PyMORGAN

Multidimensional Optical spectroscopy Graphical Analysis iNterface
Data processing and user manual

Dr. Ricardo J. Fernández-Terán

25th October 2026

Contents

1 Introduction **1**

2 Installation **1**
2.1 Quick install via PyPI (Recommended) . . . 1
2.2 Installation from Source . . . 2
2.3 Step 2 — Install uv . . . 2
2.4 Step 3 — Obtain the PyMORGAN source . . . 2
2.5 Step 4 — Create the virtual environment and install . . . 3
2.6 Step 5 — Verify the installation . . . 3
2.7 Dependencies . . . 3
2.8 Keeping packages up to date . . . 3
2.9 Console scripts . . . 3
2.10 GUI layout and compiled UI . . . 4
2.11 Activating the environment (optional) . . . 4
2.12 Cross-Platform Standalone Installers (PyMORGAN Suite) . . . 5
2.13 Software Updates (Releases & Bleeding Edge) . . . 5

3 One-dimensional data **6**
3.1 Pipeline overview . . . 6
3.2 Data model . . . 6
3.3 Loading and the loader registry . . . 6
3.4 Background correction . . . 8
3.5 Solvent subtraction . . . 8
3.6 Shockwave subtraction . . . 8
3.7 Chirp (dispersion) correction . . . 9
3.8 Plotting . . . 10
3.9 Composing figures and displaying . . . 10
3.10 Kinetic analysis . . . 11
3.11 Settings and aesthetics . . . 11
3.12 Worked example . . . 12
3.13 Graphical interface — 1-D dialogs . . . 13
3.13.1 Chirp correction dialog . . . 13
3.13.2 Solvent subtraction dialog . . . 13
3.13.3 Shockwave subtraction dialog . . . 13
3.13.4 Dataset Stitching: Wavelength and Timescale . . . 13
3.13.5 Singular Value Decomposition (SVD) . . . 14
3.13.6 Polarization Anisotropy and Magic Angle . . . 15

3.13.7 Time Derivative of Spectra ($\mathrm{d}(\Delta A)/\mathrm{d}t$) . . . 16

4 Two-dimensional data (2D-IR, 2D-ES, 2D-EV, and 2D-VE) 16
4.1 Data model . . . 17
4.2 Loading and the loader registry . . . 17
4.3 Interferogram processing, phasing, and Fourier transformation . . . 18
4.3.1 Baseline correction and preprocessing . . . 18
4.3.2 Rotating-frame parameters . . . 18
4.3.3 Interferogram time-zero ($t_1 = 0$, binzero) via the Fourier Shift Theorem . . . 18
4.3.4 Apodisation (window functions) . . . 19
4.3.5 Zero-padding and Fourier Transformation . . . 19
4.3.6 Phasing models . . . 20
4.3.7 Pump correction and background subtraction . . . 20
4.4 Single-scan contour array exploration and scan recombination . . . 20
4.5 Background correction . . . 21
4.6 Plotting . . . 21
4.6.1 Axis units and the 10^3 rule . . . 21
4.7 Analysis . . . 22
4.8 Spectral diffusion . . . 22
4.8.1 Sub-pixel refinement . . . 22
4.9 Worked example . . . 23
4.10 Graphical interface . . . 23
4.10.1 Phasing/FT sub-tab . . . 23
4.10.2 Other plt. sub-tab . . . 23
4.10.3 Single Scans panel & interactive scan contour array . . . 24
4.10.4 Movie / GIF export dialog . . . 24
4.10.5 Direct Kubo Lineshape & Generalized FFCF Fitting . . . 24
4.10.6 Interactive 2D Gaussian Peak Fitting & Cross-Peak Generator . . . 25
4.10.7 2D ROI Volume Integration & Dynamics . . . 25
4.10.8 2D Spectrum Subtraction . . . 25

5 Steady-state spectra 25
5.1 Data model . . . 26
5.1.1 `SpectrumKind` . . . 26
5.1.2 `Spectrum` . . . 26
5.1.3 `SpectrumSeries` . . . 26
5.2 Loading and the loader registry . . . 26
5.3 Transformations . . . 27
5.4 Integration with the 1-D pipeline . . . 27
5.5 Plotting . . . 27

6 Settings 28
6.1 The configuration file . . . 28
6.2 Editing settings in the GUI . . . 28
6.3 Common settings . . . 28
6.4 1-D settings . . . 28
6.5 2-D settings . . . 28
6.6 GUI and defaults settings . . . 28
6.7 Declared axis units . . . 29
6.8 Notes on individual settings . . . 29

