.

A **BernoulliTrials $\langle p \rangle$ Process** for $0 \leq p \leq 1$ is a stochastic process $M = \langle M_1, M_2, M_3, \dots \rangle$ with index set $[1..]$, where M_n is the indicator of whether a mark occurs on trial n .

The random variables $M_1, M_1, M_2, \dots$ are assumed to be **independent and identically distributed** indicators with $\mathbb{E}(M_n) = p$.

Note that an indicator with expectation p has the canonical Bernoulli $\langle p \rangle$ Distribution.

By the definition, if M is a BernoulliTrials $\langle p \rangle$ process, then the Distribution of any finite sub-collection $M_n. = \langle M_{n_1}, \dots, M_{n_k} \rangle$ is determined using independence and the common marginal Distribution of each M_{i_j} . In particular,

$$D_{M_n.}(\psi) = \sum_{b \in \{0,1\}^k} p^{\sum_j b_j} (1-p)^{k-\sum_j b_j} \psi(b), \quad (30.1)$$

because

$$\begin{aligned} \mathbb{E}(\{M_{i_1} = b_1 \wedge \dots \wedge M_{i_k} = b_k\}) &= \mathbb{E}(\{M_{i_1} = b_1\}) \cdots \mathbb{E}(\{M_{i_k} = b_k\}) \\ &= p^{\sum_j b_j} (1-p)^{k-\sum_j b_j}, \end{aligned}$$

with $\sum_j b_j$ factors of p and the rest factors of $1-p$. As above, we will use $M_{j:k} = (M_j, \dots, M_k)$ to denote a finite *segment* of the process's evolution.

Aside. The BernoulliTrials $\langle p \rangle$ process has a useful “reproducing property.” At each time k , the future of the process looks probabilistically identical to a BernoulliTrials $\langle p \rangle$ process.

For any sequence $s = \langle s_1, s_2, \dots \rangle$ and any integer $k \geq 0$, define the sequence

$$\text{shift}(s, k) = \langle s_k, s_{k+1}, \dots \rangle.$$

If M is a BernoulliTrials $\langle p \rangle$ process, then M and $\text{shift}(M, k)$ have the **same Distribution** for all integers $k \geq 0$.

This fact comes from the independence and identical Distributions of the M_n 's.

The BernoulliTrials $\langle p \rangle$ process has a simple structure: all the random variables are independent indicators with a common expectation. Yet the process has a wide variety of applications. In the following sections, we explore some of these. Throughout, M will be a BernoulliTrials $\langle p \rangle$ process.

30.1 Counting the Number of Marks in a Fixed Number of Trials

Define $N = \sum_{k=1}^n M_k$. This random variable *counts the number of marks in a fixed number n of trials*. Then, $N = \psi(M_{1:n})$, where the compatible statistic ψ is given by

$$\psi(b_1, \dots, b_n) = b_1 + \dots + b_n,$$

and by equation (30.1),

$$D_N(\varphi) = D_{M_{1:n}}(\varphi \circ \psi) \quad (30.2)$$

$$= \sum_{b \in \{0,1\}^n} p^{b_1 + \dots + b_n} (1-p)^{n-(b_1 + \dots + b_n)} \varphi(b_1 + \dots + b_n) \quad (30.3)$$

$$= \sum_{k=0}^n \binom{n}{k} p^k (1-p)^{n-k} \varphi(k). \quad (30.4)$$

The last equality follows because there are exactly $\binom{n}{k}$ binary vectors $b \in \{0,1\}^n$ for which $b_1 + \dots + b_n = k$.*

*To see why, note that each such vector determines a unique subset of $[1 \dots n]$ where k is included if $b_k = 1$.

The number of marks in a fixed number n of Bernoulli(p) trials has a **Binomial** $\langle n, p \rangle$ **Distribution**:

$$D_N(\varphi) = \sum_{k=0}^n \varphi(k) \binom{n}{k} p^k (1-p)^{n-k}.$$

We have $\mathbb{E}(N) = np$ and $\text{Var}(N) = np(1-p)$.

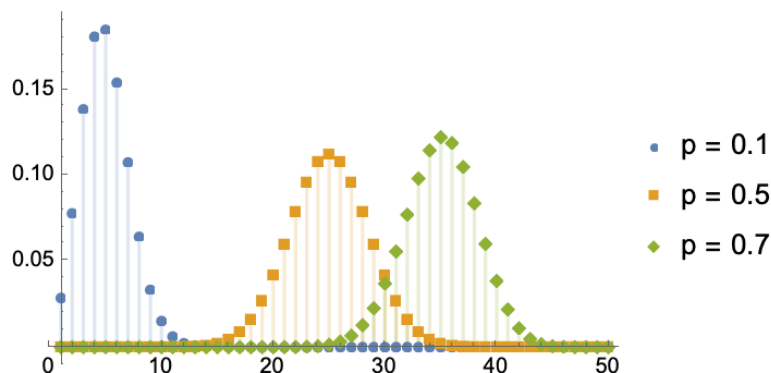


FIGURE 30.1. The probability mass functions from several Binomial $\langle n, p \rangle$ Distributions

30.2 Waiting for Marks

Let W_r be the number of Bernoulli Trials we have to wait up to and including the r^{th} mark, for integer $r \geq 1$.

There are several ways we can find the Distribution of W_r . To start, consider the event $\{W_r = n\}$. Define the compatible statistic ψ_n such that $\{W_r = n\} := \psi_n(M_{1:n})$. Notice that $\psi_n(b_1, \dots, b_n)$ equals 1 if $b_n = 1$ and $\sum_{j=1}^{n-1} b_j = r - 1$; there must be exactly r marks ending with a mark. The last trial must be 1 as n is the time at which we first see the r th mark. So,

$$\mathbb{E}(\{W_r = n\}) = D_{M_{1:n}}(\psi_n) \quad (30.5)$$

$$= \sum_{b \in \{0,1\}^n} p^{b_1 + \dots + b_n} (1-p)^{n-(b_1 + \dots + b_n)} \psi_n(b_1, \dots, b_n) \quad (30.6)$$

$$= p^r (1-p)^{n-r} \left[\sum_{b \in \{0,1\}^n} \psi_n(b_1, \dots, b_n) \right] \quad (30.7)$$

$$= \binom{n-1}{r-1} p^r (1-p)^{n-r}, \quad (30.8)$$

because $b_1 + \dots + b_n = r$ whenever $\psi_n(b_1, \dots, b_n) = 1$ by definition and because the sum in $\square$'s just counts the number of sequences for which ψ_n equals 1. There are exactly $\binom{n-1}{r-1}$ binary sequences of length n that end in 1 and have $r-1$ 1's in the first $n-1$ slots.*

*Again, each binary sequence of length n corresponds to a subset of $[1 \dots n]$, so we are just counting subsets that contain n and $r-1$ other elements.

The number of trials until we first see the r^{th} mark for integer $r \geq 1$ has a **NegativeBinomial** $\langle r, p \rangle$ **Distribution** given by

$$D_{W_r}(\psi) = \sum_{n=r}^{\infty} \binom{n-1}{r-1} p^r (1-p)^{n-r} \psi(n).$$

Note that W_r includes the trial in which we see the r^{th} mark. Here, $\mathbb{E}(W_r) = \frac{r}{p}$ and $\text{Var}(W_r) = r \frac{1-p}{p^2}$.

The special case $r = 1$ is called the Geometric $\langle p \rangle$ Distribution.

The name “NegativeBinomial” is not a wonderful name. It is not, as the name suggests, the negative of a binomial Distribution. Instead, it is a sort of dual to the Binomial Distribution. The Binomial $\langle n, p \rangle$ Distribution describes the number of marks in a fixed number of trials; the NegativeBinomial $\langle r, p \rangle$ describes the number of

trials to get a fixed number of marks.

Example 30.1. Suppose W has a Geometric(p) Distribution, find $\mathbb{E}(\{W > k\})$.

Notice that $\{W > k\} = \sum_{n=k+1}^{\infty} \{W = n\}$, and at most one of the events on the right can be 1. So the Monotone Limits Rule applies and we can write

$$\begin{aligned}\mathbb{E}(\{W > k\}) &= \sum_{n=k+1}^{\infty} \mathbb{E}(\{W = n\}) \\ &= \sum_{n=k+1}^{\infty} p(1-p)^{n-1} \\ &= p(1-p)^k \sum_{n=0}^{\infty} (1-p)^n \\ &= p(1-p)^k \frac{1}{1-(1-p)} \\ &= (1-p)^k.\end{aligned}$$

Example 30.2 **Example 30.2. The Geometric(p) is Memoryless** Suppose W has a Geometric(p) Distribution, show that

$$\mathbb{E}(\{W > r+k\} \mid W > r) = \mathbb{E}(\{W > k\})$$

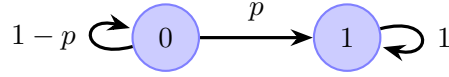
In words, the probability that we wait more than $r+k$ trials for the first mark given that we have already waited more than r trials, is the same as the probability that we wait more than k trials originally. The Geometric Distribution *does not remember* that we have already waited.

From the updating equation,

$$\begin{aligned}\mathbb{E}(\{W > r+k\} \mid W > r) &= \frac{\mathbb{E}(\{W > r+k \wedge W > r\})}{\mathbb{E}(\{W > r\})} \\ &= \frac{\mathbb{E}(\{W > r+k\})}{\mathbb{E}(\{W > r\})} \\ &= \frac{(1-p)^{r+k}}{(1-p)^r} \\ &= (1-p)^k \\ &= \mathbb{E}(\{W > k\}).\end{aligned}$$

Another approach to finding the form of the Negative Binomial Distribution is

conditioning. Let's start with $r = 1$. We envision a system with two states as follows.



W_n is the first time the system reaches state 1, starting in state 0. We will compute $\mathbb{E}(\{W_1 = n\})$ for $n \in [1..]$ by conditioning on the first trial, M_1 .

$$\begin{aligned}\mathbb{E}(\{W_1 = n\}) &= \mathbb{E}(\{M_1 = 0\}) \mathbb{E}(\{W_1 = n\} \mid M_1 = 0) \\ &\quad + \mathbb{E}(\{M_1 = 1\}) \mathbb{E}(\{W_1 = n\} \mid M_1 = 1) \\ &= (1 - p) \mathbb{E}(\{W_1 = n - 1\}) + p \{n = 1\}.\end{aligned}$$

Hence, $\mathbb{E}(\{W_1 = n\}) = p(1 - p)^{n-1}$ as before.*

For $r > 1$, we can compute $\mathbb{E}(\{W_r = n\})$ for $n \in [r..]$ by conditioning on W_1 .

$$\mathbb{E}(\{W_r = n\}) = \sum_{m=1}^{n-r+1} \mathbb{E}(\{W_1 = m\}) \mathbb{E}(\{W_r = n\} \mid W_1 = m) \quad (30.9)$$

$$= \sum_{m=1}^{n-r+1} p(1 - p)^{m-1} \mathbb{E}(\{W_{r-1} = n - m\}). \quad (30.10)$$

For example, for $r = 2$

$$\begin{aligned}\mathbb{E}(\{W_2 = n\}) &= \sum_{m=1}^{n-1} p(1 - p)^{m-1} \mathbb{E}(\{W_1 = n - m\}) \\ &= \sum_{m=1}^{n-1} p(1 - p)^{m-1} p(1 - p)^{n-m-1} \\ &= p^2(1 - p)^{n-2} \sum_{m=1}^{n-1} 1 \\ &= \binom{n-1}{1} p^2(1 - p)^{n-2}.\end{aligned}$$

*We have a recurrence with
 $p_1 = p$ and
 $p_n = p_{n-1}(1 - p)$ for $n > 1$.
 Note that
 $\mathbb{E}(\{W_1 = 0\}) = 0$.

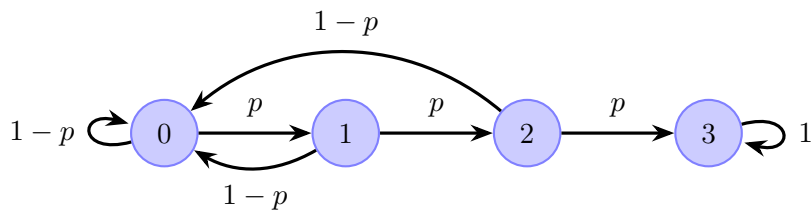
For $r = 3$,

$$\begin{aligned}
 \mathbb{E}(\{W_3 = n\}) &= \sum_{m=1}^{n-2} p(1-p)^{m-1} \mathbb{E}(\{W_2 = n-m\}) \\
 &= \sum_{m=1}^{n-2} p(1-p)^{m-1} (n-m-1)p^2(1-p)^{n-m-2} \\
 &= p^3(1-p)^{n-3} \sum_{m=1}^{n-2} (n-m-1) \\
 &= \binom{n-1}{2} p^2(1-p)^{n-2}.
 \end{aligned}$$

Which is exactly the form we found above. Plugging that suggested form into equation (30.10), we find the expected formula solves the recurrence, giving us the same probabilities as above.

Similar logic can work in many different situations. For example, let T be the number of trials we have to wait until you first see three marks in a row. What is the Distribution of T ? What is its expectation?

Think of this system as an finite-state machine:



T is the number of trials we wait until first hitting state 3, and let S be the initial state of the machine. Let's start with the expectation of T . Define $t_s = \mathbb{E}(T \mid S = s)$ for $n \in [3..]$ and condition on M_1 . Note that $t_3 = 0$ and we want t_0 . We have

$$\begin{aligned}
 t_s &= (1-p) \mathbb{E}(T \mid S = s \wedge M_1 = 0) + p \mathbb{E}(T = n \mid S = s \wedge M_1 = 1) \\
 &= (1-p)(1 + t_0) + p(1 + t_{s+1}) \\
 &= 1 + (1-p)t_0 + pt_{s+1}.
 \end{aligned}$$

So,

$$\begin{aligned}
 t_0 &= 1 + (1-p)t_0 + pt_1 \\
 t_1 &= 1 + (1-p)t_0 + pt_2 \\
 t_2 &= 1 + (1-p)t_0
 \end{aligned}$$

$$\begin{aligned}
t_1 &= 1 + (1 - p)t_0 + p + p(1 - p)t_0 \\
&= 1 + p + (1 - p)(1 + p)t_0 \\
t_0 &= 1 + (1 - p)t_0 + p(1 + p) + p(1 + p)(1 - p)t_0 \\
t_0 &= \frac{1 + p(1 - p)}{1 - (1 - p)(1 + p + p^2)}.
\end{aligned}$$

So,

$$\mathbb{E}(T \mid S = 0) = \frac{1 + p(1 - p)}{1 - (1 - p)(1 + p + p^2)}.$$

Now, define $p_{n,s} = \mathbb{E}(\{T = n\} \mid S = s)$. We want $p_{n,0}$ for all $n \in [3..]$. For instance, we have $p_{n,3} = \{n = 0\}$ and for $s \in [0..2]$ and $n > 3$:

$$\begin{aligned}
p_{n,s} &= (1 - p) \mathbb{E}(\{T = n\} \mid S = s \wedge M_1 = 0) + p \mathbb{E}(\{T = n\} \mid S = s \wedge M_1 = 1) \\
&= (1 - p)p_{n-1,0} + pp_{n-1,s+1}.
\end{aligned}$$

This can be reduced to

$$\begin{aligned}
p_{n,3} &= \{n = 0\} \\
p_{n,2} &= (1 - p)p_{n-1,0} \\
p_{n,1} &= (1 - p)(1 + p)p_{n-1,0} \\
p_{n,0} &= (1 - p)p_{n-1,0} + p(1 - p)(1 + p)p_{n-2,0} + p^3\{n = 3\},
\end{aligned}$$

for $n \in [3..]$ where $p_{2,0} = 0 = p_{1,0}$. So $p_{4,0} = p^3(1 - p)$, $p_{5,0} = p^4(1 - p)(1 + p) + p^3(1 - p)^2$, and so forth. Recurrence like the last equation can be solved exactly, and similar solutions generated for any pattern of 1s and 0s that we are waiting to see.

30.3 Counting Rare Marks

Now we imagine a turbo-charged BernoulliTrials(p) process where the first n trials occur within a fixed amount of time (call it 1 second). We let the number of trials increases and the probability of a mark decreases: $n \rightarrow \infty$ and $np \rightarrow \lambda$ for some fixed $\lambda > 0$. The number of marks in that one-second window is a count of “rare” events (because p is very small) that occur at a constant “rate” λ per unit time.

Let N be the number of marks in the one-second window. We know that for $k \in [0..n]$

$$\mathbb{E}(\{N = k\}) = \binom{n}{k} p^k (1 - p)^{n-k},$$

but as n grows large and np gets closer to λ , this expression converges to

$$\mathbb{E}(\{N = k\}) = e^{-\lambda} \frac{\lambda^k}{k!},$$

for each $k \in [0..].$

A random variable N with the **Poisson(λ) Distribution** is often used to model a count of rare events. The Distribution is given by

$$D_N(\psi) = \sum_{k=0}^{\infty} e^{-\lambda} \frac{\lambda^k}{k!} \psi(k), \quad (30.11)$$

with $\mathbb{E}(N) = \lambda$ and $\text{Var}(N) = \lambda$.

30.4 Random Uniforms

Define $U = \sum_{k=1}^{\infty} 2^{-k} M_k$.

Puzzle 175. When expand a number in a base (like 10 or 2), we express it as a sum of digits times powers of the base. This applies to positive powers (to the left of the decimal point) and negative powers to the right of the decimal point.

Explain how the definition of U above can be seen as a number in (0_1) expressed in binary with digits $M_1, M_2, M_3, \dots$

Now let $p = 1/2$ for the Bernoulli Trials process. Find $\mathbb{E}(\{j2^{-k} < U < (j+1)2^{-k}\})$ for any $j \in [0..2^k)$.

Notice that j can be expanded in k -binary digits, $j = \sum_{i=0}^{k-1} j_i 2^i$. Then,

$$\{j2^{-k} < U < (j+1)2^{-k}\} = \left\{ \bigwedge_{i=0}^{k-1} M_i = j_i \right\}.$$

Because the M_i 's are IID with expectation $1/2$,

$$\mathbb{E}(\{j2^{-k} < U < (j+1)2^{-k}\}) = \mathbb{E}\left(\left\{ \bigwedge_{i=0}^{k-1} M_i = j_i \right\}\right) = 2^{-k}.$$

Hence, U lies in any “dyadic” interval of length 2^{-k} with probability 2^{-k} . And as a result, we can show that U has a Uniform $\langle 0, 1 \rangle$ Distribution.

30.5 Random Walks

We can generate a new stochastic process S from M by defining $S_0 = 0$ and

$$S_n = \sum_{k=1}^n (2M_k - 1). \quad (30.12)$$

All integers, positive and negative, are possible states of this process. The state increases by one whenever we see a mark and decreases by one whenever we see a space.

This process is called a **random walk**.

30.6 Normals Everywhere

A fundamental property of the Normal Distributions that we will study is that sums of independent random variables, suitably shifted and scaled, have a Distribution that can be closely approximated by a Normal Distribution.