7 Spectrometer & Pulse Shaper Wavelength Calibration **30**
7.1 Pipeline Overview . . . 30
7.2 Data Model & Data Structures . . . 31
7.3 Mathematical Optical Dispersion Models . . . 32
7.3.1 Linear Grating Dispersion Model . . . 33
7.3.2 Polynomial Grating Dispersion Model . . . 33
7.3.3 Prism Dispersion Model (SF10 Glass) . . . 33
7.4 Forward Model, Baseline Correction, and Optimisation Cost Function . . . 33
7.4.1 Forward Synthetic Spectrum Model . . . 33
7.4.2 Derivative Cost Function . . . 34
7.5 Axis Inversion Procedures . . . 34
7.6 1D Gaussian Pump Spectrum Profile Diagnostics . . . 34
7.7 Supported Hardware Setup Configurations . . . 34
7.8 Graphical User Interface (Calibration Tab) . . . 35
7.9 Programmatic Usage Example . . . 36

8 Extending PyMORGAN (optional) **36**
8.1 Editing the GUI . . . 37
8.1.1 Opening the layout . . . 37
8.1.2 Moving or resizing widgets . . . 37
8.1.3 Adding a widget . . . 37
8.1.4 Plot areas and icons . . . 37
8.1.5 Do not rename . . . 38
8.2 Adding a new data-format parser . . . 38
8.2.1 A 1-D reader . . . 38
8.2.2 2-D and steady-state readers . . . 39
8.2.3 The unit dictionary . . . 39

9 License **39**

1 Introduction

PyMORGAN provides a unified interface to load, process and plot ultrafast time-resolved spectroscopy data. The package is organised by dimensionality: the `oneD` module handles one-dimensional data (transient absorption / pump–probe, time-resolved IR, fluorescence up-conversion), while `twoD` targets two-dimensional data (2D-IR, 2D-ES, 2D-VE, 2D-EV), from raw interferograms through phasing and Fourier transformation to the spectral-diffusion and Kubo lineshape analyses. A common design runs through both: a single dataset object wraps the raw arrays and exposes processing and plotting methods, data formats are resolved through a loader registry, and all presentation choices are centralised in a settings object.

This manual documents both the data-processing model and the programming interface. Chapter 2 covers installation; chapter 3 covers the one-dimensional workflow; chapter 4 covers the two-dimensional workflow, including the spectral-diffusion analysis; chapter 5 covers steady-state absorption, emission and excitation spectra; chapter 6 documents every setting and the configuration file; chapter 7 documents the spectrometer and pulse shaper wavelength calibration engine; and chapter 8 (optional) explains how to extend the GUI and add a parser for a new data format.

2 Installation

PyMORGAN requires **Python 3.12 or later**. It can be installed directly from PyPI or installed from source using **uv**.

2.1 Quick install via PyPI (Recommended)

To install the latest official release into your active Python environment:

```
pip install pymorgan
# or with uv:
uv pip install pymorgan

# Register bundled fonts into matplotlib:
pymorgan-install-fonts
```

You can also launch the graphical interface directly without prior installation using `uvx`:

```
uvx --from pymorgan pymorgan-gui
```

2.2 Installation from Source

The steps below describe the full setup for development or source installation.

Step 1 — Install Python 3.12+.

Windows. Download and run the official installer from `https://www.python.org/downloads/`. Choose **Python 3.12** or later, and tick *Add Python to PATH* on the first installer page.

Alternatively, use the **Windows Store** or Python Standalone Builds (recommended for reproducibility). After the installer finishes, verify Python is on `PATH`:

```
python --version # should print Python 3.12.x or later
```

macOS.

```
brew install python@3.12 # Homebrew
# or download from https://www.python.org/downloads/
```

Linux (Debian/Ubuntu).

```
sudo apt update
sudo apt install python3.12 python3.12-venv python3.12-dev
```

2.3 Step 2 — Install uv

uv is a single self-contained binary that replaces `pip`, `venv`, `pip-tools` and `virtualenv` in one tool. Install it once, globally:

Windows (PowerShell).

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

macOS / Linux.

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Via `pip` (any OS).

```
pip install uv
```

After installing, open a new terminal window and confirm:

```
uv --version # e.g. uv 0.6.x
```

2.4 Step 3 — Obtain the PyMORGAN source

Clone the repository (or copy the folder to a location of your choice):

```
git clone https://github.com/RJFernandezTeran/PyMORGAN.git
cd PyMORGAN
```

If you do not use git, download and extract the zip archive from GitHub, then open a terminal inside the extracted folder.

2.5 Step 4 — Create the virtual environment and install

From the repository root:

```
uv venv # creates .venv/ with the system Python (>= 3.12)
uv pip install -e ".[dev]" # editable install + ruff + pytest
uv run pymorgan-install-fonts # copy bundled fonts into matplotlib
```

What each line does:

- `uv venv` — creates a `.venv/` virtual environment in the current directory. Pass `-python 3.12` to pin a specific interpreter if multiple versions are installed.
- `uv pip install -e ".[dev]"` — installs all core dependencies (Table 1), plus the development extras (`ruff` linter and `pytest` test runner), in *editable* mode. Editable mode means changes to the source files take effect immediately without reinstalling.
- `uv run pymorgan-install-fonts` — copies the bundled `TgTermes` and other fonts used by the plot style profiles into matplotlib's font cache. Run this once after a fresh install.