A simple version of that can be seen in the BernoulliTrials(p) process. Let N be the number of marks in a fixed number of n trials, which we have seen has a Binomial(n, p) Distribution.

Define

$$Z_n = \frac{N - np}{\sqrt{np(1-p)}} = \frac{N - \mathbb{E}(N)}{\text{SD}(N)}.$$

Hence, $\mathbb{E}(Z_n) = 0$ and $\text{Var}(Z_n) = 1$. As n grows larger and larger, with p fixed, we have for every bounded and continuous compatible statistic ψ that accepts real values as input:

$$D_{Z_n}(\psi) \approx D_Z(\psi) \quad (30.13)$$

where Z has a standard Normal($0, 1$) Distribution.

Thus, we can approximately compute our predictions about N by using predictions from a Normal Distribution.

Part III

Approximations

The Joy of Approximation

31

Chapter

Contents

31.1 Approximating Distributions	1093
31.2 Approximating Sums of Random Variables	1097
31.3 Asymptotic Order of the Remainder	1100
31.4 Review: Three Important Canonical Distributions	1101

Key Take Aways

An approximate answer to the right problem is worth a good deal more than an exact answer to an approximate problem.

- John Tukey

Often, computing an exact answer to our problem is hard or infeasible, and sometimes we do not need an exact answer to achieve our goal. In these cases, we can find an *approximate* answer to the problem. We can use several different types of approximations in our analysis:

- Mathematical Approximations – construct simpler replacements for our expressions that give approximately the same numerical answers.

Examples: $\sin(\theta) \approx \theta$ for $|\theta|$ small;

Stirling's Approximation ($\ln n! \approx n \ln n - n$ for large n)

- Inequalities – get a bound (upper, lower, or both) on the quantity of interest

Examples: Markov, Chebychev, Chernoff and Hoeffding Inequalities

- Approximate the Distributions we are working with

Here, we will focus on the last of these.

It happens every day — in classrooms, in dorms, in faculty offices, in coffee shops, and on the couch in front of the TV — foreheads banging into metaphorical walls, a slight rush of blood in the ears, and a compulsive focus on a calculation that refuses to be tamed. The unfortunate fact is that even for relatively simple models we often cannot express the quantity of interest in a nice form. There is no shame in that.

Faced with an untamed calculation, you have two choices: work harder to find an exact solution or derive an approximate answer that is accurate enough to be useful. Which of these to choose must be dictated by how the result will be used and how much accuracy is needed to achieve the goal.

Working harder is often a good option because a seemingly intractable problem can become as soft and tasty as butter when viewed from a new perspective. It takes practice to learn to see a problem in a new way and to generate alternate solution strategies, but as your skills develop, you will face no shortage of directions in which to search. The trick, then, is to know when to stop the hunt, to avoid becoming Ahab in relentless pursuit of an objective that does not serve the original goal. Sometimes a calculation is intrinsically so difficult that no simplification is possible. Sometimes the extra accuracy provided by an exact solution is more than is necessary to solve a problem. And, sometimes an exact solution is so complex that it offers little insight. In all these cases, an approximation can help. As we will see, it is often a good strategy to *begin* with an approximation because a quick approximate answer can offer guidance and insight on subsequent steps.

Approximation entails a compromise between accuracy and effort. We accept some level of error in our answer in exchange for being able to find an answer at all, or being able to find an answer more quickly, or being able to understand the answer once we find it.

Using approximation effectively requires balancing both sides of this compromise. First, one needs to assess the accuracy of an approximate expression and to determine when the accuracy meets one's requirements. Second, one needs to determine what those requirements for accuracy should be. We perform any calculation to serve some overarching, global goal, such as answering a question, making a decision, or proving a theorem. Work on the calculation itself is driven by the local goal: finding the result in the simplest form possible. The local goal demands exactness, but the global goal often does not require it. An effective approximation will minimize the effort needed to get sufficient accuracy for the task at hand.

One of the most powerful and commonly used methods for generating approximations is *asymptotic analysis*. We choose some feature of a calculation (e.g., the number of terms n to use in a sum) and consider the result in the limit as that feature goes to some extreme value (e.g., as $n \rightarrow \infty$). Rather than averaging five, or fifty, or five hundred random variables, we imagine averaging longer and longer sequences. Instead of considering a small probability, we imagine the probability shrinking away to nothing. Limits tend to eliminate the messy details and make the result easier to use and understand. When, in a calculation, we replace an expression with its limiting value, we derive an *asymptotic approximation*. Such an approximation will be good when the feature in practice is extreme “enough” that the expression will be close to its limiting value. The key question, then, is when is it extreme “enough”? The answer, as above, depends on the error rate for the approximation and how that aligns with one’s goals.

The main subject of these notes is a trio of classic asymptotic approximations that have proved useful in a wide variety of situations. We call these: the **Constant Approximation**, the **Normal Approximation**, and the **Poisson Approximation**. These three approximations are usually stated as limit theorems and given the catchier names: the **Law of Large Numbers**, the **Central Limit Theorem**, and the **Law of Rare Events**. The first two are among the most important results in all of mathematics. We will look at a practical version here and save the details until later.

31.1 Approximating Distributions

Many of the problems we face in probability require that we compute the Distribution of some random variables and use that Distribution to calculate an interesting quantity. But what happens when the Distribution or quantity we need is too difficult to compute, when the sums or integrals do not work out nicely, when the usual tricks all fail?

Alas, this happens more often than we would like. Even simple models can produce complex calculations. One effective response to this problem is to *approximate* the target Distribution with another Distribution that is easier to compute and that gives *almost* the same answers. When all goes well, the approximate Distribution can then be used in place of the target without significant loss of accuracy in our predictions and decisions.

A probability Distribution D describes our best predictions of the output of algorithms that take a random variable’s value as input. (The predictions are $D(\psi)$

for each statistic ψ .) An approximating Distribution $\tilde{D}$ gives predictions that are close to the predictions of the first Distribution for each algorithm. That is, $D(\psi) \approx \tilde{D}(\psi)$ for compatible statistics ψ .

Another way to think about the Distribution is as a way to spread a unit of probability across a set of values (as described by its kernel K). An approximating Distribution (with kernel $\tilde{K}$ spreads the probability in nearly the same places, in nearly the same amounts, so $K \approx \tilde{K}$.

Suppose X is a random variable with Distribution D_X . We will say that a random variable $\tilde{X}$ *approximates* X , denoted $\tilde{X} \approx X$, when

$$D_{\tilde{X}}(\psi) \approx D_X(\psi) \quad (31.1)$$

for (at least) bounded and continuous statistic's. Or put another way

$$\mathbb{E}(\psi(\tilde{X})) \approx \mathbb{E}(\psi(X)). \quad (31.2)$$

Thus:

We say that $\tilde{X}$ approximates X when our predictions about $\tilde{X}$ are close to our predictions about X .

Keep in mind that equation (31.1) describes an approximation of *Distributions* not of random variables. We express the condition in terms of random variables because it is convenient, as we usually work in terms of random variables. But keep in mind that this is a statement about *predictions* not *values*; X and $\tilde{X}$ need not have the same value – or even be close. Indeed, $\tilde{X}$ need not even be an actual measurement in this particular random experiment. *

Example 31.1. Consider a random variable U that has a Uniform $\langle 0, 1 \rangle$ Distribution, with

$$D_U(\psi) = \int_{x=0}^1 dx \psi(x).$$

Its Kernel, the probability density function $K_U(x) = \{0 \leq x \leq 1\}$, is constant on the interval $[0, 1]$. In this example, we will define a random variable V_n that approximates U , and the larger we set n , the better the approximation.

Pick a positive integer n . The Distribution of U smears probability uniformly over each interval (within the unit interval); we will create a new Distribution by

concentrating all the probability each small interval of length $\frac{1}{n}$ down to one or two nearby points. Specifically, we will put all the probability that D_U assigns to the interval $[k/n, (k+1)/n)$ at point k/n , for each $k \in [0..n)$. This can be expressed in terms of a random variable V_n defined by

$$V_n = \frac{1}{n} \lfloor nU \rfloor.$$

V_n gives the nearest multiple of $1/n$ that is no greater than U . Then, for $k \in [0..n)$,

$$\begin{aligned} \mathbf{K}_{V_n}(k/n) &= \mathbb{E} \left(\left\{ \frac{k}{n} \leq U < \frac{(k+1)}{n} \right\} \right) \\ &= \int_{x=\frac{k}{n}}^{\frac{k+1}{n}} dx \, \mathbf{K}_U(x) \\ &= \frac{k+1}{n} - \frac{k}{n} = \frac{1}{n}. \end{aligned}$$

Hence,

$$\begin{aligned} D_{V_n}(\psi) &= \sum_{k=0}^{n-1} \frac{1}{n} \psi \left(\frac{k}{n} \right) \\ D_U(\psi) &= \int_{x=0}^1 dx \, \psi(x). \end{aligned}$$

You might recognize this sum from calculus as the Riemann sum approximation for the integral. When the function ψ varies smoothly (i.e., does not wiggle or jump too wildly), the Riemann sum is close to the integral. And in that case, $|D_{V_n}(\psi) - D_U(\psi)| \leq \frac{1}{n}$, and the error vanishes as n grows.

To see this another way, the two Distributions D_{V_n} and D_U assign nearly the same probability to any interval. In particular, any interval $\mathcal{I} \subseteq [0,1]$ of length a . must contain at least $\lfloor na \rfloor$ and at most $\lceil na \rceil$ points in $\text{range}(V_n)$. Hence,

$$\frac{1}{n} \lfloor na \rfloor \leq \sum_{\frac{k}{n} \in \mathcal{I}} \mathbf{K}_{V_n} \left(\frac{k}{n} \right) \leq \frac{1}{n} \lceil na \rceil.$$

Because $\lfloor na \rfloor/n \leq a \leq \lceil na \rceil/n$ and $a = \int_I dx \, K_U(x)$, it follows that

$$\left| \sum_{\frac{k}{n} \in \mathcal{I}} K_{V_n} \left(\frac{k}{n} \right) - \int_{\mathcal{I}} dx \, K_U(x) \right| \leq \frac{1}{n},$$

or equivalently

$$|\mathbb{E}(\{V_n \in \mathcal{I}\}) - \mathbb{E}(\{U \in \mathcal{I}\})| \leq \frac{1}{n}.$$

As we make n larger, the approximation gets better and better.

Notice that we are approximating a continuous Distribution by a discrete Distribution here. The approximation still works as advertised, but if you were interested in a better approximation, some care would be needed. Consider the approximation of the CDFs, F_U by F_{V_n} , and look on the interval $[k/n, (k+1)/n)$ for $k \in [0..n)$. Because V_n is discrete, $F_{V_n}(t) = k/n$ for all t in that interval, but $F_U(t) = t$. We are approximating a line here by a constant, but the line and constant meet at the left, so $F_U(t) - F_{V_n}(t)$ gets as large as $1/n$ at the right endpoint. The approximation error shrinks to zero as $n \rightarrow \infty$, but the approximation would be better if we approximate the line by a constant that met it in the middle. One way to achieve this is to shift the approximate Distribution, and specifically to shift the CDF of V_n rightward by $1/2n$: the random variable $W_n = V_n + \frac{1}{2n}$ gives a worst-case error of $1/2n$, half as big as before. This correction is called a “continuity correction” and sometimes can markedly improve approximation accuracy.

Many of the approximations we will use in practice share a feature with this example. Instead of a single approximate Distribution, we construct a family of Distributions that vary in how close they are to the target. In the example, the parameter n indexes the Distributions of V_n . Like turning a knob, we can sharpen the approximation’s accuracy by increasing n or coarsen it by decreasing n . For instance, as n increases, the CDFs of V_n and U get closer and closer together, as do $\mathbb{E}(\psi(V_n))$ and $\mathbb{E}(\psi(U))$ for compatible statistics ψ . And in the limit of extreme values for the parameter, $n \rightarrow \infty$ in this case, the approximation error reaches 0. It is in that sense that we would say, for instance, that the Distribution of V_n converges to a Uniform $\langle 0, 1 \rangle$ as n goes to ∞ . Approximation results are often written as such “limit theorems,” as we will see.

31.2 Approximating Sums of Random Variables

Key Take Aways

Here we consider approximations that have a similar structure, with ingredients:

1. Random variables $X_1, X_2, X_3, \dots$ that are **independent and identically distributed** with $\mathbb{E}(X_i) = \mu$ and $\text{Var}(X_i) = \sigma^2$ where $0 < \sigma^2 < \infty$.
2. Random variables $S_1, S_2, S_3, \dots$ defined by

$$S_n = \sum_{i=1}^n X_i. \quad (31.3)$$

3. An approximation $D_{S_n} \approx D_{A_n}$, or with random variables $S_n \approx A_n$.

We study three major approximation results that work for “large” n :

1. The Constant Approximation

$$S_n \approx n\mu, \quad (31.4)$$

or equivalently

$$\frac{S_n}{n} \approx \mu. \quad (31.5)$$

2. **The Normal Approximation** There exists a $\text{Normal}\langle 0, 1 \rangle$ distributed random variable Z such that

$$S_n \approx n\mu + \sqrt{n}\sigma Z, \quad (31.6)$$

or equivalently,

$$\frac{S_n - n\mu}{\sigma\sqrt{n}} \approx Z. \quad (31.7)$$

3. **The Poisson Approximation** When the X_i s are indicators and $\mu = \mathbb{E}(\{X_i = 1\})$ is small,

$$S_n \approx K, \quad (31.8)$$

for a random variable K with a $\text{Poisson}\langle n\mu \rangle$ Distribution.

These results are commonly stated as **limit theorems**, where they are known, respectively, as

1. the **Law of Large Numbers**,
2. the **Central Limit Theorem**, and
3. the **Law of Rare Events**.

The first two in particular are among the most important theorems in mathematics and statistics.

We will focus here on approximating the Distributions of sums of independent and identically distributed random variables. Sums of independent variables are good targets for approximations for three reasons. First, except in very special cases, their Distributions are hard to compute, or at least hard to compute with. For instance, if X_1, X_2, X_3 are independent, continuous random variables, then $X_1 + X_2 + X_3$ has Distribution

$$D_S(\psi) = \int_t dt \int_u du \int_w dw \, \mathcal{K}_{X_1}(t-u) \mathcal{K}_{X_2}(u-w) \mathcal{K}_{X_3}(w) \psi(t).$$

That's a bit nasty, and that's only three variables. Consider the Distribution of the sum of 100 or 1000 or 10,000. Second, such sums arise frequently in problems. Many random quantities of interest can be decomposed as a sum of random contributions from different sources, although the summation structure is often obscured. For instance, your height, the value of your portfolio, and the damage suffered by a critical component in your computer are all the sum of many smaller random contributions (cellular growth events, price fluctuations, and physical shocks) over time. Third, such sums are amenable to approximation under weak conditions. While the individual random terms in sum may have their own idiosyncracies, those idiosyncracies tend to cancel each other out when enough of them are added together. It is like looking at a large crowd from a distance; however unique are the individuals that comprise it, the crowd looks homogeneous.

For the remainder of these notes, unless specifically noted, we will make a few consistent assumptions:

1. $X_1, X_2, X_3, \dots$ are *independent* real-valued random variables with $\mathbb{E}(X_i^2) < \infty$.

We will actually assume slightly more: that the X_i 's are identically distributed with $\mathbb{E}(X_i) = \mu$ and $\text{Var}(X_i) = \sigma^2 > 0$.

2. Define $S_n = \sum_{i=1}^n X_i$ be the sum of the first n X_i 's.
3. We will look at approximations for the Distribution of S_n for “large” values of n . (Or equivalently, we will consider the limiting Distribution of S_n as $n \rightarrow \infty$.)

We will focus on three main results:

- The **Constant Approximation** tells us that for large n , the Distribution of S_n has most of its probability concentrated close to the single point $n\mu$. In other words, the Distribution of S_n is approximately the Distribution of the *constant* $n\mu$. And in this case, because the Distribution of a constant has all its mass at one point, we can say that $S_n \approx n\mu$.

This means, equivalently, that $S_n/n \approx \mu$. Our approximation works for sums *or* averages.

Notice that the constant here is just the expectation of S_n or S_n/n , respectively. The constant approximation says that for large enough n , S_n and S_n/n are well approximated by their expectations.

- The **Normal Approximation** tells us that for large n , the Distribution of S_n is approximately a Normal Distribution with the same expectation and variance as S_n . A corresponding statement can also be made about the average S_n/n . Specifically, we have $\mathbb{E}(S_n) = n\mu$ and $\text{Var}(S_n) = n\sigma^2$, and $\mathbb{E}(S_n/n) = \mu$ and $\text{Var}(S_n/n) = \sigma^2/n$. The Normal approximation tells us that

$$S_n \text{ has approximately a Normal } \langle n\mu, n\sigma^2 \rangle \text{ Distribution}$$

$$\frac{S_n}{n} \text{ has approximately a Normal } \langle \mu, \sigma^2/n \rangle \text{ Distribution.}$$

The Normal Approximation is a *direct generalization* of the Constant Approximation. In particular, it tells us the error in the Constant Approximation.

- The **Poisson Approximation** tells us that for large n and *small* μ (in a sense to be described), S_n has approximately a Poisson Distribution with the same expectation as S_n .

The Poisson Approximation applies in a case where the Normal approximation is less accurate and thus gives us a broader scope for getting a good approximation.

31.3 Asymptotic Order of the Remainder

Key Take Aways

When we have a sequence of random variables, it is useful to characterize, in rough terms, something about the typical magnitude of the random variables. We describe the “stochastic order of magnitude” as a function of the number of terms in the sum, n . This is “big oh P” and “little oh P” notation.

If Y_n is a sequence of random variables, we say that $Y_n = O_P(a_n)$ as $n \rightarrow \infty$, “**big Oh P**”, when

For every $\epsilon > 0$, there is a constant $c > 0$ and an integer $n_0 > 0$ such that

$$\mathbb{E}(\{|Y_n| \leq c|a_n|\}) \geq 1 - \epsilon,$$

for every $n \geq n_0$.

This means that $|Y_n/a_n|$ is bounded with high probability for large n .

If Y_n is a sequence of random variables, we say that $Y_n = o_P(a_n)$ as $n \rightarrow \infty$, “**little Oh P**”, when

$$\lim_{n \rightarrow \infty} \mathbb{E}(\{|Y_n| \leq c|a_n|\}) = 1,$$

for every $c > 0$. This means that $|Y_n/a_n|$ is nearly 0 with high probability for large n .

Before proceeding, it is worthwhile to cover one last point, an element of notation that proves very useful in practice. In many situations, we will find that a random variable, say S_n , is approximated by

$$S_n = \text{something with a nice Distribution} + \text{remainder},$$

where the “remainder” term is complicated or unenlightening. The error in our approximation is determined by the size of the remainder, but since the remainder is a random variable, we can only make a *probabilistic bound* on the size of the error.

Because we will usually be considering approximations that work best when n is large, what we really care about the error is how it grows (or shrinks) with n . It is often useful to not specify that error term too precisely but rather to indicate merely

its “stochastic order,” roughly speaking how big it can be with high probability.

There are two shorthand notations for stochastic order O_P (“big Oh P”) and o_P (“little Oh P”). These are similar to the “big Oh” and “little Oh” notations you have seen in the analysis of algorithms but with a probabilistic flavor. (The P stands for probability.) These terms stand in for the remainder random variables in our approximations. We do not specify these remainders too exactly because for our purposes there is no need to go to that much effort. Instead, specify the biggest the remainder can be *with high probability* and for large n .

In each case, we indicate which type (big or little) and indicate the order of approximation as a function of n . For example, we might write $S_n/\sqrt{n} = O_P(1)$ or $S_n = n\mu + o_P(n)$. The term in parentheses of the O_P or o_P is a function of n . The O_P and o_P act just like terms in any mathematical expression, but they let us leave these terms unspecified.

We define O_P and o_P precisely as follows. We say that a sequence of random variables X_n satisfies $X_n = O_P(1)$ as $n \rightarrow \infty$ if the X_n s are *all* bounded (by the same constant) with high probability. Specifically, for *any* $0 \leq \epsilon < 1$ there is some constant c_ϵ and a positive integer n_0 such that

$$\mathbb{E}(\{|X_n| \leq c_\epsilon\}) \geq 1 - \epsilon \quad \text{for all } n \geq n_0.$$

If $g(n)$ is a function of n , we say that $X_n = O_P(g(n))$ if $X_n/g(n) = O_P(1)$. We will see examples in what follows.

We say that a sequence of random variables X_n satisfies $X_n = o_P(1)$ as $n \rightarrow \infty$ if the X_n s put their mass closer and closer to zero as n grows large. This means that for large n , the X_n ’s are essentially zero. Specifically, for *any* constant c ,

$$\mathbb{E}(\{|X_n| > c\}) \rightarrow 0,$$

as $n \rightarrow \infty$. If $g(n)$ is a function of n , we say that $X_n = o_P(g(n))$ if $X_n/g(n) = o_P(1)$.

31.4 Review: Three Important Canonical Distributions

Key Take Aways

This section reviews three important families of Distributions that arise often in an approximation problems.

1. The Binomial Family **Binomial** $\langle n, p \rangle$

The Distribution of the number of IID events $V_1, \dots, V_n$ that occur:

$$N = V_1 + V_2 + \dots + V_n$$

$$D_N(\psi) = \sum_{k=0}^n \binom{n}{k} p^k (1-p)^{n-k} \psi(k).$$

The Bernoulli $\langle p \rangle$ (= Binomial $\langle 1, p \rangle$) family is the Kind/Distribution of a single event.

2. The Normal Family **Normal** $\langle \mu, \sigma^2 \rangle$

A ubiquitous and important family discussed in detail in Interlude C. The Normal Distribution is determined by its expectation and variance (matrix) which are its two parameters.

$$D_X(\psi) = \int \dots \int_x dx \frac{1}{\sqrt{\det 2\pi\sigma^2}} e^{-\frac{1}{2}(x-\mu)^T \sigma^{-2}(x-\mu)} \psi(x).$$

All Normal Distributions can be obtained by a simple affine transformation from the standard Normal Distribution which has $\mu = 0$ and σ^2 the identity matrix (i.e., 1 in the scalar case).

3. The Poisson Family **Poisson** $\langle \lambda \rangle$

A model for a count of *rare events*

$$D_M(\psi) = \sum_{m=0}^{\infty} \frac{\lambda^m}{m!} e^{-\lambda} \psi(m).$$

The idea of **canonical families** of Distributions, introduced in Chapter 1, is that some patterns recur so often that it is worth recognizing and naming them. In this section, we highlight three canonical families of particular relevance to our approximations. The first, the Normal (a.k.a. Gaussian) family, is central and arises in diverse applications and models across many different disciplines. It was discussed in Interlude C. We have seen the other two, the Binomial and Poisson families, in multiple examples, and they were briefly mentioned in Interlude C. We will see in the next Chapter that these both arise from a family of random processes, but for now, we will highlight a few of their properties.

If $V_1, V_2, \dots, V_n$ are independent events with all $\mathbb{E}(V_i) = p$ for some $0 \leq p \leq 1$,

then the random variable

$$N = V_1 + \cdots + V_n \quad (31.9)$$

counts how many of the events have occurred.

If we look at the independent join $V = V_1 \star \cdots \star V_n$, we can see by independence that any possible value $\langle i_1, \dots, i_n \rangle$ has weight

$$p^{i_1}(1-p)^{1-i_1}p^{i_2}(1-p)^{1-i_2}\cdots p^{i_n}(1-p)^{1-i_n} = p^{\sum_j i_j}(1-p)^{n-\sum_j i_j},$$

with one factor of p for every $i_j = 1$ and one factor of $1-p$ for every $i_j = 0$.

Notice that N is a transform of V by the statistic **Sum**, i.e., $N := \mathbf{Sum}(V)$. Each value of V with k 1's and $n-k$ 0's leads to $N = k$, and there is one such value for every subset of $[1 \dots n]$ corresponding to where we place the k 1's. Hence,

$$\mathbf{p}_N(k) = \mathbf{K}_N(k) = \binom{n}{k} p^k (1-p)^{n-k} \{k \in [0 \dots n]\}. \quad (31.10)$$

(Recall that $\mathbf{p}_N$ is the probability mass function of N , the discrete part of the kernel $\mathbf{K}_N$.) Hence,

$$\mathbf{D}_N(\psi) = \sum_{k=0}^n \binom{n}{k} p^k (1-p)^{n-k} \psi(k). \quad (31.11)$$

We can find $\mathbb{E}(N)$ from this formula, but it is easier to do so directly using additivity:

$$\begin{aligned} \mathbb{E}(N) &= \mathbb{E}\left(\sum_{j=1}^n V_j\right) \\ &= \sum_{j=1}^n \mathbb{E}(V_j) \\ &= \sum_{j=1}^n p \\ &= np. \end{aligned}$$

Similarly, using independence of the events V_j , we can find $\text{Var}(N)$:

$$\begin{aligned} \text{Var}(N) &= \text{Var}\left(\sum_{j=1}^n V_j\right) \\ &= \sum_{j=1}^n \text{Var}(V_j) \end{aligned}$$

Later we will say this counts the number of *marks* in n IID trials.

$$\begin{aligned}
&= \sum_{j=1}^n (\mathbb{E}(V_j^2) - p^2) \\
&= \sum_{j=1}^n (\mathbb{E}(V_j) - p^2) \\
&= \sum_{j=1}^n (p - p^2) = np(1 - p),
\end{aligned}$$

because $V_j^2 = V_j$ for any event.

Example 31.2.

A circuit contains eight components of a type that are prone to failing. Let $F_1, \dots, F_8$ be the events that the respective components fail during one year of operation. We assume that these are independent and that all fail with the same probability $p = 3/7$.

What is the probability that no components fail? What is the probability that more than one fails? What is the expected number that fail in a year?

Let $N := \sum_{j=1}^8 F_j$ be the total number of failures. We have

$$\begin{aligned}
\mathbb{E}(\{N = 0\}) &= D_N(\{\blacksquare = 0\}) \\
&= \binom{8}{0} p^0 (1-p)^8 \\
&= (1-p)^8 = \left(\frac{4}{7}\right)^8,
\end{aligned}$$

and

$$\begin{aligned}
\mathbb{E}(\{N > 1\}) &= D_N(\{\blacksquare \geq 1\}) \\
&= \sum_{k=1}^8 \binom{8}{k} p^k (1-p)^{8-k} \\
&= 1 - D_N(\{\blacksquare = 0\}) \\
&= 1 - (1-p)^8 = 1 - \left(\frac{4}{7}\right)^8.
\end{aligned}$$

And we have $\mathbb{E}(N) = \frac{24}{7}$.

The Poisson family is often used to model the counts of rare events that accumulate over some range of time or space. For instance, typographic errors are unlikely for

any particular letter or word, but they accumulate over the extent of a manuscript. A meteorite strike at any position on the planet is unlikely, but over enough time and area we will observe some. We can usefully model the counts of typographic errors in a stretch of text or of meteorite strikes over a range of time and space with a Poisson Distribution. We will formalize this idea in the next Chapter.

A random variable M has a Poisson $\langle\lambda\rangle$ Distribution if

$$D_M(\psi) = \sum_{m=0}^{\infty} \frac{\lambda^m}{m!} e^{-\lambda} \psi(m). \quad (31.12)$$

We can see right away that $D_M(\langle x \rangle \mapsto 1)$ is just $e^{-\lambda}$ times the series expression for e^{λ} and so equals 1.

We have $\mathbb{E}(M) = \lambda$ and $\text{Var}(M) = \lambda$. To see the former, plug $\langle x \rangle \mapsto x$ into D_M :

$$\begin{aligned} \mathbb{E}(M) &= \sum_{m=0}^{\infty} m \frac{\lambda^m}{m!} e^{-\lambda} \\ &= \sum_{m=1}^{\infty} m \frac{\lambda^m}{m!} e^{-\lambda} \\ &= \sum_{m=1}^{\infty} \frac{\lambda^{m-1} \lambda}{(m-1)!} e^{-\lambda} \\ &= \lambda \sum_{m=0}^{\infty} \frac{\lambda^m}{m!} e^{-\lambda} = \lambda, \end{aligned}$$

where we simply shifted the indices in the sum in the fourth equality (a “change of variables”). A similar approach gives $\mathbb{E}(M^2)$ and thus $\text{Var}(M)$.

The Constant Approximation

32

Chapter

Key Take Aways

The Constant Approximation. For IID random variables $X_1, X_2, \dots$ with well defined expectation $\mathbb{E}(X_i) = \mu$ and finite variance $\text{Var}(X_i) = \sigma^2 < \infty$, the partial sums $S_n = \sum_{i=1}^n X_i$ satisfy $S_n \approx n\mu$ and $S_n/n \approx \mu$.

More precisely,

$$S_n = n\mu + O_P(\sqrt{n}) \quad (32.1)$$

$$\frac{S_n}{n} = \mu + O_P\left(\frac{1}{\sqrt{n}}\right). \quad (32.2)$$

If the expectation $\mathbb{E}(X_i)$ exists but $\text{Var}(X_i) = \infty$, then we can only make a weaker approximation

$$S_n = n\mu + o_P(n) \quad (32.3)$$

$$\frac{S_n}{n} = \mu + o_P(1). \quad (32.4)$$

We can view any constant as a random variable that *always* has the same value. The Distribution of a constant puts all its mass at a single point. (And a constant random variable is independent of any other random variable because no knowledge changes our perfect prediction of the constant.)

The **Constant Approximation** tells us that the Distributions of S_n/n and S_n become increasingly concentrated around a constant value as n grows. Thus, their Distributions are well-approximated by a constant. The average S_n/n concentrates around the same constant μ for each n ; the sum S_n concentrates around a constant $n\mu$ that changes with n .

We can assess the error in this approximation directly using Chebychev's Inequality:

$$\mathbb{E}(\{|S_n - n\mu| \geq c\sqrt{n}\}) = \mathbb{E}\left(\left\{\left|\frac{S_n}{n} - \mu\right| \geq \frac{c}{\sqrt{n}}\right\}\right) \leq \frac{\sigma^2}{c^2}.$$

Hence, if for $0 < \epsilon < 1$, we take $c_\epsilon \geq \sigma/\sqrt{\epsilon}$, we have

$$\mathbb{E}(\{|S_n - n\mu| \leq c_\epsilon \sqrt{n}\}) = \mathbb{E}\left(\left\{\left|\frac{S_n}{n} - \mu\right| \leq \frac{c_\epsilon}{\sqrt{n}}\right\}\right) \geq 1 - \epsilon,$$

and consequently

$$\begin{aligned} S_n &= n\mu + O_P(\sqrt{n}) \\ \frac{S_n}{n} &= \mu + O_P\left(\frac{1}{\sqrt{n}}\right). \end{aligned}$$

This gives us the Constant Approximation.

The Constant Approximation. For IID random variables $X_1, X_2, \dots$ with well defined expectation $\mathbb{E}(X_i) = \mu$ and finite variance $\text{Var}(X_i) = \sigma^2 < \infty$, the partial sums $S_n = \sum_{i=1}^n X_i$ satisfy $S_n \approx n\mu$ and $S_n/n \approx \mu$.

Specifically:

$$S_n = n\mu + O_P(\sqrt{n}) \quad (32.5)$$

$$\frac{S_n}{n} = \mu + O_P\left(\frac{1}{\sqrt{n}}\right). \quad (32.6)$$

If the expectation $\mathbb{E}(X_i)$ exists but $\text{Var}(X_i) = \infty$, then we can only make a weaker approximation

$$S_n = n\mu + o_P(n) \quad (32.7)$$

$$\frac{S_n}{n} = \mu + o_P(1). \quad (32.8)$$

Example 32.1 The House Always Wins A distinguishing feature of casino parlor games like craps or roulette is that they are designed to give the house an edge. All the bets that a player can place have expected winnings close to but strictly less than 0. You can view this as the “house’s take” for playing the game, but the implications for the player are grim.

Suppose you place a series of identical bets on repeated trials of a parlor game with expected payoff $\mu < 0$ per trial. For example, betting 10 dollars on red in roulette gives $\mu = -10/19$. Let $B_1, B_2, \dots$ be the payoff (positive to you, negative to the house) of each bet. Assume for simplicity that you bet the same stakes each

time and that the choice of whether or not to bet is not based on your betting history, so the B_i s form an independent and identically distributed sequence. Your total payoff after n bets is $W_n = \sum_{i=1}^n B_i$. The Constant Approximation gives us that $W_n \approx n\mu < 0$, or more specifically that $W_n = n\mu + O_P(\sqrt{n})$. As n grows, n becomes much larger than $\sqrt{n}$, so with probability going to 1, the O_P remainder term is dominated by $n\mu$. (This follows from the Chebychev's argument earlier; when the stakes of the bets are bounded, like in Roulette, we can use Hoeffding's inequality to get an even stronger bound.) This means that **if you play long enough, you will eventually lose**.

Looking at the situation from the house's point of view, the Constant Approximation tells us why casinos are so profitable. Imagine a casino with one game, such as Roulette, for which bets of only a single stake (say 10 dollars) are allowed. Let $R_1, R_2, R_3, \dots$ be the payoff (this time positive to the house, negative to the players) of every roulette spin over the casino's history. Assuming again for simplicity that all bets are independent, let $D_n = \sum_{i=1}^n R_i$ be the net payoff to the casino. As n grows large, the Constant Approximation gives $D_n \approx n\mu > 0$. More specifically, $D_n = n\mu + O_P(\sqrt{n})$, and again with large n , the remainder term is almost certain to be smaller in absolute value than the $n\mu$ term. So, if the casino can attract enough bettors to its game, it will eventually make money that scales with the number of bets.

It is interesting and worthwhile to adjust several of the assumptions underlying this analysis, but the basic story remains the same. Playing a "subfair" game repeatedly is a losing proposition for the player. You can't beat the house in the long term.

Example 32.2 Monte Carlo Simulation

Monte Carlo simulation is a technique where we use random samples from a Distribution to calculate the solution to deterministic problems. The Constant Approximation (aka Law of Large Numbers) plays a fundamental role in many Monte Carlo calculations.

Here we consider two applications of Monte Carlo simulations.

First, suppose we want to find the integral of a function over an interval $[a, b]$: $I = \int_{t=a}^b g(t) dt$. We generate IID random variables $X_1, X_2, X_3, \dots$ with a

n	$\hat{\pi}$	$ \hat{\pi} - \pi $
10^2	3.1600000	0.0184073
10^3	3.1280000	0.0135926
10^4	3.1464000	0.0048073
10^5	3.1345600	0.0070326
10^6	3.1390600	0.0025326
10^7	3.1413352	0.0002574
10^8	3.1414527	0.0001398
10^9	3.1415471	0.0000455
π	3.1415926	

TABLE 32.1. Monte Carlo estimates of π and approximation error in Example 32.2.

Uniform $\langle a, b \rangle$ Distribution. Then,

$$\mathbb{E}(X_i) = \frac{1}{b-a} \int_{t=a}^b dt \, g(t) = \frac{1}{b-a} I.$$

The constant approximation tells us that

$$\frac{b-a}{n} S_n \approx I.$$

Second, we will use Monte Carlo simulation to estimate π . Let $(X_1, Y_1), \dots, (X_n, Y_n)$ be IID random variables that are uniform in the square $[-2, 2] \times [-2, 2]$. Define $C_n = \{X_n^2 + Y_n^2 \leq 1\}$, the event that the n th point lands in the unit circle.

Define $S_n = 4 \sum_{i=1}^n C_i$. Because $\mathbb{E}(C_i) = \pi/4$, the Constant Approximation tells us that

$$\frac{S_n}{n} \approx \pi.$$

If we run this simulation for large n , we see that the results match our theory quite well, as shown in Table 32.1.

As mentioned earlier, each approximation result here corresponds to a *limit theorem* that gives convergence (in some sense) of the Distribution of S_n or S_n/n as $n \rightarrow \infty$. The limit theorem that corresponds to the Constant Approximation is called the **Law of Large Numbers**. It states that $S_n/n \rightarrow \mu$ as $n \rightarrow \infty$, meaning that in the limit, the distribution of S_n/n becomes totally concentrated at the single point μ . There are various versions (called Weak and Strong) of the Law of Large Numbers depending on the type of convergence that is claimed, but for our purposes

we can elide these distinctions. Here, we will take the Constant Approximation and the Law of Large Numbers as synonymous.

It is also worth noting that the Constant Approximation applies more broadly than in the situation presented here (i.e., sums of IID random variables). For instance, one can show that if the X_i 's are independent but *not* necessarily identically distributed, with expectations $\mathbb{E}(X_i) = \mu_i$ and variances $\sigma_i^2 = \text{Var}(X_i) \leq s^2 < \infty$ for some positive s , then, again the Distributions of S_n and S_n/n concentrate around a constant:

$$S_n = \sum_{i=1}^n \mu_i + O_P(\sqrt{n})$$

$$\frac{S_n}{n} = \frac{1}{n} \sum_{i=1}^n \mu_i + O_P\left(\frac{1}{\sqrt{n}}\right).$$

The Normal Approximation

33

Chapter

Key Take Aways

The Normal Approximation There exists a $\text{Normal}\langle 0, 1 \rangle$ distributed random variable Z such that

$$S_n \approx n\mu + \sqrt{n}\sigma Z, \quad (33.1)$$

or equivalently,

$$\frac{S_n - n\mu}{\sigma\sqrt{n}} \approx Z. \quad (33.2)$$

This result, expressed as a limit theorem, is called the **Central Limit Theorem**.

The Normal Approximation is often accurate for surprisingly small n . It is more accurate when the Distribution of X_i is symmetric and can be corrected in various ways to improve the approximation when the Distribution of X_i is discrete or highly skewed.

The Normal Approximation generalizes the Constant Approximation. Not only do the Distributions of S_n/n and S_n concentrate around a constant value, they do so in such a way that, to first order, the deviation in their Distribution around that constant looks like a Normal Distribution.