If you only need to run the GUI (no development work):

```
uv pip install -e . # core only (no ruff / pytest)
uv run pymorgan-install-fonts
```

2.6 Step 5 — Verify the installation

Confirm the package imports and reports its version:

```
uv run python -c "import pymorgan; print(pymorgan.__version__)"
```

Launch the graphical interface:

```
uv run pymorgan-gui
```

Run the full test suite (requires the `dev` extras):

```
uv run pytest # 112 tests, all should pass
```

Run the end-to-end smoke check:

```
uv run python scripts/run_checks.py
```

2.7 Dependencies

Table 1 lists the core runtime dependencies. All are installed automatically by `uv pip install -e .`; no manual `pip install` is needed.

Package	Minimum version	Purpose
`numpy`	2.5.2	Array operations
`scipy`	1.18.0	Signal processing and fitting
`matplotlib`	3.11.1	Plotting back-end
`pandas`	3.0.5	Tabular data and I/O
`h5py`	3.16.0	HDF5 file I/O (some loaders)
`mpl-axes-aligner`	1.3	Aligned multi-axis figures
`tomlkit`	0.15.1	Settings file read/write
`PyQt6`	6.11.0	Graphical interface framework
`PyQt6_sip`	13.12.0	Qt bindings interface
`defusedxml`	0.7.1	Hardened XML parsing for Qt Designer UI files

Table 1: Core runtime dependencies (minimum version floors as of August 2026). Exact versions are pinned in `uv.lock`.

Package	Minimum version	Purpose
`ruff`	0.16.2	Linter and formatter
`pytest`	9.1.1	Test runner
`pytest-qt`	4.5.0	Qt widget testing helpers

Table 2: Development extras (`uv pip install -e ".[dev]"`).

2.8 Keeping packages up to date

To upgrade all dependencies to the latest compatible versions:

```
uv lock --upgrade # resolve latest versions, update uv.lock
uv sync # install the resolved versions into .venv
```

2.9 Console scripts

Five console scripts are installed with the package:

Script	Purpose
`pymorgan-gui`	Launch the PyQt6 graphical interface
`pymorgan-install-fonts`	Install bundled fonts into matplotlib
`pymorgan-edit-gui`	Open the GUI layout in Qt Designer
`pymorgan-settings`	Standalone settings editor (no data loaded)
`pymorgan-setup-env`	1-click local Python environment setup wizard

Table 3: Console scripts installed with the package.

All scripts are run via `uv run <script>` (e.g. `uv run pymorgan-gui`), or directly if the environment is activated.

2.10 GUI layout and compiled UI

The GUI layout is defined in `src/pymorgan/gui/main_window.ui` (Qt Designer XML). For performance, PyMORGAN also ships a pre-compiled Python translation, `main_window_ui.py`, generated by `pyuic6`. Loading the compiled module is approximately $1.5\times$ faster than parsing the XML at runtime.

No manual action is needed when editing the `.ui` file. When `main_window.ui` is saved in Qt Designer and the GUI is next launched, PyMORGAN detects that the compiled module is stale (via file modification times) and regenerates it automatically using `pyuic6` before loading the window. If `pyuic6` is unavailable or fails, the raw XML is parsed as a transparent fallback.

To force a manual recompile (e.g. for CI):

```
uv run pyuic6 src/pymorgan/gui/main_window.ui \
            -o src/pymorgan/gui/main_window_ui.py
```

2.11 Activating the environment (optional)

All commands in this manual use the `uv run` prefix, which automatically targets the project's `.venv` without requiring activation. If you prefer an activated shell (e.g. for interactive work in a terminal):

```
# Linux / macOS
source .venv/bin/activate

# Windows (PowerShell)
.venv\Scripts\Activate.ps1

# Windows (Command Prompt)
.venv\Scripts\activate.bat
```

Once activated, commands can be run without the `uv run` prefix (e.g. `pymorgan-gui` instead of `uv run pymorgan-gui`).