The **Normal Approximation** can be stated simply: **a sum of of IID random variables with finite variance can be approximated by the Normal Distribution with same expectation and variance as the sum**. In mathematical terms:

S_n has approximately a $\text{Normal}\langle \mathbb{E}(S_n), \text{Var}(S_n) \rangle$ Distribution.

More explicitly.

S_n has approximately a $\text{Normal}\langle n\mu, n\sigma^2 \rangle$ Distribution, and

$\frac{S_n}{n}$ has approximately a $\text{Normal}\langle \mu, \frac{\sigma^2}{n} \rangle$ Distribution.

Because Distributions in the Normal family are determined exactly by two parameters – the expectation and variance – we find the Normal Approximation by matching the expectation and variance of the Distribution being approximated. For sums of IID random variables, once we know the expectation and variance of the terms in the sum, we can write down the approximation directly.

Let's express the above in terms of random variables. The Normal Approximation claims that there is a random variable Z with *standard Normal Distribution* such that

$$S_n \approx n\mu + \sigma\sqrt{n}Z$$

$$\frac{S_n}{n} \approx \mu + \frac{\sigma}{\sqrt{n}}Z.$$

Notice that this approximation works regardless of whether the variables being summed are discrete or continuous. Expectations computed with the Distribution on one side of the equation are well-approximated by expectations computed with the Distribution on the other side.

The CDF of S_n , for instance, is thus given approximately by

$$F_{S_n}(t) \approx \Phi\left(\frac{t - n\mu}{\sigma\sqrt{n}}\right).$$

Why? The right-hand side is just the CDF of a $\text{Normal}\langle n\mu, n\sigma^2 \rangle$ Distribution.

Example 33.1 Sums of Uniforms Suppose the X_i 's are IID $\text{Uniform}\langle -1, 1 \rangle$ random variables. Then $\mu = \mathbb{E}(X_1) = 0$ and

$$\sigma^2 = \text{Var}(X_1) = \mathbb{E}(X_1^2) = \int_{t=-1}^1 dt \frac{1}{2}t^2 = \frac{1}{3}.$$

The Normal Approximation tells us that S_n has approximately a $\text{Normal}\langle 0, n/3 \rangle$ Distribution.

This is easiest to visualize by looking at $T_n = S_n\sqrt{3/n}$ which has approximately a standard Normal Distribution. See Figures 33.1 and 33.2 below to look at the Distributions of T_n and their deviations from the Normal. Notice that the approximation is very good even for n as small as 6.

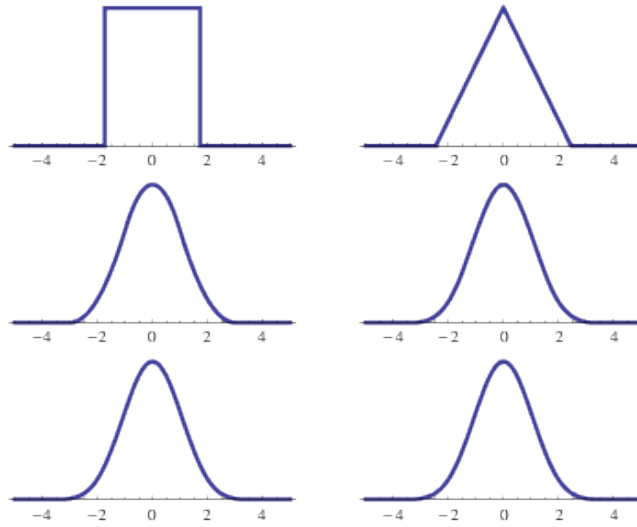


FIGURE 33.1. Kernel of T_n for $n \in [1..6]$ (left to right, top to bottom)

Example 33.2 Normal Approximation to the Binomial The Binomial Distributions model the number of marks in a fixed number of Bernoulli Trials with the same mark probability. Random variables with a Binomial Distribution are thus sums of IID random variables with bounded variance; the Normal Approximation thus applies.

Let M have a Binomial $\langle n, p \rangle$ Distribution for positive integer n and $0 \leq p \leq 1$. We can write $M = \sum_{k=1}^n M_k$, where M_k is an indicator indicating a mark on trial k , with $\mathbb{E}(M_k) = p$ and $\text{Var}(M_k) = p(1 - p)$.

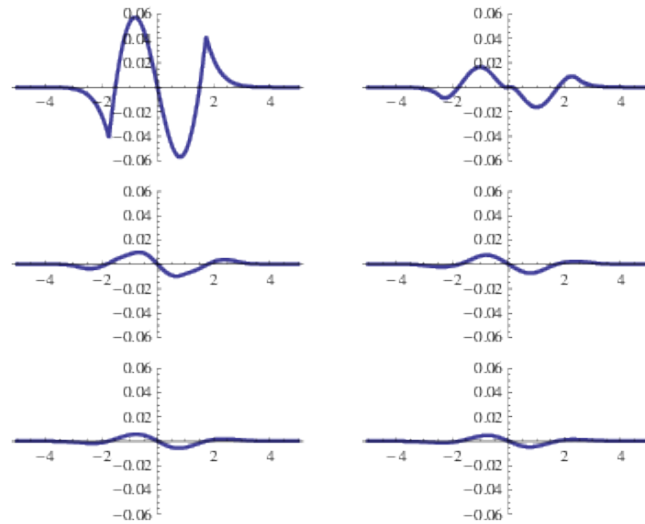
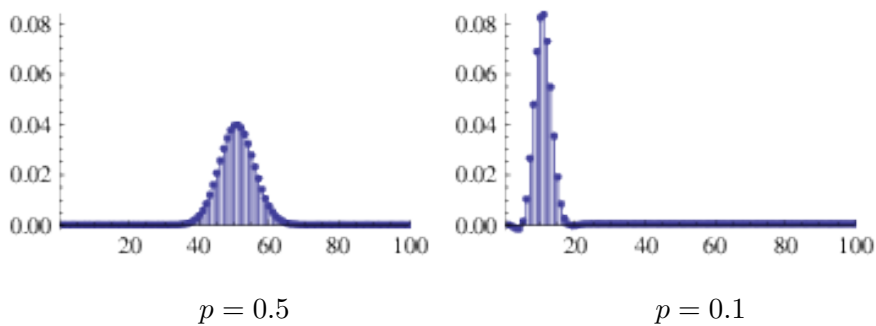
Hence, we have immediately that M is approximately Normal $\langle np, np(1 - p) \rangle$. Put another way:

$$F_M(t) \approx \Phi \left(\frac{t - np}{\sqrt{np(1 - p)}} \right).$$

This is the Normal approximation to the Binomial Distributions.

This approximation will be better when p is closer to $1/2$ because the Binomial Distribution in that case is more symmetric. In such cases, the approximation is good for fairly small n , roughly $n \geq 30$. But for p in reasonable ranges, say $0.1 \leq p \leq 0.9$, the approximation still does a reasonable job. Figure 33.3 shows the error in the basic approximation for $n = 100$ and selected values of p .

In the case of the Normal Approximation to the Binomial, we are approximating a discrete Distribution by a continuous Distribution. The quality of the approximation can thus be affected by how we smear the mass around any discrete value. For

FIGURE 33.2. $F_{T_n} - \Phi$ for $n \in [1..6]$ (left to right, top to bottom)FIGURE 33.3. Error in Normal Approximation to Binomial $\langle 100, p \rangle$ kernel.

example, the Normal Approximation above puts probability mass $K(k)$ over the interval $[k, k+1)$, where K is the corresponding Binomial kernel. So the probability closest to k is $K(k-1)/2 + K(k)/2$. Instead, we'd like to smear the probability $K(k)$ over the interval $[k - \frac{1}{2}, k + \frac{1}{2})$. This is called the **continuity correction** and applies when approximating discrete Distributions with the Normal Approximation.

Example 33.3 Normal Approximation to the Binomial with Continuity Correction Again, take M to have a Binomial $\langle n, p \rangle$ Distribution. Applying the continuity correction means shifting where we put the approximate Distribution

by $1/2$. That is, we take

$$F_M(t) \approx \Phi \left(\frac{t + \frac{1}{2} - np}{\sqrt{np(1-p)}} \right).$$

This implies for instance that

$$\mathbb{E}(\{j \leq M \leq k\}) = F_M(k) - F_M(j-1) \approx \Phi \left(\frac{k + \frac{1}{2} - np}{\sqrt{np(1-p)}} \right) - \Phi \left(\frac{j - \frac{1}{2} - np}{\sqrt{np(1-p)}} \right).$$

(Notice that $\mathbb{E}(\{j < M \leq k\})$ would have a $j + \frac{1}{2}$ not $j - \frac{1}{2}$ in the second term.)

Figure 33.4 shows the error in the approximation with continuity correction for selected values of p . Notice that the correction helps both for $p = 0.5$ and for $p = 0.1$, but the change is more marked for the former. Why? Because when $p = 0.5$ the approximation is already very good, so the continuity error is the dominant term in the error. For $p = 0.1$, on the other hand, the main part of the error in approximation results both from the asymmetry of the Distribution and the lack of continuity, so the correction has a smaller effect.

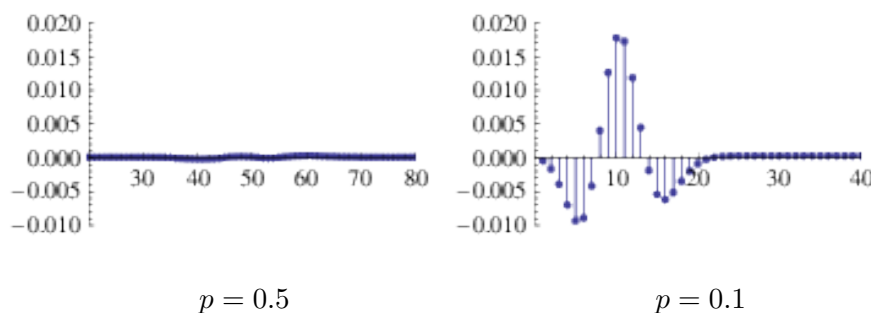


FIGURE 33.4. Error in Normal Approximation to Binomial $\langle 100, p \rangle$ kernel.

Example 33.4. First, we flip a coin 100 times, where the coin has probability $1/2$ of coming up heads. What is the chance that we get more than 40 and fewer than 60 flips? Second, we flip a coin 100 times, where the coin has probability $1/10$ of coming up heads. What is the chance that we get more than 4 and fewer than 16 flips?

For the first question, we want to find $\mathbb{E}(\{40 < S_n \leq 60\})$. We have

$$\begin{aligned}\mathbb{E}(\{40 < S_n \leq 60\}) &= F_{S_n}(60) - F_{S_n}(40) \\ &\approx \Phi\left(\frac{60 - 100 \cdot \frac{1}{2}}{\sqrt{100 \cdot \frac{1}{2} \cdot \frac{1}{2}}}\right) - \Phi\left(\frac{40 - 100 \cdot \frac{1}{2}}{\sqrt{100 \cdot \frac{1}{2} \cdot \frac{1}{2}}}\right) \\ &= \Phi\left(\frac{10}{\sqrt{25}}\right) - \Phi\left(\frac{-10}{\sqrt{25}}\right) \\ &= \Phi(2) - \Phi(-2) \approx 0.9545,\end{aligned}$$

which differs from the true value, about 0.9540, by 0.0005, roughly 0.06%. Quite good.

For the second question, we want to find $\mathbb{E}(\{4 < S_n \leq 16\})$. We have

$$\begin{aligned}\mathbb{E}(\{4 < S_n \leq 16\}) &\approx \Phi\left(\frac{16 - 100 \cdot 0.1}{\sqrt{100 \cdot 0.1 \cdot 0.9}}\right) - \Phi\left(\frac{4 - 100 \cdot 0.1}{\sqrt{100 \cdot 0.1 \cdot 0.9}}\right) \\ &= \Phi(2) - \Phi(-2) \approx 0.9545,\end{aligned}$$

which differs from the true value, about 0.9557, by 0.0012, roughly 0.12%. Also, good, though with about twice the error.

The Normal Approximation uses a symmetric Distribution to do the approximation, so it stands to reason that if the Distribution being approximated is not symmetric, the approximation will suffer. We can attempt to correct for that by adjusting the approximation for a measure of asymmetry in the target Distribution. In the case of sums of IID random variables, we can do that only using knowledge of the asymmetry in an individual term.

One measure of asymmetry in a Distribution is called **skewness**, given by

$$\text{Skew}(X) = \mathbb{E}\left[\left(\frac{X - \mathbb{E}(X)}{\text{SD}(X)}\right)^3\right].$$

For a symmetric Distribution, the skewness is zero. When the Distribution puts more mass far above the expectation than below it, the skewness is positive. When the Distribution puts more mass far below the expectation than above it, the skewness is negative. See Figure 33.5.

For example, the skewness of a Binomial(n, p) Distribution is $(1 - 2p)/\sqrt{np(1 - p)}$, which is positive when $p < \frac{1}{2}$, zero when $p = \frac{1}{2}$, and negative when $p > \frac{1}{2}$.

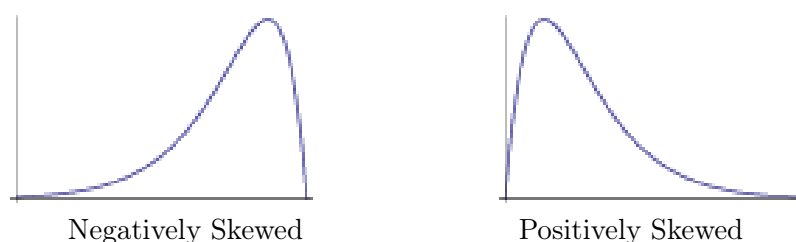


FIGURE 33.5. Illustrating Skewness, positive and negative.

We can adjust the Normal approximation for skewness by adding some asymmetry to the approximation commensurate with the skewness of the individual terms. The result is the **Skewness-Corrected Normal Approximation**. Recall that Φ is the CDF of the standard Normal Distribution.

The Skewness-Corrected Normal Approximation

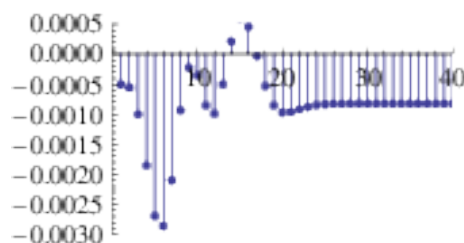
$$F_{S_n}(t) \approx \Phi\left(\frac{t - n\mu}{\sigma\sqrt{n}}\right) - \frac{\text{Skew}(X_1)}{6\sqrt{n}}\Phi'''\left(\frac{t - n\mu}{\sigma\sqrt{n}}\right).$$

When the X_i 's are discrete, we can apply the continuity correction as well:

$$F_{S_n}(t) \approx \Phi\left(\frac{t + \frac{1}{2} - n\mu}{\sigma\sqrt{n}}\right) - \frac{\text{Skew}(X_1)}{6\sqrt{n}}\Phi'''\left(\frac{t + \frac{1}{2} - n\mu}{\sigma\sqrt{n}}\right).$$

The Φ''' term, the third derivative of Φ , adds asymmetry because symmetry in a CDF is seen by $F(-t) = 1 - F(t)$.

If we apply the Skewness-Corrected Normal Approximation to the Binomial $\langle 100, 0.01 \rangle$ we see that the correction leads to nontrivial improvement. This is shown in Figure 33.6.

FIGURE 33.6. Error in Skew-Corrected Normal Approximation to Binomial $\langle 100, 0.1 \rangle$ kernel.

In practice, the continuity correction is quite often used, though the improvement it offers is not always large. The skewness correction is less often used, though it can

be handy at times. The Normal Approximation can be further adjusted for other features of the X_i 's Distribution to make the approximation better, with decreasing returns.

Finally, as mentioned earlier, each approximation result here corresponds to a *limit theorem* that gives convergence (in some sense) of the Distribution of S_n or S_n/n as $n \rightarrow \infty$. The limit theorem that corresponds to the Normal Approximation is called the **Central Limit Theorem** (or CLT for short). The CLT is one of the most important theorems in the mathematical sciences; it tells us that the sum or average of independent random variables (with a few conditions) properly normalized converges to a standard Normal Distribution. For instance, the CLT tells us that the Distribution of $(\frac{S_n}{n} - \mu)/(\sigma/\sqrt{n})$ converges (in a particular sense) to a Normal $\langle 0, 1 \rangle$ Distribution as $n \rightarrow \infty$. The CLT can be strengthened substantially from what we have described here, but for our purposes, the IID version is what we most often need. Here, we will take the Normal Approximation and the Central Limit Theorem as synonymous.

The Poisson Approximation

34

Chapter

Key Take Aways

The Poisson Approximation. If S_n has an Binomial $\langle n, \mu \rangle$ Distribution and K_n has a Poisson $\langle n\mu \rangle$ Distribution, then $S_n \approx K_n$. Hence, the Distribution of S_n is approximated by a Poisson $\langle n\mu \rangle$ Distribution.

More precisely, for any subset $\mathcal{A}$ of the non-negative integers,

$$\left| \mathbb{E}(\{S_n \in \mathcal{A}\}) - \mathbb{E}(\{K_n \in \mathcal{A}\}) \right| \leq \mu (1 - e^{-n\mu}) \leq \mu \min(1, n\mu). \quad (34.1)$$

Expressed as a limit theorem, this is sometimes called the **Law of Rare Events**.

The Poisson Approximation is accurate in situations (highly skewed Distribution of an indicator) where the Normal Approximation is less accurate.

The Poisson Distributions model counts of rare events. This claim manifests itself in the **Poisson Approximation**. This approximation applies in only limited circumstances, but when it does, it is both simpler and more accurate than the Normal Approximation.

Assume that the X_i 's are indicators with $\mathbb{E}(X_i) = \mu$, where $0 < \mu < 1$. This implies that S_n has a Binomial $\langle n, \mu \rangle$ Distribution. When μ is small, the Distribution of S_n is approximately Poisson $\langle n\mu \rangle$. This approximation ensures that the target and approximate probabilities are simultaneously close for all events.

The Poisson Approximation to the Binomial Distribution. If S_n has an Binomial $\langle n, \mu \rangle$ Distribution and K_n has a Poisson $\langle n\mu \rangle$ Distribution, then for any subset $\mathcal{A}$ of the non-negative integers,

$$\left| \mathbb{E}(\{S_n \in \mathcal{A}\}) - \mathbb{E}(\{K_n \in \mathcal{A}\}) \right| \leq \mu (1 - e^{-n\mu}) \leq \mu \min(1, n\mu). \quad (34.2)$$

We usually focus on the case where μ is no greater than λ/n for some $\lambda > 0$. In that case, we can think of the underlying Binomial Distribution as counting marks

in many, many Bernoulli trials where the mark probability is very low. The marks are the rare events, and the count of rare events is approximately Poisson, with the approximation better the more trials there are and the rarer the marks are.