2.12 Cross-Platform Standalone Installers (PyMORGAN Suite)

For lab workstations and end-users who prefer a self-contained installation without manually managing a Python environment, **PyMORGAN Suite** ships native standalone distributions for **Windows**, **macOS**, and **Linux**, built automatically on every release via PyInstaller:

- **Windows** (`x64`)**:**
  - **Setup Wizard** (`PyMORGAN_Suite_Setup_v*.exe`)**:** Compiled with Inno Setup; registers Windows Start Menu shortcuts, Desktop icons with official software emblems, and native file associations for 1D (`.pdat`) and 2D (`.p2dat`) datasets.
  - **Portable Archive** (`PyMORGAN_Suite_v*_Windows_x64_Portable.zip`)**:** Zero-install zip containing `PyMORGAN_Suite.exe`, `PyMORGAN.exe`, and `PyRATE-TA.exe`.
- **macOS** (`Apple Silicon / Intel`)**:**
  - **Disk Image** (`PyMORGAN_Suite_v*_macOS.dmg`) **& Application Bundle** (`PyMORGAN Suite.app`)**:** Deep code-signed macOS bundle containing `PyMORGAN_Suite`, `PyMORGAN`, and `PyRATE-TA` inside `Contents/MacOS` with native `pymorgan.icns` icons.
- **Linux** (`x86_64`)**:**
  - **Tarball & Desktop Installer** (`PyMORGAN_Suite_v*_Linux_x86_64.tar.gz`)**:** Contains standalone binaries (`bin/PyMORGAN_Suite`, `bin/PyMORGAN`, `bin/PyRATE-TA`), freedesktop `.desktop` launchers, and automated `install.sh` / `uninstall.sh` scripts.

- **1-Click Local Python Environment Setup** (`pymorgan-setup-env`)**:** Even when installed via the standalone Suite installer, users can open **Tools** → **Setup in Local Python Environment…** in the Suite Launcher (or click **Setup in Python…** in the **About** dialog) to automatically configure any local Conda, Homebrew, `uv`, or system Python environment so `import pymorgan` and `import pyrate_ta` work directly in scripts and Jupyter notebooks.

2.13 Software Updates (Releases & Bleeding Edge)

PyMORGAN includes an integrated software update delivery system supporting both official releases and bleeding-edge development builds:

Automatic background checks When the application launches, PyMORGAN performs a non-blocking GitHub API query to check for newer releases (throttled to at most once every 24 hours). When a new version is detected, a pill badge appears in the status bar (e.g. [PyMORGAN vX.Y.Z Available]).

Release vs. Bleeding-Edge channel Users can check for updates at any time via **Help** → **Check for Updates (Release / Bleeding Edge)…** or from the **About** dialog, choosing either the latest tagged GitHub Release or syncing directly with the bleeding-edge `main` branch.

In-app download and installation When an update is confirmed, PyMORGAN displays the release notes/commits and streams the platform-matched installer or package update with a real-time progress bar.

3 One-dimensional data

3.1 Pipeline overview

The one-dimensional branch is organised as a *load* → *process* → *plot* → *analyse* pipeline. Each stage is a module under `pymorgan.oneD`, and the `Dataset1D` object ties them together: it holds the raw arrays and exposes one set of methods per stage. The stage functions take the dataset as their first argument, so they remain reusable on their own and new routines integrate with the whole pipeline. Table 4 maps each stage to its module and entry points.

Stage	Module	Entry point(s)
load	`pymorgan.oneD.load`	`pm.load_1D`, `Dataset1D.from_file`
process	`pymorgan.oneD.process` / `chirp`	`Dataset1D.background_correct`, `apply_chirp_correction`
plot	`pymorgan.oneD.plot`	`Dataset1D.plot_*`
analyse	*(PyRATE-TA)*	*separate project*

Table 4: The 1-D pipeline stages and where they live.

3.2 Data model

A one-dimensional measurement is represented by a `Dataset1D` object, which bundles the arrays produced by a loader. The signal is stored as a three-dimensional array to generalise over multiple detectors,

$$Z \in \mathbb{R}^{N_\tau \times N_\lambda \times N_d}, \tag{1}$$

indexed by delay τ, probe coordinate λ (wavelength, wavenumber or energy) and detector d. The accompanying axes are the delay vector $\tau \in \mathbb{R}^{N_\tau}$ and the probe vector $\lambda \in \mathbb{R}^{N_\lambda}$. Table 5 lists the seven fields every loader returns.

The most-processed signal is exposed through the `Z` property, which returns the background-corrected array once a correction has been applied and falls back to the raw array otherwise.

Field	Symbol / shape	Meaning
`Zavg_R`	$N_\tau \times N_\lambda \times N_d$	averaged signal
`delays`	N_τ	delay axis
`probe`	N_λ	probe (spectral) axis
`Units`	dict	unit labels (see §6)
`Nscans`	scalar	number of scans (`nan` if unknown)
`Zss_R`	$N_\tau \times N_\lambda \times N_d \times N_s$	single-scan signal
`Zstdv`	$N_\tau \times N_\lambda$	standard deviation

Table 5: Fields returned by a 1-D loader and stored on `Dataset1D`.