Like the other approximations, the Poisson Approximation holds more generally. For instance, if the X_i 's are independent but *not necessarily* identically distributed indicators with $\mathbb{E}(X_i) = \mu_i$ and if $\lambda = \sum_i \mu_i$, then S_n has approximately a $\text{Poisson}(\lambda)$ Distribution, and

$$\left| \mathbb{E}(\{S_n \in \mathcal{A}\}) - \mathbb{E}(\{K_n \in \mathcal{A}\}) \right| \leq \min(1, 1/\lambda) \sum_i \mu_i^2,$$

for $\text{Poisson}(\lambda)$ random variable K_n . This reduces to the above case when the X_i 's are IID.

Also, like the other approximations, this one has an associated limit theorem, sometimes called the **Law of Rare Events**.

Example 34.1. Every few years a story appears in a newspaper describing in awed tones some mind-boggling coincidence that has occurred. A staple of such stories is the lottery coincidence: a woman who won the New York State jackpot twice within four months, a couple who won two California lotteries on the same day, two states with the same lottery number on the same day, and an eerie match between the lottery numbers on a day and the day's events, such as 9-1-1 in New York on 11 Sep 2002 or 5-8-7 in New York on the day Flight 587 crashed off Long Island. The newspaper stories – and sometimes even the Lottery commissions – quote miniscule probabilities for these events, suggesting that we have witnessed a near miracle. But are these coincidences so unlikely? The Poisson Approximation can help us answer that question.

For concreteness, let's consider the case of Angelo and Maria Gallina, a California couple that won the Fantasy Five \$126,000 jackpot and a SuperLotto Plus \$17 million jackpot on the same day. [From *San Francisco Chronicle* story “Double lottery winners beat odds of 1 in 24,000,000,000,000” by Steve Rubenstein.]

12 December 2002 – Belmont, CA.

It had to happen sooner or later for Angelo and Maria Gallina, who figure they have spent \$124,000 over the years on lottery tickets.

What happened was that they won the jackpot – not once, but twice,

on the same day. An hour after winning \$126,000 in the Fantasy Five game, they won \$17 million in SuperLotto Plus.

That's never been done before, lottery officials said Wednesday, maybe because the odds of its happening are 1 in 24 trillion – which is a 24 followed by 12 zeros.

And maybe because there have never been lottery players like the Gallinas.

Angelo Gallina, a 78-year old retired railroad machinist from Belmont, cheerfully admitted that he has bought \$20 worth of lottery tickets since the lottery started in 1985.

The newspaper's number "1 in 24 trillion" derives from a simple random experiment: *you* walk into a Gas 'n Sip and purchase one ticket for each of the two games for *next Saturday's drawing*. The quoted probabilities of winning the Fantasy Five and SuperLotto Plus are $1/575,757$ and $1/41,416,353$ respectively. Because the two drawings are independent, the probability that you win both is the product of those probabilities, which is indeed about one in 24 trillion. That's a small number, hard to fully comprehend.

But the Gas 'n Sip experiment is not the same as the one run by the Gallinas. The Gallinas decided back in 1985 that they would play these two games every time they were offered and that each time they would purchase a fixed number of tickets. In the Gas 'n Sip experiment, there is a single, indicator random variable W that represents whether you win both drawings next Saturday. Here, $\mathbb{E}W \approx 4.19 \times 10^{-14}$. In the Gallinas' experiment, however, there is a sequence of independent and identically distributed indicator random variables $W_1, W_2, W_3, \dots$, where W_i represents a simultaneous win on both games on the i th day they play. Assuming they spend \$10 on each game on the days when both games are offered, $\mathbb{E}W_i \approx 4.19 \times 10^{-12}$, larger by a factor of 100 than in the previous experiment.

Let $S_n = W_1 + \dots + W_n$ be the number of days on which the Gallinas win both games. Because $\mu = \mathbb{E}(W_1)$ is small, the Poisson Approximation tells us that $\mathbb{E}(\{S_n \geq 1\}) \approx 1 - e^{-n\mu} \approx n \times 4.19 \times 10^{-12}$, which is small for any realistic n . The event $\{S_n \geq 1\}$ contains the outcomes in which the Gallinas have won both games simultaneously by day n . At the time of the newspaper story, the Gallinas had played both games together $n = 1768$ times, giving

$\mathbb{E}(\{S_{1768} \geq 1\}) \approx 7.41 \times 10^{-9}$. This comes from taking $\mathcal{A} = \{1, 2, 3, \dots\}$ in equation (34.2). The formal version of the approximation tells us that

$$|\mathbb{E}(\{S_{1768} \geq 1\}) - (1 - e^{-n\mu})| \leq \mu \min(1, n\mu) < 3.1 \times 10^{-20}, \quad (34.3)$$

which is a negligible difference, so the approximation is a good one. Although $\mathbb{E}(\{S_{1768} \geq 1\})$ is still small – about 1 in 135 million or 1/3 the probability of winning the SuperLotto Plus on a single play – it is much, much larger than the dramatic 1 in 24 trillion figure quoted in the story.

But this seemingly simple analysis is still deceptive because the Gallinas' experiment, important as it is to them, is not the random experiment being reported on by the newspaper. To address the question of whether the Gallinas' double win is a mind-boggling coincidence, we have to determine the probability that an apparent lottery coincidence is enough to prompt a newspaper story. A story would still have been written if Mr.~and Mrs.~Gallina had won both lotteries the same week, the same month, or even the same year. A story would still have been written if one of their neighbors had won twice instead. A story would probably have been written if any two neighbors (or siblings or ex-spouses or ...) won the jackpot on separate plays in the same week. *Any* combination of wins by *any* related players would prompt a story if it appeared sufficiently surprising, so a newsworthy coincidence is much more likely than it would seem.

This mismatch between the surprise invoked by an apparent coincidence and the actual probability has both psychological and mathematical components. For example, if you flipped a coin twelve times and obtained HTHHTHTHTHT, your friend might express surprise at the pattern of the results because it seems so unlikely to get such a pattern by chance. People tend to see such patterned outcomes as less likely than other possibilities (e.g., THHTTTHTHTT). That is the psychological component. The mathematical component is that there is a large number of other patterns that might have seemed equally surprising had they occurred instead (e.g., HHHHHHTTTTTT, HHTTHHTTHHT, ...). When all of these “surprising” patterns are accounted for, the probability that we see one of them is much larger. For instance, the probability of seeing HTHHTHTHTHT is 2^{-12} , but the probability of seeing any of M surprising patterns is $M \cdot 2^{-12}$, which is much larger if M is large.

As an illustration, we will examine a small subset of possible lottery coincidences: double wins on the Fantasy Five and SuperLotto Plus that occur for

the same player in the same week. To keep our analysis simple and concrete, we will assume the following: that (1)~1\who have played both games twice every week for the 17 years the games have been available; that (2)~each such player purchases 2 tickets for each game every week(as opposed to 5 each for Angelo and Maria); and that (3)~we can ignore the small effect caused by multiple players choosing the same numbers to play. While untested, these assumptions seem plausible, and it is straightforward to vary the assumptions to see how sensitive our conclusions are to the particular assumptions we make.

For player i during week j , define the random variable D_{ij} to be the indicator that the player won both games at least once during the week. If $p_F = 1/575757$ and $p_L = 1/41416351$ denote the respective probabilities of winning the Fantasy Five and SuperLotto Plus on a single ticket,

$$\mathbb{E}D_{ij} = (1 - (1 - 2p_F)^2)(1 - (1 - 2p_L)^2) \approx 4p_F \cdot 4p_L \approx 6.71 \times 10^{-13} \quad (34.4)$$

because each player plays both games twice during the week. After 17 years, the players have played for $52 \cdot 17 = 884$ weeks. Define

$$S = \sum_{j=1}^{884} \sum_{i=1}^{350,000} D_{ij}. \quad (34.5)$$

This is a sum of 309,400,000 indicators. The Poisson Approximation tells us that

$$\mathbb{E}(\{S \geq 1\}) \approx 1 - e^{309,400,000 \times 6.71 \times 10^{-13}} \approx 0.00021, \quad (34.6)$$

about 1 in 5000.p Compared to 1 in 24 trillion this is enormous. Because there are more players and more ways for an apparent coincidence to occur, the probability of a “surprising outcome” is larger. As the set of apparently surprising outcomes grows, the probability increases quickly. For example, if we call it coincidental for someone to win both games during a calendar year, the probability of a coincidence grows to about 0.01. And we can further include double wins by related players and allow for the double wins among any of the state lotteries running over the same time span. Once we do so, the probability becomes larger still, and what seemed a coincidence now seems almost mundane.

The newspaper reporter derived his probability from the Gas 'n Sip experiment, and though the calculation was correct, that choice of experiment was not relevant to the question of how coincidental the double win is. Notice that I

was careful in that experiment to specify the player and the time of the draw. Angelo and Maria Gallina made their decision (in principle) using the Gallinas experiment. (The calculation of whether to invest their \$124,000 in the lottery ticket purchases would need to account for partial winnings along the way and the discounted value of their investment.) The Gallinas might be surprised that *they* won both games on the same day, the probability that that would happen to *them* is quite small. But as the Multi-Coincidence Experiment shows, *we* should be much less surprised that *someone* won two lotteries on the same day.

Statisticians Persi Diaconis and Fred Mosteller have proposed what they call the Law of Truly Large Numbers: “With a large enough sample, any outrageous thing is likely to happen.” Events which seem astoundingly unlikely but which still happen frequently do so because there are so many “trials” on which the unlikely event can occur.

Concentration Inequalities

35

Chapter

Contents

35.1 Markov and Chebychev Inequalities	1132
35.2 Chernoff Bounds	1134
35.3 Hoeffding's Inequality	1142
35.4 More Examples	1145

Key Take Aways

We are often interested in “tail events”, unlikely or extreme outcomes whose occurrence will have significant implications.

For a random variable whose Distribution is **concentrated** around a particular value, like its expectation, deviations of the random variable from that value are tail events. In many practical cases, we can *bound* the probability of such events, and often quite tightly, even in cases where the exact probabilities are difficult to obtain.

Such **concentration inequalities**, also known as tail bounds, are approximations that can typically be made more easily than full approximations of a Distribution. Here we consider several commonly used tail bounds that are applicable in a range of circumstances.

- **Markov's Inequality.**

If X is a random variable and ψ is a compatible statistic such that $\text{fact}(\psi(X) \geq 0)$, then for any constant $c > 0$:

$$\mathbb{E}(\{\psi(X) \geq c\}) \leq \frac{1}{c} \mathbb{E}(\psi(X)). \quad (35.1)$$

- **Chebychev's Inequality.**

If X is a random variable with finite variance, then

$$\mathbb{E}(\{|X - \mathbb{E}(X)| \geq c\}) \leq \frac{1}{c^2} \text{Var}(X). \quad (35.2)$$

- **Chernoff Bounds.**

If X is a scalar random variable:

$$\mathbb{E}(\{X \geq c\}) \leq \min_{t>0} e^{-tc} \mathbb{E}(e^{tX}). \quad (35.3)$$

If $S_n = \sum_{i=1}^n X_i$ is a sum of independent random variables:

$$\mathbb{E}(\{S_n \geq c\}) \leq \min_{t>0} e^{-tc} \prod_{i=1}^n \mathbb{E}(e^{tX_i}). \quad (35.4)$$

If $S_n = \sum_{i=1}^n X_i$ is a sum of independent *events*:

$$\mathbb{E}(\{S_n \geq a \mathbb{E}(S_n)\}) \leq e^{-(a \log a - a + 1) \mathbb{E}(S_n)}. \quad (35.5)$$

- **Hoeffding's Inequality (one sided).**

If $S_n = \sum_{i=1}^n X_i$ is a sum of independent random variables with $\mathbb{E}(\{a_k \leq X_k \leq b_k\}) = 1$ for some $-\infty < a_k \leq b_k < \infty$, then for $t > 0$, the following are equivalent:

$$\mathbb{E}(\{S_n - \mathbb{E}(S_n) \geq t\sqrt{n}\}) \leq \exp\left(-\frac{2t^2}{\frac{1}{n} \sum_{k=1}^n (b_k - a_k)^2}\right) \quad (35.6)$$

$$\mathbb{E}\left(\left\{\bar{S}_n - \mathbb{E}(\bar{S}_n) \geq \frac{t}{\sqrt{n}}\right\}\right) \leq \exp\left(-\frac{2t^2}{\frac{1}{n} \sum_{k=1}^n (b_k - a_k)^2}\right) \quad (35.7)$$

where $\bar{S}_n = S_n/n$.

- **Hoeffding's Inequality (two sided).**

Under the same conditions as the last item, the following are equivalent for $t > 0$:

$$\mathbb{E}(\{|S_n - \mathbb{E}(S_n)| \geq t\sqrt{n}\}) \leq 2 \exp\left(-\frac{2t^2}{\frac{1}{n} \sum_{k=1}^n (b_k - a_k)^2}\right) \quad (35.8)$$

$$\mathbb{E}\left(\left\{|\bar{S}_n - \mathbb{E}(\bar{S}_n)| \geq \frac{t}{\sqrt{n}}\right\}\right) \leq 2 \exp\left(-\frac{2t^2}{\frac{1}{n} \sum_{k=1}^n (b_k - a_k)^2}\right). \quad (35.9)$$

“Tail events” are unlikely or extreme outcomes whose occurrence can have significant implications. Massive asteroid impacts, big stock-market crashes, and pandemics are tail events.¹ But many important examples are less ... Earth shaking: Does a gambler lose all his money? Does the damage from random shocks cause a bridge to collapse? Does an algorithm exhibit worst-case performance on a random input? These are events that are unlikely but not so unlikely as to be implausible or beyond concern. We want to be able to predict such events, for instance to mitigate or manage the risk or at least to be suitably prepared.

In many (even most) cases of interest, it is difficult to compute the exact proba-

¹The values far from the center of a Distribution are often referred to as the “tails” of the Distribution.

bilities of tail events, but for many purposes we do not need an *exact* probability, a sufficiently tight *bound* on the probability is almost as good. A bound is a type of approximation in which we have constrained a quantity to lie within some interval. Here, we will focus on *upper* bounds that tell us that the probability of a tail event cannot be bigger than some threshold. Such bounds can usually be expressed in a form that looks like

$$\mathbb{E}(\{X \text{ extreme}\}) \leq \text{some upper bound}$$

for some random variable X of arbitrary dimension.

This section derives a number of such bounds that arise in practice. The key to these is that the random variable X has a Distribution that is *concentrated* around some particular value, such as its expectation. Concentrated means that most of the mass is near that particular value. Concentration of the mass means that the tails of the Distribution will be “small,” allowing us to bound the mass sufficiently far out in the tails. We thus call the inequalities in this section **concentration inequalities**, but they are also known **tail bounds**.

Concentration inequalities arise in a wide variety of applications, including:

- Bounding risk
- Worst-case-scenario analysis
- Analysis of algorithms
- Evaluation of statistical procedures
- Approximations for probabilities that are otherwise difficult to calculate
- ...

In this section, we will develop a variety of such bounds, inequalities on the probabilities of extreme events, that apply quite widely. All of these bounds are, by academic convention, named after mathematicians that first proved or popularized them: Markov, Chebychev, Chernoff, and Hoeffding. These names serve as reasonable markers for us to organize the identities. These inequalities vary in sharpness (how tightly the bounds must match the true probabilities) and generality (the breadth of situations to which they apply). Typically, the sharper the bound, the stronger the assumptions required, and we will see that here.

A word of reassurance. Several of these inequalities are “symbolically complex,” and seem to have a lot of moving parts. However, this is for the most part a misleading impression. The strategy for using these inequalities with a particular random variable is to put the event whose probability we want into the form that fits the inequality and then simply read off the bounds. Despite the apparent complexity, this is among

the simpler and more automatic techniques we will cover.

Example 35.1.

A friend flips a balanced coin 1000 times and claims to have seen at least 900 heads in those 1000 flips. Assume that all flips are independent and have probability $\frac{1}{2}$ of coming up heads. Do you find your friend's claim surprising?

Define relevant random variables.

- $H_1, H_2, \dots, H_{1000}$ are indicators of heads on successive flips.
- N is the total number of heads in 1000 flips.

State what you know. We know that the H_i 's are independent and have $\mathbb{E}(H_i) = \frac{1}{2}$, with

$$N = \sum_{i=1}^{1000} H_i.$$

So, N has a Binomial $\langle 1000, 1/2 \rangle$ Distribution.

State what you want to find. $\mathbb{E}(\{N \geq 900\})$.

Find it. We can write an exact expression for this

$$\mathbb{E}(\{N \geq 900\}) = \sum_{k=900}^{1000} \binom{1000}{k} 2^{-1000} \approx 6.7 \times 10^{-162},$$

though it is unpleasant to compute. Your friend's result is surprising, more than picking out a particular atom at random out of all the atoms in the observable universe. This is an example of a tail event, both in that the probability is very small and because literally it concerns extreme values of N "in the tail." Can we find a simpler – if approximate – way to understand how likely this extreme event is? You won't be surprised to hear that the answer is yes.

The various inequalities we will cover give:

Inequality	Tail Bound on $\mathbb{E}(\{N \geq 900\}) = 6.7 \times 10^{-162}$
Markov	0.556
Chebychev	0.00078125
Chernoff	9.38725×10^{-57}
Hoeffding	1.06112×10^{-139}

These bounds give increasingly sharp results but require increasingly strong assumptions. We will look at each of these in turn.

Note that Markov's and Chebychev's inequalities give very loose bounds here, but we should not dismiss them as tools because they apply much more broadly than the others.

35.1 Markov and Chebychev Inequalities

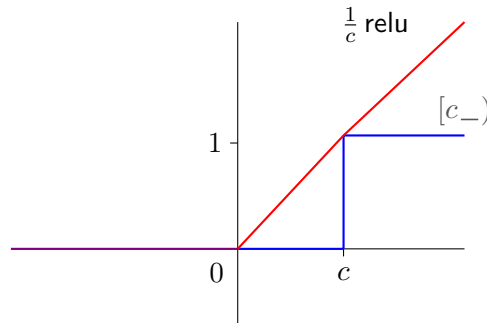
The Expectation Ordering Rule tells us that if random variables U, W satisfy $\text{fact}(U \leq W)$, then $\mathbb{E}(U) \leq \mathbb{E}(W)$. We use this to derive Markov's inequality.

Define the statistic relu by $\text{relu}(x) = x\{x \geq 0\} = \max(x, 0)$. Then for any real number $c > 0$, we have

relu stands for Rectified Linear Unit.

$$[c_-) \leq \frac{1}{c} \text{relu}.$$

You can see this immediately from the graphs of the two functions below.



Now, suppose that X is a non-negative random variable, i.e., $\text{fact}(X \geq 0)$. It follows that for any $c > 0$, $\{X \geq c\} \leq \frac{1}{c} \text{relu}(X)$ or put more succinctly:

$$\{X \geq c\} \leq \frac{1}{c} X. \quad (35.10)$$

Applying the Expectation Ordering Rule to equation (35.10) gives us Markov's Inequality.

Markov's Inequality. If X is a non-negative scalar random variable and $c > 0$, then

$$\mathbb{E}(\{X \geq c\}) \leq \frac{1}{c} \mathbb{E}(X). \quad (35.11)$$

In Example 35.1, we are interested in computing $\mathbb{E}(\{N \geq 900\})$ which is of the form in Markov's Inequality with $c = 900$. Because N is a count, $\text{fact}(N \geq 0)$, and

Markov's Inequality applies with $\mathbb{E}(N) = 500$:

$$\mathbb{E}(\{N \geq 900\}) \leq \frac{500}{900} \approx 0.55556.$$

The bound is correct, but not very tight, as we know that the probability is in fact much smaller.

Markov's Inequality often gives rough bounds like this, but its strength is that it is very general. And it can often be used to derive stronger inequalities.

For example, suppose X is any random variable, of arbitrary dimension, and ψ is a compatible statistic of co-dimension 1 that *returns only non-negative numbers*, i.e., $\psi \geq 0$. Then, Markov's Inequality applies to the scalar random variable $\psi(X)$ and hence

$$\mathbb{E}(\{\psi(X) \geq c\}) \leq \frac{1}{c} \mathbb{E}(\psi(X)). \quad (35.12)$$

For an arbitrary random variable X , let us apply this to the statistic $\psi(x) = |x - \mathbb{E}(X)|^2$. Applying Markov's Inequality with the constant as $c^2 > 0$, equation (35.12) yields

$$\mathbb{E}(\{|X - \mathbb{E}(X)|^2 \geq c^2\}) \leq \frac{1}{c^2} \mathbb{E}(|X - \mathbb{E}(X)|^2)$$

or equivalently,

$$\mathbb{E}(\{|X - \mathbb{E}(X)| \geq c\}) \leq \frac{1}{c^2} \mathbb{E}(|X - \mathbb{E}(X)|^2),$$

because equivalent conditions means equal events.

When X is a scalar, $\mathbb{E}(|X - \mathbb{E}(X)|^2) = \text{Var}(X)$, and in general $\mathbb{E}(|X - \mathbb{E}(X)|^2) = \text{Tr Var}(X)$, where the trace of a square $d \times d$ matrix M , $\text{Tr } M$, is the sum of the diagonal elements,

$$\text{Tr } M = \sum_{i=1}^d M_i^i.$$

When $d = 1$, the trace is just the number itself. This gives us **Chebychev's Inequality**.

If X is a random variable with variance $\text{Var}(X)$, then

$$\mathbb{E}(\{|X - \mathbb{E}(X)| \geq c\}) \leq \frac{1}{c^2} \text{Tr Var}(X). \quad (35.13)$$

Revisiting Example 35.1, we have that $\mathbb{E}(N) = 500$ and $\text{Var}(N) = 1000 \cdot \frac{1}{2} \cdot \frac{1}{2} = 250$.

This statistic works for any dimension. If v has dimension d , $|v| = \sqrt{v_1^2 + v_2^2 + \cdots + v_d^2}$, which reduces to the absolute value of v when $d = 1$.

We are computing the probability of the event $\{N \geq 900\}$, but we have to re-write this in the form that Chebychev's Inequality expects. The inequality gives us

$$\{|N - 500| \geq 400\} \leq \frac{250}{400^2} \approx 0.0015625,$$

but

$$\{|N - 500| \geq 400\} = \{N \geq 900\} + \{N \leq 100\}.$$

Because the mass function (kernel) of N is symmetric around 500 in this case, we know that

$$\mathbb{E}(\{N \geq 900\}) = \mathbb{E}(\{N \leq 100\}),$$

so

$$\begin{aligned} \mathbb{E}(\{N \geq 900\}) &= \frac{1}{2} \mathbb{E}(\{|N - 500| \geq 400\}) \\ &\leq 0.00078125, \end{aligned}$$

as shown in the table above.

35.2 Chernoff Bounds

We can extend Markov's Inequality even more effectively with a different statistic in equation (35.12). For each $t \geq 0$, define $\psi_t(x) = e^{tx}$. When $t = 0$, this is just the constant statistic returning 1. For $t > 0$, this is a positive statistic and is increasing in x . In that case, it follows that

$$\{X \geq c\} = \{e^{tX} \geq e^{tc}\} = \{\psi_t(X) \geq e^{tc}\},$$

and we can apply Markov to the positive variable $\psi_t(X)$.

$$\mathbb{E}(\{X \geq c\}) = \mathbb{E}(\{\psi_t(X) \geq e^{tc}\}) \leq e^{-tc} \mathbb{E}(e^{tX}).$$

This gives us a bound for every positive t , all of which hold (since the left-hand side doesn't depend on t), and the same bound holds for $t \geq 0$ as probabilities cannot be bigger than 1. So we are free to choose the best one. The result is the basic Chernoff bound.

Let X be a scalar random variable; then the **basic Chernoff bound** is given by:

$$\mathbb{E}(\{X \geq c\}) \leq \min_{t \geq 0} e^{-tc} \mathbb{E}(e^{tX}). \quad (35.14)$$

To use this bound, we must find the minimizing t on a case-by-case basis.

The mapping $\langle t \rangle \mapsto \mathbb{E}(e^{tX})$ has some special properties. It is called the *moment generating function* of X , $\mathbf{M}_X$, and is described in more detail in subsection 35.2 below.

Example 35.2. Let W be the time a commuter must wait until her bus arrives, measured in minutes. Assume that W has an Exponential $\langle 1/10 \rangle$ Distribution. Find a bound on the probability that the commuter waits at least an hour.

Because we can compute the needed probability exactly, we can evaluate the effectiveness of the bounds.

First, suppose W has a general Exponential $\langle \lambda \rangle$ Distribution. As shown in Example 35.7 below,

$$\mathbb{E}(e^{-tW}) = \frac{\lambda}{\lambda - t} \{0 \leq t < \lambda\} + \infty \{t \geq \lambda\}.$$

Minimizing $e^{-tc} \mathbb{E}(e^{-tW})$ over t (by setting derivatives to zero where it is defined), we get the minimizer $t_* = \frac{c\lambda-1}{c}$, giving the best Chernoff bound of $\lambda c e^{1-c\lambda}$. Hence,

$$\mathbb{E}(\{W \geq c\}) \leq \lambda c e^{1-c\lambda}.$$

Markov's inequality gives a bound of $\frac{1}{\lambda c}$.

Now, take $c = 60$ and $\lambda = 1/10$, we have $c\lambda = 6$. The Chernoff bound is about 0.0404, and the Markov bound is about 0.1667.

The exact answer is

$$\begin{aligned} \mathbb{E}(\{W \geq 60\}) &= \int_{t=60}^{\infty} dt \lambda e^{-\lambda t} \\ &= \int_{u=6}^{\infty} du e^{-u} \\ &= e^{-6} \approx 0.0025. \end{aligned}$$

Both bounds are conservative, but the Chernoff bound is four times better than the Markov bound.

The basic Chernoff bound is very general. We can get sharper inequalities from the same logic if we make stronger assumptions. Assume, for instance, that $S_n = \sum_{i=1}^n X_i$, where the X_i 's are *independent* random variables. Then,

$$M_{S_n}(t) = \mathbb{E}(e^{tS_n}) = \mathbb{E}(e^{t\sum_i X_i}) = \prod_{i=1}^n \mathbb{E}(e^{tX_i}) = \prod_{i=1}^n M_{X_i}(t).$$

The basic Chernoff bound gives us the **Chernoff bound for sums of independent random variables**:

$$\mathbb{P}(\{S_n \geq c\}) \leq \min_{t>0} e^{-tc} \prod_{i=1}^n \mathbb{E}(e^{tX_i}). \quad (35.15)$$

When the X_i 's are independent *events* with $\mathbb{E}(X_i) = p_i$, then

$$\mathbb{E}(e^{tX_i}) = (1 - p_i) + p_i e^t = 1 + (e^t - 1)p_i \leq \exp(p_i(e^t - 1)),$$

and the Chernoff bound is

$$\begin{aligned} \mathbb{E}(e^{tS_n}) &= \prod_{i=1}^n \mathbb{E}(e^{tX_i}) \\ &\leq \exp\left((e^t - 1) \sum_{i=1}^n p_i\right) \\ &\leq \exp((e^t - 1)\mathbb{E}(S_n)) \\ \mathbb{P}(\{S_n \geq c\}) &\leq \min_{t>0} \exp((e^t - 1)\mathbb{E}(S_n) - ct). \end{aligned}$$

Setting $c = a \mathbb{E}(S_n)$ for $a > 1$,

$$\mathbb{P}(\{S_n \geq a \mathbb{E}(S_n)\}) \leq \min_{t>0} \exp(\mathbb{E}(S_n)(e^t - at - 1)).$$

The optimal bound is given by

$$\min_{t>0} \exp(\mathbb{E}(S_n)(e^t - at - 1)) = \exp\left(\mathbb{E}(S_n) \min_{t>0} (e^t - at - 1)\right).$$

This is minimized at $t = \log(a)$, which yields

The **Chernoff bound for a sum independent events**:

$$\mathbb{P}(\{S_n \geq a \mathbb{E}(S_n)\}) \leq e^{-(a \log a - a + 1) \mathbb{E}(S_n)}. \quad (35.16)$$

Example 35.3 Admissions Officer

You are an admissions officer at a small but respected research university, and you issue invitations to fill the incoming class.

Even though slots at your school are highly sought after, some prospective students will choose a competing institution. You can accept up to 999 students before the cost of accommodating the extra students becomes prohibitive.

Based on years of experience in the position, you expect 850 of those invited to accept. Give an upper bound on the probability that the incoming class is too large.

Let A_j be the event that student j accepts and let's assume that the A_j 's are independent.

Let $N = \sum_{j=1}^n A_j$ be the number of students who accept.

We need a bound on $\mathbb{E}(\{N \geq 1000\})$.

Here we don't know n or $\mathbb{E}(A_j)$, but we do know $\mathbb{E}(N) = 850$ and that's enough. We have

$$\{N \geq 1000\} = \left\{ N \geq \frac{1000}{850} \mathbb{E}(N) \right\}.$$

So we can use the Chernoff bound for Sums of Events with $a = 1000/850$:

$$\begin{aligned} \mathbb{E}(\{N \geq 1000\}) &= \mathbb{E}\left(\left\{ N \geq \frac{1000}{850} \mathbb{E}(N) \right\}\right) \\ &\leq \exp\left(-\left(\frac{1000}{850} \log\left(\frac{1000}{850}\right) - \frac{1000}{850} + 1\right) \cdot 850\right) \\ &= \exp\left(-\left(1000 \log\left(\frac{1000}{850}\right) - 150\right)\right) \\ &\approx 3.66 \times 10^{-6}. \end{aligned}$$

Aside: The Moment Generating Function

In this subsection, we define and discuss the function $M_X(t) = \mathbb{E}(e^{t \cdot X})$, which is called the **Moment Generating Function** (MGF) of a random variable X . It is a determining function² that can be used to find or characterize the Distribution of X . This subsection is optional material; unless it has been assigned explicitly, you can skip this on first read through.

If X is a scalar random variable and k is a positive integer, then the expectation $\mathbb{E}(X^k)$ is called the k^{th} **moment** of X . As with any expectation, moments are properties of the Kind/Distribution. For example, with a Distribution that is symmetric

²See Sections 6.3 and 25.3.

around zero, odd moments will vanish. Also, when X is non-negative and $k \geq j$, the k^{th} gives more weight to the larger possible values of X than does the j^{th} moment. The moments, therefore, are summaries that (partially) describe how the Distribution allocates mass.

Definition 64. The **moment generating function (MGF)** of a random variable X , M_X , is defined by

$$M_X(t) = \mathbb{E}e^{t \cdot X}. \quad (35.17)$$

If X is a scalar random variable, t is a real number. If X has dimension $d > 1$, t is a d -vector and $t \cdot X = t_1X_1 + t_2X_2 + \cdots + t_dX_d$ is the vector dot product.

The MGF may be infinite for some values of t . In the language of engineering and some other fields, the MGF is called the Laplace transform of the Distribution.

In most situations, we care more about the function M_X as a whole rather than its value at specific t . For most purposes, all we need to use the MGF is that it be well defined for in at least a small neighborhood of 0, i.e., $t \in (-\epsilon, \epsilon)$ for some $\epsilon > 0$ however small. When $X \geq 0$, for example, $M_X(t)$ is finite for all $t \leq 0$; in order for it to be defined for at least small $t > 0$, the kernel of X must decay faster than an exponential function towards large values.

Example 35.4 mgf Not Defined

Let U be a continuous, positive random variable with $K_U(x) = x^{-2}\{x \geq 1\}$. U 's kernel decays as the reciprocal of a polynomial which is much slower than the decay of an exponential function. Hence, for $t > 0$,

$$M_U(t) = \int_{y=1}^{\infty} dy \, e^{ty} y^{-2} = \infty. \quad (35.18)$$

Even though the MGF is finite for negative values, it is not defined in an interval around zero. So, we say that M_U is not well defined.

Example 35.5 The Standard Normal Distribution

Let Z be a Normal $\langle 0, 1 \rangle$ random variable. Then,

$$\begin{aligned}
 M_Z(t) &= \int_u du \, K_Z(u) e^{tu} \\
 &= \int_u du \, \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2} e^{tu} \\
 &= \int_u du \, \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(u^2 - 2tu + t^2)} e^{\frac{1}{2}t^2} du && \text{Complete the square; multiply by } e^{\frac{1}{2}t^2} e^{-\frac{1}{2}t^2}. \\
 &= e^{\frac{1}{2}t^2} \int_u du \, \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(u-t)^2} du && \text{The Normal}\langle t, 1 \rangle \text{ kernel integrates to 1.} \\
 &= e^{\frac{1}{2}t^2}.
 \end{aligned}$$

This is defined for all real values t .

Example 35.6 The Poisson Distribution

Let N be a Poisson $\langle \lambda \rangle$ random variable with $\lambda > 0$. This is a discrete random variable taking values in the non-negative integers with kernel:

$$K_N(n) = e^{-\lambda} \frac{\lambda^n}{n!} \{n \in [0..)\}.$$

Therefore,

$$\begin{aligned}
 M_N(t) &= \sum_{n=0}^{\infty} e^{tn} e^{-\lambda} \frac{\lambda^n}{n!} \\
 &= e^{-\lambda} \sum_{n=0}^{\infty} \frac{(\lambda e^t)^n}{n!} \\
 &= e^{-\lambda} e^{\lambda e^t} && \text{By the power series for } e^x. \\
 &= e^{\lambda(e^t - 1)}.
 \end{aligned}$$

This is defined for all real t .

The properties of the MGF make it useful for characterizing the Distribution or finding moments when the kernel is complicated to use. First, the MGF determines the Distribution uniquely, so two random variables with the same MGF have the same Distribution. Second, the MGF of a sum of independent random variables is the product of the MGFs because

$$\mathbb{E}e^{t(X_1 + \dots + X_n)} = \mathbb{E}e^{tX_1} \dots e^{tX_n} = \mathbb{E}e^{tX_1} \dots \mathbb{E}e^{tX_n}. \quad (35.19)$$

For example, we can combine these two properties to learn something that would be hard to find otherwise. Suppose that $N_1, \dots, N_r$ are independent random variables such that N_i has a $\text{Poisson}\langle\lambda_i\rangle$ Distribution, for $\lambda_1, \dots, \lambda_r > 0$. Define $N = N_1 + \dots + N_r$ and $\lambda = \lambda_1 + \dots + \lambda_r$. Then,

$$M_N(t) = \prod_{i=1}^n M_{N_i}(t) = \prod_{i=1}^n e^{\lambda_i(e^t-1)} = e^{\lambda(e^t-1)}.$$

It follows that the sum of independent Poisson random variables also has a Poisson Distribution, with parameter equal to the sum of the individual terms' parameters.

Third, we obtain moments of the Distribution by taking derivatives of the MGF. Assuming that derivatives with respect to t can pass inside the expected value, we get $M'_X(t) = \mathbb{E}(Xe^{tX})$ and similarly $M_X^{(k)}(t) = \mathbb{E}(X^k e^{tX})$ for $k \geq 1$. Hence, $\mathbb{E}(X) = M'_X(0)$ and $\mathbb{E}(X^k) = M_X^{(k)}(0)$.

Finally, we can relate the MGF of $aX + b$, for scalar constant a and constant b with the same dimension as X . Specifically,

$$\begin{aligned} M_{aX+b}(t) &= \mathbb{E}(e^{t \cdot (aX+b)}) \\ &= e^{bt} \mathbb{E}(e^{(at) \cdot X}) \\ &= e^{bt} M_X(at). \end{aligned}$$

This is an example of how an operation on the random variable behaves very differently in transformed coordinates of the MGF. Using the second to last example, if X is a $\text{Normal}\langle\mu, \sigma^2\rangle$ random variable, we know that there is a standard Normal variable Z such that $X = \mu + \sigma Z$. Hence,

$$M_X(t) = e^{\mu t + \frac{1}{2}\sigma^2 t^2}. \quad (35.20)$$

Combining this with the other properties, we see that if $X_1, \dots, X_n$ are independent random variables where X_i has a $\text{Normal}\langle\mu_i, \sigma_i^2\rangle$ Distribution, then $Y = X_1 + \dots + X_n$ has

$$M_Y(t) = \prod_{i=1}^n M_{X_i}(t) = \prod_{i=1}^n e^{\mu_i t + \frac{1}{2}\sigma_i^2 t^2} = e^{\mu t + \frac{1}{2}\sigma^2 t^2},$$

where $\mu = \sum_i \mu_i$ and $\sigma^2 = \sum_i \sigma_i^2$. Thus, the sum of independent normal random variables are also Normal with parameters that are sums of the individual terms parameters.

Properties of the MGF

- Random variables X and Y have the same Distribution (i.e., $D_X = D_Y$) if and only if $M_X = M_Y$.
- If $X_1, \dots, X_n$ are independent random variables, then

$$M_{X_1 + \dots + X_n} = \prod_{i=1}^n M_{X_i}. \quad (35.21)$$

- For integer $k \geq 1$,

$$\mathbb{E}(X^k) = \left. \frac{d^k}{dt^k} M_X(t) \right|_{t=0}. \quad (35.22)$$

- If a is a scalar constant and b is a constant with the same dimension as X , then

$$M_{aX+b}(t) = e^{b \cdot t} M_X(at). \quad (35.23)$$

Example 35.7.

Let L be the lifetime of some circuit component and assume that L has an Exponential $\langle \lambda \rangle$ Distribution. Then,

$$\begin{aligned} M_L(u) &= \int_u du \, K_L(x) e^{ux} \\ &= \int_{x=0}^{\infty} dx \, \lambda e^{-\lambda x} e^{ux} \\ &= \frac{\lambda}{\lambda - u} \int_0^{\infty} dx \, (\lambda - u) e^{-(\lambda - u)x} \\ &= \begin{cases} \frac{\lambda}{\lambda - u} & \text{if } u < \lambda \\ \infty & \text{otherwise.} \end{cases} \end{aligned}$$

So, M_L is defined for all $u < \lambda$ and thus in an interval containing 0. For such u , we can write

$$M_L(u) = \frac{\lambda}{\lambda - u} = \frac{1}{1 - \frac{u}{\lambda}} = \sum_{k=0}^{\infty} \frac{u^k}{\lambda^k} = \sum_{k=0}^{\infty} k! \lambda^{-k} \frac{u^k}{k!}. \quad (35.24)$$

It follows that $\mathbb{E}(L^k) = k! \lambda^{-k}$. Specifically, $\mathbb{E}(X) = 1/\lambda$.