3.3 Loading and the loader registry

Data formats are decoupled from the dataset through a *loader registry* (`pymorgan.oneD.load`). A loader is any callable that maps a file path to the seven fields of Table 5. Loaders are registered under one or more data-type names and resolved at load time:

```
import pymorgan as pm

data = pm.load_1D("scan.pdat", data_type="PDAT")
print(pm.available_loaders())
```

Format Token	Instrument / Origin	Description
`PDAT`	PyMORGAN standard	Comma-separated averaged data (`.pdat`) + optional noise (`.pdatn`)
`UniGE_fsTA`	UniGE fs-TA	Step-scan femtosecond TA (`.dat`) with C-engine vectorised parsing
`UniGE_nsTA`	UniGE ns-TA	Nanosecond TA (`.dat`) with delay, pixel, and TA signal columns
`UoS_IRpp`	Univ. of Sheffield	Transient IR (`.2D` and `.DT`), 96 or 192 pixels, single-scan detection
`MESS_TRIR`	MESS TRIR	State-modulated TRIR directories with inverse-variance scan averaging
`MESS_TRUVIS`	MESS TRUVIS	State-modulated transient UV-Vis directories
`HARPIA`	Light Conversion	Harpia ultrafast transient absorption files (`.dat`)
`Helios_TA`	Ultrafast Systems	Helios TA files (`.csv`) with multi-scan discovery and interpolation
`PyMESS_PP`	PyMESS HDF5	Broadband pump–probe HDF5 datasets (`.h5`), including slow-modulation states and
`PyMESS_Aniso`	PyMESS HDF5	Slow-modulation polarization anisotropy HDF5 datasets (`.h5`)
`PyMESS_t2scan`	PyMESS HDF5	2D-IR population waiting-time (t_2) scan HDF5 datasets (`.h5`)
`Exported_TXT`	Generic exports	Wavelength-vs-delay tabular text files (`.txt`)

Table 6: Supported one-dimensional data formats in PyMORGAN.

A new format is added by decorating a reader with `@pm.register_loader("MyInstrument")`:

```
@pm.register_loader("MyInstrument")
def read_my_instrument(path):
    ...
    return Zavg_R, delays, probe, Units, Nscans, Zss_R, Zstdv
```

High-performance file parsing. Large experimental files (such as UniGE fsTA and Helios TA scans spanning tens of thousands of lines) are parsed using high-throughput C-accelerated engines (`pandas.read_csv` with `engine="c"`) with automated fallbacks for non-standard whitespace or formatting anomalies. Probe wavelength calibration is automatically resolved from local directory structures (e.g. `CalibratedProbe.csv` or `pix2lam.mat`).

The PDAT format. A `.pdat` file is comma-separated. The first line carries the time and probe units separated by an asterisk (e.g. `ps*cm^{-1}`). The first data row holds the probe axis (with a leading 0), and

each subsequent row begins with a delay value followed by the signal at every probe coordinate. PDAT files store averaged data already in m OD, so no rescaling is applied on read. Per-point standard deviations are optionally stored in a sibling `.pdatn` file with an identical layout.

PyMESS HDF5 slow-modulation and multi-signal layouts. The PyMESS HDF5 readers (`PyMESS_PP`, `PyMESS_Aniso`, and `PyMESS_t2scan`) automatically classify and parse both standard single-state pump–probe datasets and multi-state acquisition schemas:

- **Slow modulation (`/data/slow_modulation/state_XX_<name>`):** State datasets are discovered while filtering out their `*_individual_scans` companions from state enumeration. Per-state single scans (`state_XX_<name>_individual_scans`) are loaded into `Zss_R` (shape $N_\tau \times N_\lambda \times N_{\text{states}} \times N_{\text{scans}}$) and used to compute a true empirical per-point standard deviation `Zstdv`. For polarization anisotropy datasets (`PyMESS_Aniso`), the isotropic Magic Angle ($\Delta A_{\text{iso}} = (\Delta A_{\parallel} + 2G\,\Delta A_{\perp})/3$) and Anisotropy ($r = (\Delta A_{\parallel} - G\,\Delta A_{\perp})/(\Delta A_{\parallel} + 2G\,\Delta A_{\perp})$) channels are appended automatically alongside the parallel and perpendicular states, including their derived per-scan tensors.
- **Multi-signal fast modulation (`/data/signals/signal_XX_*`):** Multi-beam or multi-phase chop/modulation schemes stored under `/data/signals` (such as pump-only, push-only, and pump+push difference channels) are exposed as named detector/modulation states in `Units["detectors"]`, with companion `*_noise` datasets mapped directly to `Zstdv`.
- **Shaper population-time scans (`PyMESS_t2scan`):** 2D-IR waiting-time (t_2) scans acquired at a fixed or zero coherence time ($t_1 = 0$) are supported whether raw interferograms are stored under `/data/interferograms_raw` or pre-demodulated state matrices are stored under `/data/slow_modulation`.