35.3 Hoeffding's Inequality

Finally, we consider an inequality that is highly specialized but often provides a tight bound. Hoeffding's Inequality applies to *sums of independent, bounded random variables*. It comes in two flavors depending on whether we want to bound the probability of a tail event in both directions (two-sided version) or only one direction (one-sided version). What makes Hoeffding's Inequality powerful is that it gives us exponentially small bounds for events sufficiently in the tails, so when it applies it tends to give much better results than Markov or Chebychev, for instance.

Let $S_n = \sum_{k=1}^n X_k$, where the X_k 's are independent, scalar random variables and write $\bar{S}_n = S_n/n$ for the average. We assume that the X_k 's satisfy $\mathbb{E}(\{a_k \leq X_k \leq b_k\}) = 1$ for some $-\infty < a_k \leq b_k < \infty$. The $\langle a_k, b_k \rangle$ can differ across the X_k 's but they must all be finite. Moreover, the bounds get better the smaller the $b_k - a_k$'s are.

We state two versions of the inequality. In the one-sided version, we are concerned with the possibility that $S_n \geq c$ for some $c > \mathbb{E}(S_n)$ in *one direction*. We express this as the event $\{S_n - \mathbb{E}(S_n) \geq t\sqrt{n}\}$ for some constant $t > 0$, or equivalently in terms of the averages $\{\bar{S}_n - \mathbb{E}(\bar{S}_n) \geq t/\sqrt{n}\}$. These events occur when the sum (or average) is *bigger than* its expectation by at least the specified amount.

In the two-sided version, we are concerned with the possibility that S_n can deviate in both directions from its expectation. We express this as the event $\{|S_n - \mathbb{E}(S_n)| \geq t\sqrt{n}\}$ for some constant $t > 0$, or equivalently in terms of the averages $\{|\bar{S}_n - \mathbb{E}(\bar{S}_n)| \geq t/\sqrt{n}\}$. These events occur when the sum (or average) is *either bigger or smaller than* its expectation by at least the specified amount.

Hoeffding's inequality (one-sided) Let $S_n = \sum_{k=1}^n X_k$, where X_k are independent random variables and satisfy $\mathbb{E}(\{a_k \leq X_k \leq b_k\}) = 1$ for some $-\infty < a_k \leq b_k < \infty$. Then for $t > 0$,

$$\mathbb{E}(\{S_n - \mathbb{E}(S_n) \geq t\sqrt{n}\}) \leq \exp\left(-\frac{2t^2}{\frac{1}{n} \sum_{k=1}^n (b_k - a_k)^2}\right), \quad (35.25)$$

or equivalently

$$\mathbb{E}\left(\left\{\bar{S}_n - \mathbb{E}(\bar{S}_n) \geq \frac{t}{\sqrt{n}}\right\}\right) \leq \exp\left(-\frac{2t^2}{\frac{1}{n} \sum_{k=1}^n (b_k - a_k)^2}\right). \quad (35.26)$$

We get the two-sided bound from the one-sided bound because

$$\begin{aligned}\{|S_n - \mathbb{E}(S_n)| \geq c\} &= \{S_n - \mathbb{E}(S_n) \geq c\} + \{S_n - \mathbb{E}(S_n) \leq -c\} \\ &= \{S_n - \mathbb{E}(S_n) \geq c\} + \{(-S_n) - \mathbb{E}(-S_n) \geq c\}\end{aligned}$$

Denote $Z_n = -S_n$ and $c = t\sqrt{n}$ and take expectations

$$\mathbb{E}(\{|S_n - \mathbb{E}(S_n)| \geq t\sqrt{n}\}) = \mathbb{E}(\{S_n - \mathbb{E}(S_n) \geq t\sqrt{n}\}) + \mathbb{E}(\{Z_n - \mathbb{E}(Z_n) \geq t\sqrt{n}\})$$

Notice that the same Hoeffding bound applies for Z_n as for S_n . This gives us the two-sided bound.

Hoeffding's inequality (two-sided) Let $S_n = \sum_{k=1}^n X_k$, where X_k are independent random variables and satisfy $\mathbb{E}(\{a_k \leq X_k \leq b_k\}) = 1$ for some $-\infty < a_k \leq b_k < \infty$. Then for $t > 0$,

$$\mathbb{E}(\{|S_n - \mathbb{E}(S_n)| \geq t\sqrt{n}\}) \leq 2 \exp\left(-\frac{2t^2}{\frac{1}{n} \sum_{k=1}^n (b_k - a_k)^2}\right). \quad (35.27)$$

or equivalently

$$\mathbb{E}\left(\left\{|\bar{S}_n - \mathbb{E}(\bar{S}_n)| \geq \frac{t}{\sqrt{n}}\right\}\right) \leq 2 \exp\left(-\frac{2t^2}{\frac{1}{n} \sum_{k=1}^n (b_k - a_k)^2}\right). \quad (35.28)$$

The right-hand sides of the one-sided and two-sided inequalities differ only by a factor of 2.

Example 35.8. Let $X_1, X_2, \dots$ be independent Uniform $\langle -1, 1 \rangle$ random variables.

Then, $\mathbb{E}(X_k) = 0$ and $\text{Var}(X_k) = 1/3$.

Let $S_n = \sum_{k=1}^n X_k$. Find a bound for $\mathbb{E}(\{S_{10000} \geq 100\})$ and for $\mathbb{E}(\{S_{10000} \geq 500\})$.

How do we solve this? Look at the inequalities we have and match the pattern.

1. $-1 \leq X \leq 1$ is always true, and $\frac{1}{n} \sum_{k=1}^n (1 - (-1))^2 = 4$.

2. $\mathbb{E}(S_n) = 0$

3. We want to bound $\mathbb{E}(\{S_n - \mathbb{E}(S_n) \geq t\sqrt{n}\})$ for $n = 10,000$ and for $t = 1$ and $t = 5$, respectively.

This points to Hoeffding's inequality (one-sided). (If we wanted $\mathbb{E}(\{|S_{10000}| \geq 100\})$, we would use the two-sided version.) We simply plug this information into the formula:

$$\begin{aligned}\mathbb{E}(\{S_{10000} \geq 100\}) &= \mathbb{E}\left(\left\{S_{10000} - \mathbb{E}(S_{10000}) \geq 1 \cdot \sqrt{10000}\right\}\right) \\ &\leq e^{-\frac{2 \cdot 1^2}{4}} = e^{-0.5} \approx 0.6065. \\ \mathbb{E}(\{S_{10000} \geq 500\}) &= \mathbb{E}\left(\left\{S_{10000} - \mathbb{E}(S_{10000}) \geq 5 \cdot \sqrt{10000}\right\}\right) \\ &\leq e^{-\frac{2 \cdot 5^2}{4}} = e^{-12.5} \approx 3.727 \times 10^{-6}. \\ \mathbb{E}(\{S_{10000} \geq t \cdot 100\}) &= \mathbb{E}\left(\left\{S_{10000} - \mathbb{E}(S_{10000}) \geq t \cdot \sqrt{10000}\right\}\right) \\ &\leq e^{-\frac{2 \cdot t^2}{4}} = e^{-\frac{1}{2}t^2}. \\ \mathbb{E}(\{|S_{10000}| \geq t \cdot 100\}) &\leq 2e^{-\frac{1}{2}t^2}.\end{aligned}$$

We actually know something else in this problem: by the Normal Approximation, S_{10000} has approximately a Normal $\langle 0, \sigma^2 \rangle$ Distribution, with $\sigma^2 = 10000/3 = (100/\sqrt{3})^2$. So,

$$\begin{aligned}\mathbb{E}(\{S_{10000} \geq t \cdot 100\}) &= \mathbb{E}\left(\left\{\frac{S_{10000}}{\sigma} \geq \frac{t \cdot 100}{\sigma}\right\}\right) \\ &= \mathbb{E}\left(\left\{\frac{S_{10000}}{\sigma} \geq t\sqrt{3}\right\}\right) \\ &\approx 1 - \Phi(t\sqrt{3}). \\ \mathbb{E}(\{S_{10000} \geq 100\}) &= 1 - \Phi(\sqrt{3}) \approx 0.0416 \\ \mathbb{E}(\{S_{10000} \geq 500\}) &= 1 - \Phi(5\sqrt{3}) \approx < 10^{-16},\end{aligned}$$

showing that Hoeffding's Inequality is quite tight. We also can use another tail bound that derives from Markov, called *Mill's Inequality*:

$$1 - \Phi(t) \leq \frac{1}{t} \Phi'(t) = \frac{1}{t\sqrt{2\pi}} e^{-\frac{1}{2}t^2},$$

for "large" t . So, when t is large,

$$\mathbb{E}(\{S_{10000} \geq t \cdot 100\}) \approx 1 - \Phi(t\sqrt{3}) \leq \frac{1}{t\sqrt{6\pi}} e^{-\frac{3}{2}t^2} \leq e^{-\frac{1}{2}t^2},$$

which is similar.

35.4 More Examples

Example 35.9. A friend flips a balanced coin 100 times and claims to have seen at least 75 heads in those 100 flips. Do you find that claim surprising? Assume that all flips are independent and have probability $\frac{1}{2}$ of coming up heads.

Define relevant random variables.

- $H_1, H_2, \dots, H_{100}$ are indicators of heads on successive flips.
- N is the total number of heads in 100 flips.

State what you know.

We know that the H_i 's are independent and have $\mathbb{E}(H_i) = \frac{1}{2} = \mathbb{E}(\{H_i = 1\})$. We also know that

$$N = \sum_{i=1}^{100} H_i$$

State what you want to find.

$$\mathbb{E}(\{N \geq 75\})$$

Find it. $\mathbb{E}(\{N \geq 75\}) = \sum_{k=75}^{100} \mathbb{E}(\{N = k\})$ seems inconvenient to calculate by hand. Let's find an upper bound instead.

Let's try Markov's inequality:

$$\mathbb{E}(\{N \geq 75\}) \leq \frac{\mathbb{E}(N)}{75} = \frac{2}{3}$$

This is not a particularly useful bound. It does not resolve whether $\mathbb{E}(\{N \geq 75\})$ is small or large since it might be that the bound is loose.

Since N is a sum of independent indicators, we know that its Chernoff bound is

$$\mathbb{E}(\{N \geq a \mathbb{E}(N)\}) \leq \exp(-(a \log a - a + 1) \mathbb{E}(N))$$

We know that $\mathbb{E}(N) = 50$ and want $\mathbb{E}(\{N \geq 75\})$, so we set $a = \frac{3}{2}$. This yields

$$\begin{aligned} \mathbb{E}(\{N \geq 75\}) &\leq \exp(-(a \log a - a + 1) \mathbb{E}(N)) \\ &= \exp\left(-\left(\frac{3}{2} \log\left(\frac{3}{2}\right) - \frac{1}{2}\right) \cdot 50\right) \\ &\approx 4.5 \cdot 10^{-3}. \end{aligned}$$

A vastly different answer than what we got from Markov!

Since N is a sum of bounded and independent random variables, we can also apply Hoeffding's inequality. Here $a_k = 0$ and $b_k = 1$, so Hoeffding gives

$$\mathbb{E}(\{N - \mathbb{E}(N) \geq t\sqrt{n}\}) \leq e^{-2t^2}$$

Substituting $\mathbb{E}(N) = 50$, $n = 100$, and $t = 2.5$,

$$\mathbb{E}(\{N \geq 75\}) = \mathbb{E}(\{N - 50 \geq 2.5 \cdot 10\}) \leq e^{-2(2.5)^2} = e^{-12.5} \approx 3.73 \times 10^{-6}.$$

This is even better than Chernoff!

The exact answer obtained from the Binomial $\langle 100, 1/2 \rangle$ Distribution is $\mathbb{E}(\{N \geq 75\}) \approx 2.8 \cdot 10^{-7}$.

Example 35.10. You are designing a multiple choice exam. The exam has n questions and each has 4 possible answers. To pass the exam, a student must get at least half of the questions right. You want to make sure that a student with no knowledge of the subject has at most 1 % chance of passing the exam. How many questions should the exam have?

Let's assume that a student with no knowledge will choose his/her answers independently for each question and in such a way that each answer is equally likely to be chosen.

Let C_j be the event that question j is correct. We have $\mathbb{E}(\{C_j = 1\}) = \mathbb{E}(C_j) = \frac{1}{4}$.

Let $N = \sum_{j=1}^n C_j$ be the number of correct answers. We have $\mathbb{E}(N) = \frac{n}{4}$. We want to choose n such that $\mathbb{E}(\{N \geq \frac{n}{2}\}) \leq \frac{1}{100}$.

Let's first try Markov:

$$\mathbb{E}(\{N \geq \frac{n}{2}\}) \leq \frac{\mathbb{E}(N)}{\frac{n}{2}} = \frac{\frac{n}{4}}{\frac{n}{2}} = \frac{1}{2}.$$

This won't help us choose n .

Let's rewrite

$$\mathbb{E}(\{N \geq \frac{n}{2}\}) = \mathbb{E}(\{N \geq 2\mathbb{E}(N)\})$$

This is of the same form as the Chernoff bound, so let's try that:

$$\begin{aligned}\mathbb{E}\left(\left\{N \geq \frac{n}{2}\right\}\right) &= \mathbb{E}(\{N \geq 2\mathbb{E}(N)\}) \\ &\leq \exp(-(2\log(2) - 2 + 1)\mathbb{E}(N)) \\ &= \exp\left(-\frac{n}{4}(2\log 2 - 1)\right)\end{aligned}$$

This is more helpful since now the bound depends on n . To choose n , we need to require

$$\exp\left(-\frac{n}{4}(2\log 2 - 1)\right) \leq \frac{1}{100}$$

This implies $n \geq \lceil \frac{4\log 100}{2\log 2 - 1} \rceil = 48$.

Interlude Z

Generating Functions

The Power of Alternate Representations

36

Chapter

Contents

36.1 Counting Configurations	1152
36.2 Power Series and Their Coefficients	1153
36.3 Operating on Generating Functions	1154
36.4 Distribution Example	1155

36.1 Counting Configurations

Coming Soon...

36.2 Power Series and Their Coefficients

Coming Soon...

36.3 Operating on Generating Functions

Coming Soon...

36.4 Distribution Example

Coming Soon...

Probability Generating Functions

37

Chapter

Contents

37.1 Basics	1158
37.2 Distributional Properties	1159
37.3 Operating on Generating Functions	1160

37.1 Basics

Coming Soon...

37.2 Distributional Properties

Coming Soon...

37.3 Operating on Generating Functions

Coming Soon...

Moment Generating Functions

38

Chapter

Contents

38.1 Basics	1162
38.2 Distributional Properties	1163
38.3 Operating on MGFs	1164
38.4 Cumulant Generating Functions	1165

38.1 Basics

Coming Soon...

38.2 Distributional Properties

Coming Soon...

38.3 Operating on MGFs

Coming Soon...

38.4 Cumulant Generating Functions

Coming Soon...

Characteristic Generating Functions

39

Chapter

Contents

39.1 A Primer on Complex Numbers	1168
39.2 Basics: The Fourier Transform	1169
39.3 Distributional Properties	1170
39.4 Operating on CGFs	1171

39.1 A Primer on Complex Numbers

Coming Soon...

39.2 Basics: The Fourier Transform

Coming Soon...

39.3 Distributional Properties

Coming Soon...

39.4 Operating on CGFs

Coming Soon...

Part IV

Stochastic Processes

Coming Soon...

Part V

Information and Entropy

Coming Soon...

Interlude G

Graphs

Coming Soon...

Part VI

Learning from Data

Probability versus Statistics

40

Chapter

Contents

40.1 Forward and Inverse Problems	1182
40.2 Statistical Models	1183
40.3 What does it mean to learn from data?	1184
40.4 The Frequentist Framework	1185
40.5 The Bayesian Framework	1186

40.1 Forward and Inverse Problems

Coming Soon...

40.2 Statistical Models

Coming Soon...

40.3 What does it mean to learn from data?

Coming Soon...

40.4 The Frequentist Framework

Coming Soon...

40.5 The Bayesian Framework

Coming Soon...

Estimation and Inference

41

Chapter

Contents

41.1 Statistical Problems	1188
41.2 Maximum Likelihood Estimation	1189
41.3 Bayes and Other Estimation Methods	1190
41.4 Confidence Sets	1191
41.5 Posterior Distributions	1192

41.1 Statistical Problems

Coming Soon...

41.2 Maximum Likelihood Estimation

Coming Soon...

41.3 Bayes and Other Estimation Methods

Coming Soon...

41.4 Confidence Sets

Coming Soon...

41.5 Posterior Distributions

Coming Soon...

Regression

42

Chapter

Contents

42.1 The Regression Problem	1194
42.2 Smoothing	1195
42.3 Linear Regression	1196
42.4 A Geometric Perspective	1197
42.5 Neural Networks and More	1198

42.1 The Regression Problem

Coming Soon...

42.2 Smoothing

Coming Soon...

42.3 Linear Regression

Coming Soon...

42.4 A Geometric Perspective

Coming Soon...

42.5 Neural Networks and More

Coming Soon...

Classification

43

Chapter

Contents

43.1 The Classification Problem	1200
43.2 High-Dimensional Space	1201
43.3 Methods	1202

43.1 The Classification Problem

Coming Soon...

43.2 High-Dimensional Space

Coming Soon...

43.3 Methods

Coming Soon...

Index

- O_P , [1101](#)
- 2 , [570](#)
- $2^{\mathcal{A}}$ (power set), [570](#)
- $\binom{\mathcal{A}}{k}$ (subsets of size), [576](#)
- $::$, [675](#)
- $\leftarrow$, [73](#)
- M_X , [1137](#)
- o_P , [1101](#)
- O_P , [1100](#), [1101](#)
- o_P , [1100](#), [1101](#)
- DiscreteUniform($\mathcal{V}$), [895](#)
- FRP
 - dimension, [16](#)
 - size, [16](#)
 - type, [16](#)
 - width, [16](#)
- frplib
 - uniform, [23](#)
-], [898](#)
- algebraic laws
 - associativity, [700](#)
 - commutativity, [700](#)
 - identity element, [700](#)
 - unit element, [700m](#)
- approximate Distribution, [1094](#)
- arbitrage price, [386](#), [389](#)
- associativity, [231](#), [237](#), [240](#)
- asymptotic order, [786](#)
- bag, *see* multiset
- basis, [751](#)
- Bayes's Rule, [361](#), [362](#), [365](#), [368](#)
- Bernoulli Trials Process, [1079](#)
- Big 3+1, [69](#), [81](#)
- big-O, [786](#)
- bijection, [407](#)
- binomial coefficient, [801](#)
- Binomial Distribution, [1102](#)
- binomial theorem, [803](#)
- bit, [398](#), [423](#)
- Boolean expression, [942](#), [943](#)
- Boolean expressions, [336](#)
- bound
 - lower, [790](#)
 - upper, [790](#)
- canonical families, [1102](#)
- canonical form, [204](#)
- CDF, [972](#), [1025](#)
- Central Limit Theorem, [531](#), [1093](#),
[1120](#)
- chain rule, [776](#)
- Chebychev's Inequality, [1133](#)
- Chernoff bound
 - basic, [1135](#)
 - sum of events, [1136](#)
 - sums, [1136](#)
- clone, [256](#)
- clones, [225](#)
- CLT, *see* Central Limit Theorem
- codimension, [271](#)
- collider, [1055](#)
- combinator
 - statistic, [79](#), [95](#), [98](#), [108](#), [114](#), [177](#),
[305](#)
- common cause, [1054](#)
- common effect, [1055](#)
- commutative diagram, [171](#), [628](#)
- comparable, [780](#)
- compatible, [116](#), [323](#)
- complementary event, [923](#), [924](#)
- complex conjugate, [769](#)
- complex numbers, [771](#)
 - polar form, [771](#)

- component projection, [673](#)
- composition, [98](#), [620](#)
 - diagrammatic order, [621](#)
 - pipeline order, [98](#), [621](#)
 - syntactic order, [98](#), [621](#)
- concentration inequalities, [1130](#)
- condition, [78](#), [94](#), [94](#), [100](#), [114](#), [329](#), [334](#), [337](#), [939](#), [941](#)
- conditional FRP
 - join, [275](#)
- conditional FRP, [259](#), [261](#), [270](#)
 - codimension, [264](#)
 - compatibility, [273](#)
 - dimension, [265](#)
 - domain, [264](#)
 - target, [264](#)
 - target dimension, [265](#)
 - type, [265](#)
- conditional FRP, FRP
 - conditional, [264](#)
- conditional Kind, [265](#), [268](#), [270](#)
 - codimension, [268](#)
 - compatibility, [273](#)
 - dimension, [268](#)
 - domain, [268](#)
 - join, [281](#)
 - target, [268](#), [845](#)
 - target dimension, [268](#)
 - type, [268](#)
- conditioning, [315](#), [318](#), [323](#)
 - operator, [316](#)
- Constant Approximation, [1093](#), [1099](#), [1107](#), [1108](#)
- constraint processor, [78](#), [338](#), [360](#)
- continuity correction, [1116](#)
- continuous, [866](#), [871](#)
- convex combination, [583](#)
- convolution, [727](#)
- coordinates
 - polar, [901](#)
- correlation, [1038](#), [1044](#)
- countably infinite, [865](#)
- covariance, [1038](#), [1043](#)
- covector, [749](#), [750](#)
- Cumulative Distribution Function, [1025](#)
- cumulative distribution function, [844](#)
- cycle
 - permutation, [674](#), [674](#)
- data, [39](#)
- data-question, [314](#)
- deconstruct, [111](#)
- decorator, [74](#), [101](#), [102](#), [128](#), [152](#)
- DeMorgan's Laws, [549](#)
- determining class, [427](#), [1024](#)
- determining function, [1025](#)
- determining functions method, [1028](#)
- deterministic, [39](#)
- diagram
 - commutes, [628](#)
- differentials, [878](#)
- digraph, [574](#)
- dilation, [186](#)
- dimension, [16](#), [16](#), [271](#)
- discrete, [866](#), [871](#)
- discrete time, [993](#)
- Distribution
 - Binomial, [1081](#)
 - canonical, [1067](#)
 - canonical family, [1067](#)
 - Negative Binomial, [1082](#)
 - Poisson, [1087](#)

- distribution, [3](#), [831](#)
 - conditional, [845](#), [848](#), [960](#)
 - discrete uniform, [895](#)
 - uniform
 - finite set, [902](#)
- distribution operator, [430](#), [431](#)
- entropy, [397](#), [398](#), [422](#)
- equation
 - expectation updating, [957](#)
 - mass-balancing, [931](#)
 - updating, [986](#)
- equivalence classes, [684](#)
- erosion, [186](#)
- essential certainty, [382](#)
- event, [165](#), [333](#), [333](#), [337](#), [360](#), [941](#)
 - complementary, [334](#), [335](#)
 - complements, [432](#)
 - occur, [165](#)
 - occurs, [337](#)
- exchangeable, [979](#)
- expectation, [35](#), [44](#), [71](#), [74](#), [80](#), [330](#), [393](#), [393](#), [413](#)
 - additivity, [977](#)
 - additivity rule, [905](#)
 - operator, [146](#)
- Expectation Rules, [919](#), [921](#), [922](#)
- exponential, [771](#)
- fact, [952](#)
- factorial power
 - falling, [597](#), [805](#), [811](#)
 - rising, [815](#)
- factory, [279](#)
 - FRP, [161](#), [182](#), [234](#)
 - kind, [174](#), [235](#), [242](#), [461](#), [473](#)
 - statistic, [92](#), [95](#), [96](#), [98](#), [114](#), [165](#), [187](#), [192](#), [409](#), [510](#)
- factory trick, [101](#), [112](#), [188](#)
- fiber, [643](#)
- field, [731](#), [770](#)
- fixed point, [472](#), [644](#)
- forest, [177](#)
- fork, [86](#), [634](#), [1054](#)
- formal power series, [592](#)
- frplib, [4](#), [12](#)
 - combinators, [6](#)
 - factory, [6](#)
- FRP, [14](#)
 - activated, [9](#)
 - clone, [225](#), [228](#), [229](#)
 - compatible, [116](#)
 - components, [158](#), [158](#), [223](#), [256](#)
 - scalar, [224](#), [256](#)
 - conditional, [76](#)
 - demo, [38](#)
 - dimension, [11](#)
 - Element, [48](#)
 - factory, [161](#), [182](#), [234](#)
 - fresh, [9](#)
 - marginal, [158](#)
 - numeric, [10](#)
 - observed, [9](#)
 - possible values, [14](#)
 - scalar, [11](#), [16](#)
 - scalar component, [133](#)
 - transformed, [115](#), [126](#)
 - value, [9](#)
- function, [563](#)
 - affine, [562](#)
 - analytic, [668](#)
 - anonymous, [95](#), [114](#), [170](#), [605](#)

- argument, 565
 - bijection, 632, 654, 660, 899
 - bounded, 667
 - codomain, 563
 - component, 673
 - composition, 621
 - concave, 669
 - strict, 669
 - constant, 562
 - continuous, 657, 668
 - continuously differentiable, 668
 - convex, 669
 - strict, 669
 - differentiable, 668, 668
 - domain, 563
 - evaluating, 565
 - graph, 564
 - higher order, 587
 - idempotent, 643, 644
 - identity, 562
 - image, 581, 691
 - inclusion, 563
 - increasing, 666
 - indicator, 19, 95, 114, 610
 - infinitely-differentiable, 668
 - injection, 654, 656, 900
 - inverse, 638
 - inverse image, 581
 - linear, 749
 - monotone, 666, 779, 788
 - multiple parameters, 565
 - non-decreasing, 666
 - non-increasing, 666
 - parameter, 536, 565
 - partial, 687
 - piecewise constant, 616, 844
 - polynomial, 562
 - post-inverse, 640
 - pre-inverse, 640
 - range, 564
 - restriction, 583
 - surjection, 654, 658, 900
 - type, 564
- gradient, 776
 - graph, 161, 456, 574
 - coloring, 650
 - complete, 649
 - connected components, 685
 - cycle, 783
 - directed, 574, F.10.18, 576
 - path, 783
 - edge, 161, 456, 574
 - description function, 574
 - incident nodes, 574
 - loop, 574
 - homomorphism, 648
 - multigraph, 574
 - node, 161, 456, 574
 - simple, 456, 574
 - undirected, 456, 576
 - with loops, 574
 - without loops, 456, 574
 - group, 730
 - Hoeffding's inequality
 - one-sided, 1142
 - two-sided, 1143
 - hole, 85
 - holonym, 784
 - homomorphism, 648, 661, 698
 - automata, 648
 - graph, 648

- monoid, [716](#)
- preorder, [788](#)
- idempotent, [643](#), [644](#)
- iid, [981](#)
- increment, [37](#), [544](#)
- independence, [224](#), [369](#)
 - conditional, [984](#)
 - mutual, [977](#)
- independent, [217](#), [224](#), [244](#), [256](#), [967](#)
- independent and identically
 - distributed, [981](#)
- independent join, [43](#), [62](#), [256](#)
- indicator, [336](#), [611](#)
- indicators, [129](#), [151](#), [905](#)
- infimum, [791](#)
- interval, [544](#)
 - unit, [545](#)
- involution, [640](#)
- isomorphic, [699](#), [771](#)
- isomorphism, [478](#), [662](#), [698](#), [699](#), [716](#)
- Iverson braces, [95](#), [114](#), [129](#), [133](#), [336](#), [432](#), [614](#)
- Jacobian Method, [1033](#)
- join, [48](#), [52](#), [62](#), [76](#), [212](#), [213](#), [215](#), [222](#), [275](#), [281](#), [323](#), [790](#), [791](#)
 - clean, [281](#), [281](#), [282](#), [282](#), [301](#), [324](#), [336](#)
 - database, [691](#)
 - independent, [70](#), [130](#), [217](#), [222](#), [223](#), [225](#), [226](#), [227](#), [233](#), [256](#), [394](#)
 - clone construction, [228](#)
 - flat construction, [228](#)
 - independent power, [258](#)
 - multiplexer, [213](#), [222](#)
 - source, [212](#), [222](#), [272](#)
 - target, [212](#), [222](#), [272](#)
- join-marginal, [314](#)
- kernel, [831](#)
 - conditional, [845](#), [848](#)
 - simple, [887](#), [887](#)
- Kind, [14](#), [16](#)
 - canonical form, [18](#), [142](#), [202](#), [204](#), [209](#)
 - compact, [17](#)
 - compact form, [204](#)
 - compatible, [138](#)
 - completeness rule, [16](#)
 - conditional, [76](#), [265](#), [268](#), [377](#)
 - dimension, [16](#)
 - Distribution operator, [839](#)
 - equivalence, [201](#), [209](#)
 - equivalent, [39](#)
 - factory, [20](#), [23](#), [42](#), [174](#), [235](#), [242](#), [461](#), [473](#)
 - fully unfolded, [17](#)
 - join
 - independent, [240](#)
 - kernel, [19](#), [20](#), [20](#), [22](#), [142](#), [152](#), [159](#), [160](#), [246](#), [250](#), [269](#), [307](#), [309](#), [315](#), [359](#), [366](#), [396–398](#), [419](#), [432](#), [834](#), [834](#)
 - level rule, [16](#)
 - marginal, [158](#)
 - prefix rule, [16](#)
 - size, [16](#)
 - type, [16](#)
 - unfolded form, [204](#)
 - weight, [16](#)
 - weights, [14](#)
 - width, [16](#)

- language, 724
 - pattern, 483
 - regular, 724
- Law of Large Numbers, 1093, 1110
- Law of Rare Events, 1093, 1122
- likelihood, 59, 81, 82, 192
 - maximum, 189
 - principle, 189
- likelihood principle, 58, 69, 81, 82
- limit theorem, 1110
- linear combination, 583, 749
 - of functions, 582
- linearity, 392
- linearly dependent, 751
- linearly independent, 751
- magnitude, 666
- marginal, 314
- marginalization, 158
- mark, 1079
- Markov Chain, 998
- Markov chain, 1054
- Markov Property, 487
- Markov property, 455, 983, 993
- Markov's Inequality, 1132
 - generalized, 1133
- mass-balancing equation, 419, 425, 931
- matrix, 731, 732, 747, 754
 - determinant, 767
 - diagonal, 744, 768
 - direct sum, 738, 739
 - exponential, 771
 - identity, 738, 766
 - inverse, 767
 - invertible, 767
 - multiplication, 765
 - orthogonal, 774
 - positive definite, 767
 - positive semi-definite, 767
 - product, 734
 - square, 761
 - dimension, 761
 - symmetric, 741, 742, 761
 - symmetric square root, 767
 - trace, 768
 - transpose, 741, 742
- Mean Absolute Deviation, 396
- measure, 879, 896, 910, 911
- measure-preserving map, 911
- meet, 790, 791
- meronym, 784
- MGF, 1137
- Mill's Inequality, 1144
- model, 48, 54, 184
- Moment Generating Function, 1137
- monoid, 703
 - action, 714, 714
 - commutative, 703
 - dual, 711
 - identity, 703
- Monte Carlo, 1109
- multinomial coefficient, 803
- Multiplication Rule, 991, 1053
- multiset, 570, 571, 816
 - base set, 571
- norm, 667
- Normal Approximation, 1093, 1099, 1113, 1113
- Normal Distribution, 1102
- number
 - nothing, 10

- observation, 44, 70, 78, 174, 327, 332–334, **337**, 360, **944**
- observations, 332, 828
- operator
 - difference, 588
 - given, 79, 332, 338
 - join, 76
 - overloaded evaluation, 72, 73, 75, **116**, 125, **138**
 - overloading, 589
 - transform, 66, 72, 73, 75, **116**, 125, **138**
- order
 - lexicographic, **666**
 - partial, **792**
 - strict total, 779
 - total, 779, 792, **792**
- overload, 714
- parameters, 54
 - exogenous, 565, 572
- partial order, 791, 792
 - strict, 792, **792**
- partition
 - natural number, **819**
 - part, 658
- permutahedron, 697
- permutation, 221, **639**, 674, 800, 902
- pigeonhole principle, **812**, 821
- plate, 1057
- Poisson Approximation, 1093, 1099, **1121**
- Poisson Distribution, 1102
- polymorphism, 566
- polynomial, 726
 - coefficients, 726
 - degree, 726, 748
- poset, 791, **792**
- possible value, **872**
- predicate, **680**
- preorder, 779, 780, **780**
 - discrete, 780
 - lower set, 789
 - opposite, **781**
 - strict, 792, **792**
 - upper set, 789
- principle of inclusion-exclusion, **825**
- probability, 59, 168, **432**, 433, 941
- projection, 153, **153**, 684
- property, **680**
- Pythagoras's Theorem, 762
- quotient, 684
- random system, 39, 68
 - run, **39**
- random variable, 3
- random walk, **1088**
- reduction, 786
- region, 910
- regular expression, 724
- relation, **680**
 - antisymmetric, 682, **682**
 - arity, 680
 - asymmetric, 682, **682**
 - binary, 201, 679, 680
 - equivalence, 201, **684**, 782, 818
 - equivalence class, 202
 - heterogeneous, **680**
 - homogeneous, **680**
 - irreflexive, 682, **682**
 - join, **691**
 - order, 779
 - predicate, 680

- reflexive, 681, **682**, 780
- symmetric, 682, **682**
- ternary, 680
- total, 686
- transitive, 682, **682**, 780
- transitive closure, **685**, 780
- unary, 680
- univalent, 686
- relational database, 688
 - foreign key, 689
 - join, 691
 - primary key, 688
- ring, **731**
- risk-neutral price, 35, 57, 330, 383, 387, 390
- sampling variability, 38
- sampling variation, 27
- scalar, 11, 87, 745
 - multiplication, 746
- SDF, **1025**
- selector switch, 258
- semiring, 720, **720**
 - Arctic, 725
 - commutative, 720, **721**
 - Tropical, 725
- sequence, 562
- set, **542**
 - cardinality, 542
 - Cartesian product, 547, **548**, 634, 797
 - coproduct, 548
 - countable, 550
 - difference, 549
 - disjoint, 547, **547**
 - disjoint union, **548**, 634, 795
 - element, 542
 - empty, 542
 - increment, 37, 544
 - intersection, 546, **547**
 - interval, 544
 - ordered, 779
 - partially ordered, 791, **792**
 - partition, **658**
 - refinement, 659
 - power, 570, 661, 705, 721, 782
 - subset, 542, 543
 - strict, 543
 - uncountable, 550
 - union, 545, **546**
 - discriminated, 548
- singular value decomposition, 772
- size, 16, **16**
- skewness, **1118**
- space, 1079
- split kernel, **891**
- standard basis, **752**
 - dual, 752
- standard deviation, 1037, 1041
- state, 446
- statistic, 43, 56, 64, 69, 84, 87, 88, 114
 - built-in, 95, **95**, 114
 - codimension, **88**, 114
 - combinator, 66, 85, 93, 95, 98, 100, 108, 114, 177, 305, 329
 - condition, 165
 - custom, 95, **103**, 114
 - dimension, **88**, 114
 - factory, 92, 95, **96**, 98, 99, 103, 114, 165, 187, 188, 192, 409, 510
 - hole, 96
 - inline, 132, **133**

- condition, [942](#)
 - inlined form, [132](#)
 - projection, [97](#)
 - scalar, [88](#)
 - type, [88](#), [114](#)
 - valid inputs, [87](#), [88](#)
- statistic combinator, [85](#)
- steepest ascent, [776](#)
- Stirling cycle number, [824](#)
- Stirling subset numbers, [812](#)
- stochastic process, [993](#)
 - path segment, [993](#)
- subsets
 - all, *see* set, power
 - of specified cardinality, $\binom{A}{k}$, [576](#)
- substitution property, [412](#)
- supremum, [791](#)
- Survival Distribution Function, [1025](#)
- tail bounds, [1130](#)
- target dimension, [271](#)
- tensor product, [841](#)
- The Poisson Approximation to the Binomial Distribution, [1121](#)
- The Skewness-Corrected Normal Approximation, [1119](#)
- transform, [56](#)
- transformation
 - composition method, [1018](#), [1019](#)
- transition, [456](#)
- transition probability matrix, [997](#)
- transitions, [993](#)
- transpose, [759](#)
- tree, [165](#)
- tuple, [547](#)
 - components, [547](#)
 - dimension, [10](#), [547](#)
 - empty, [672](#)
- type, [16](#), [271](#), [478](#), [482](#), [542](#)
 - function, [478](#)
 - primitive, [478](#)
 - unit, [478](#), [565](#)
- types, [478](#)
- uncertainty, [382](#)
- uncountably infinite, [865](#)
- underflow, [716](#)
- updating equation, [986](#)
- value, [10](#), [14](#), [16](#), [87](#), [87](#), [88](#), [114](#), [482](#)
- values, [88](#)
- variance, [397](#), [398](#), [402](#), [426](#), [431](#), [838](#), [1037](#), [1039](#)
 - sample, [713](#)
 - shortcut, [402](#)
- vector, [672](#)
 - dot product, [761](#)
 - magnitude, [761](#)
 - orthogonal, [762](#)
 - orthonormal, [762](#)
 - projection, [764](#)
 - unit, [761](#)
 - zero, [745](#)
- vector space, [746](#), [746](#)
 - basis, [752](#)
 - orthonormal, [762](#)
 - dimension, [751](#)
 - direct sum, [753](#)
 - dual basis, [752](#)
 - finite dimensional, [751](#)
 - subspace, [753](#)
- weights
 - canonical, [32](#), [34](#)
- width, [16](#), [16](#)

yield operation, [73](#)