3.4 Background correction

The pre-time-zero background is estimated by averaging the signal over a delay window $[\tau_{\min}, \tau_{\max}]$ and subtracting it from every delay,

$$Z_C(\tau, \lambda, d) = Z(\tau, \lambda, d) - \left\langle Z(\tau', \lambda, d) \right\rangle_{\tau' \in [\tau_{\min}, \tau_{\max}]}, \tag{2}$$

where the average is taken with nan-aware statistics (`pymorgan.oneD.process`). The operation is applied through

```
data.background_correct(tmin=-20, tmax=-5) # apply
data.background_correct(tmin=-20, tmax=-5, do_correct=False) # passthrough
```

With `do_correct=False` the raw signal is copied unchanged into the corrected fields, so downstream code can always read `Z` uniformly. The method returns `self`, allowing calls to be chained.

3.5 Solvent subtraction

For measurements in solution, the pure-solvent response often has to be subtracted. The module exposes two entry points.

Manual subtraction. A scaled, time-shifted solvent trace is removed from the sample signal:

```
data.subtract_solvent(solvent, dt=0.0, scale=1.0)
```

`dt` shifts the solvent delay axis before interpolation (accounts for a timing offset between the two measurements) and `scale` multiplies the solvent signal before subtraction.

Automatic solvent fitting. `fit_solvent_auto` models the solvent contribution pixel by pixel using the solvent trace convolved with the Instrument Response Function (IRF) and removes it:

```
fit = data.fit_solvent_auto(solvent)
data.subtract_solvent_fit(fit)
```

Optional per-pixel fits for the time-zero offset and the amplitude scale can be enabled through the `per_pixel_dt` and `per_pixel_scale` settings (Section 6); the defaults are `false` and `true`, respectively.

3.6 Shockwave subtraction

In transient IR / UV-Vis measurements using flowing samples, acoustic shockwaves generated by the pump beam can appear as a time-dependent artefact that is correlated across *all* probe wavelengths. The artefact is removed by averaging a selected subset of detector pixels (or a probe-axis range) into a one-dimensional shockwave trace and subtracting it from every pixel:

```
from pymorgan.oneD.process import subtract_shockwave

Z_corrected = subtract_shockwave(Z, pixels=[0, 1, 2]) # pixel indices (0-based)
```

The GUI exposes this through the **Subtract Shockwave…** button in the 1-D tab, which opens the `ShockwaveSubtractionDialog`. The dialog lets the user choose between pixel-index and probe-axis-value selection, displays a live preview of the proposed shockwave trace, and applies the correction on confirmation (see §3.13).

3.7 Chirp (dispersion) correction

Ultrafast time-resolved spectroscopy measurements often suffer from Group Delay Dispersion (GDD), commonly referred to as "chirp". As the probe pulse passes through various dispersive optical elements (such as white-light generation crystals, lenses, sample cell windows, and the solvent itself), different spectral components are temporally stretched. Consequently, the effective time-zero $t_0(\lambda)$ varies across the probe wavelength range λ.

PyMORGAN provides tools to characterise and correct this dispersion. The time-zero profile $t_0(\lambda)$ is modelled using a generalised Cauchy dispersion equation:

$$t_0(\lambda) = c_0 + \sum_{n=1}^{N_C-1} c_n \left(\frac{\lambda_{\text{ref}}}{\lambda}\right)^{2n}, \tag{3}$$

where $c_0, \ldots, c_{N_C-1}$ are the dispersion coefficients, λ_{ref} is a reference wavelength (which defaults to the mean of the fitted range to make the coefficients dimensionless and of comparable magnitude), and N_C is the number of terms (usually 6).

To determine $t_0(\lambda)$ across the probe range, PyMORGAN implements three fit modes in the `pymorgan.oneD.chirp` module:

Automatic Fits the coherent artefact and early dynamics at every pixel independently. The model consists of a Gaussian Instrument Response Function (IRF), its optional first and second derivatives, a constant offset, and an optional erf-broadened exponential decay. PyMORGAN uses Variable Projection (VARPRO) to eliminate the linear amplitudes in closed form, optimizing only the nonlinear parameters $(t_0, \text{FWHM}, \tau_{\text{exp}})$ per pixel. A second pass uses smoothed parameters from the first pass as initial guesses to improve stability.

Step function A simplified alternative that jointly fits a Gaussian IRF (+ derivatives) and a Heaviside step convolved with the same Gaussian over a narrow early-time window. This is robust when the early-time exponential decay is not needed or is ill-conditioned.

Manual Fits the Cauchy dispersion equation directly to a set of user-picked (λ, t_0) points (e.g., clicked interactively on the contour map in the GUI), bypassing the per-pixel IRF fitting.

Once $t_0(\lambda)$ is obtained, the Cauchy dispersion equation is fit using bound-constrained, iteratively reweighted linear least-squares (IRLS with Tukey's bisquare weights) to handle outliers and reject bad pixel fits.

To apply the correction, the dataset's delay axes are interpolated onto a common chirp-free delay grid. Two boundary handling strategies are available:

- `crop`: Slices the delay axis to keep only the delay region where all wavelength channels are defined, discarding the out-of-bounds regions that become `NaN` after shifting.
- `extrapolate`: Fills out-of-bound values with the nearest valid boundary endpoint value, keeping the original delay range.

```
# Fit chirp automatically and apply the correction
fit = data.fit_chirp_automatic(deriv_mask=(True, True), use_exp=True)
data.plot_chirp_diagnostics(fit)
data.apply_chirp_correction(fit, boundary_handling="crop")
```

3.8 Plotting

Five plotters live in `pymorgan.oneD.plot`; each is exposed both as a `Dataset1D` method and as a stage-level function that takes the dataset as its first argument, so a new view only needs `(data, ax, ...)` and inherits the whole pipeline's context. Each returns the matplotlib axis (the kinetic view also returns the extracted traces). They are listed in Table 7.

Method / function	View
`plot_contour`	delay-vs-probe contour map
`plot_surface`	filled delay-vs-probe surface
`plot_spectra`	transient spectra at selected delays
`plot_kinetics`	kinetic traces at selected probe positions
`plot_species_spectra`	EAS/SAS from a global-analysis fit

Table 7: The 1-D plotters (`pymorgan.oneD.plot`).

```
data.plot_contour(Zscale=20) # method form
data.plot_spectra([0.5, 1, 5, 20, 100], doSmooth=1)
ax, t, Y = data.plot_kinetics([2132, 2218], plotStyle="-")

from pymorgan.oneD import plot # stage-function form
plot.plot_contour(data, label_style="/", cmap_ID="Rd/Wh/Bu v2")
```

The label style, colourmap, time-axis scale and ΔA unit convention are taken from the active settings (§6); any of them can be overridden per figure through the corresponding keyword argument.

For datasets with high dynamic range where weak features would otherwise be washed out by dominant bleach or ESA peaks, `plot_contour` provides non-linear arcsinh colour scaling via `Asinh=True` (also toggled in the GUI plot controls). Rather than modifying the underlying signal values, PyMORGAN applies an inverse hyperbolic sine colour normalization (`matplotlib.colors.AsinhNorm`) paired with non-linear sinh-spaced contour levels:

$$z_k = s \cdot \sinh(u_k), \quad u_k \in \left[-\operatorname{arcsinh}\left(\frac{|Z_{\min}|}{s}\right), \operatorname{arcsinh}\left(\frac{Z_{\max}}{s}\right) \right], \tag{4}$$

where $s = \frac{p}{100} \cdot \max(|Z_{\min}|, |Z_{\max}|)$ is the linear crossover threshold set by `asinh_pct` (p, default 5.0%). Signals within $\pm s$ behave linearly, while larger amplitudes are compressed logarithmically. Because the underlying data array is not mutated, coordinates and hover values retain their physical units (mOD), and colorbar ticks remain linear and clean (centred at zero and rounded to sensible numbers).

3.9 Composing figures and displaying

By default every plotter creates its own figure and axis. Passing an existing axis through `ax=` draws into it instead, which is how panels are combined into a layout or several datasets are overlaid on one axis. Repeated elements (axis labels and, for the maps, the colorbar) can be switched off so they are not drawn more than once; the toggles are listed in Table 8.

Keyword	Default	Plotters
`show_xlabel`	`True`	all
`show_ylabel`	`True`	all
`show_colorbar`	`True`	`plot_contour`, `plot_surface`
`show_colorbar_label`	`True`	`plot_contour`, `plot_surface`

Table 8: Per-call toggles for composing multi-panel figures.

```
import matplotlib.pyplot as plt
fig, (axL, axR) = plt.subplots(1, 2)

# two maps side by side; suppress the duplicated colorbar / y-axis label
dataA.plot_contour(ax=axL, show_colorbar=False)
dataB.plot_contour(ax=axR, show_ylabel=False)

# overlay two datasets' transient spectra on a single axis
dataA.plot_spectra([1, 5, 20], ax=axL)
dataB.plot_spectra([1, 5, 20], ax=axL, show_ylabel=False)

pm.show_plots() # wrapper around plt.show(); pm.show is an alias
```

When a plotter is given an existing axis it does not resize the parent figure, so dropping a plot into a layout leaves the surrounding panels untouched. `pm.show_plots(block=False)` returns immediately (e.g. inside a GUI event loop), and `pm.close_plots()` closes all open figures.

3.10 Kinetic analysis

Kinetic analysis is *not* part of PyMORGAN. Trace extraction and fitting — multi-exponential decays, and full global/target analysis with rate-matrix models (the source of the EAS/SAS consumed by `plot_species_spectra`) — live in the companion project **PyRATE-TA**, which shares this loader/dataset layer.

PyMORGAN keeps the two ends of that seam. Traces are obtained from `plot_kinetics`, whose third return value is the extracted data itself:

```
ax, t, Y = data.plot_kinetics([1950, 2000], plotStyle="-")
Y.shape # [Ndelays x Ncuts]
```

and fitted spectra are rendered by `plot_species_spectra`, which takes plain arrays (`Sfit`, `Taus`, `TauErr`, `isFixTau`, `modelType`) and is therefore independent of how the fit was produced.

The third piece of the seam is the noise. `Dataset1D.noise_array()` returns the per-point standard deviation — from `Zstdv` when a sibling `.pdatn` file was loaded, otherwise the spread across single scans, and `None` when the dataset carries neither:

```
sigma = data.noise_array() # [Ndelays x Npixels x Ndetectors], or None
```

A weighted fit uses it as `1/sigma`, which is what turns a fit statistic into a true reduced χ^2; without it the statistic is a plain sum of squared residuals. PyMORGAN owns the noise because it owns the loaders; PyRATE-TA never estimates it.

3.11 Settings and aesthetics

Presentation choices that are not intrinsic to the data are collected in a single `Settings` object. It is framework-agnostic: the GUI binds to the fields it describes, but the package never depends on the GUI. The fields are summarised in Table 9.

Field	Values	Effect
`profile`	regular / poster / inset	selects a `.mplstyle` file
`font_scale`	float	multiplies the profile font sizes
`label_style`	`()` `[]` / (empty)	axis-label delimiter
`delta_a_units`	`mOD` / `x1E3`	ΔA label convention
`cmap`	string	default contour colourmap
`time_axis_scale`	symlog / log / lin	default time-axis scale
`freq_label`	Omega_n / Omega_PP / …	2-D frequency-axis convention
`x_axis_unit`	native / nm / cm-1 / eV / THz	spectral X-axis display unit
`secondary_axis`	bool	complementary X axis (top)
`chirp_boundary_handling`	crop / extrapolate	boundary handling for chirp correction

Table 9: Fields of the `Settings` object.

The `label_style` field controls the delimiter so that an axis reads, for example, "Wavelength (nm)", "Wavelength [nm]" or "Wavelength / nm". The `delta_a_units` field switches the signal-axis label between "ΔA / m OD" and "$\Delta A \times 10^3$"; because $1\,\mathrm{mOD} \equiv \Delta A \times 10^3$, this is a labelling choice only and never rescales the stored data. The active `label_style` delimiter is honoured for the m OD label on every axis, including the contour colorbar (so "/" yields "ΔA / m OD"); the $\times 10^3$ form is always shown without a delimiter.

The style profiles wrap the existing hand-tuned `.mplstyle` files rather than replacing them; `font_scale` is applied as a light multiplicative overlay on top. Text is always rendered with matplotlib's mathtext engine; the LaTeX backend (`text.usetex`) is never enabled.

The `x_axis_unit` field converts the spectral (probe) X axis of the transient-spectra and contour plots between wavelength (nm), wavenumber (cm^{-1}), energy (eV) and frequency (THz); `native` keeps each dataset's stored unit. The conversion is physical (the axis values are recomputed, pivoting through the wavelength), not a relabelling. When `secondary_axis` is enabled a second X axis is drawn on top in the *complementary* unit: $\mathrm{cm}^{-1} \leftrightarrow$nm, with nm used as the complement for eV and THz. Both fields default to the active settings and can be overridden per call, e.g. `data.plot_spectra([1, 5, 20], x_axis_unit="nm", secondary_axis=True)`. The earlier `dualScale` keyword is retained as a deprecated alias (a truthy value maps to `secondary_axis=True`). Whenever a wavenumber axis (main or secondary) exceeds $5000\,\mathrm{cm}^{-1}$ its tick labels are divided by 1000 and the axis is relabelled "Wavenumber ($\times 10^3$ cm^{-1})", so UV/Vis wavenumbers stay legible; IR ranges are left unscaled. The axis data, stored limits and GUI spin boxes remain in true cm^{-1} – only the displayed tick labels and unit are scaled.

The `chirp_boundary_handling` field controls how out-of-bounds delays are treated during the chirp correction interpolation step, selecting between `crop` and `extrapolate` (see §3.7).

Settings are read from and written to a TOML file, with comments preserved on write so the file remains hand-editable after the GUI saves it:

```
pm.load_settings("settings.toml") # make the file's settings active
```

```
pm.update_settings(profile="poster", label_style="/")
pm.apply_style() # apply the profile to matplotlib
pm.save_settings("settings.toml") # persist (keeps comments)
```

A module-level *active* settings object is read by the plotting methods. Temporary overrides can be scoped with a context manager:

```
with pm.use_settings(delta_a_units="mOD"):
    data.plot_spectra([1, 5, 20]) # this figure only
```

3.12 Worked example

A complete script reproducing the standard workflow is provided in `examples/pump_probe_1D.py`. In outline, the four stages read:

```
import pymorgan as pm

pm.load_settings("settings.toml")
pm.apply_style()

data = pm.load_1D("testData/testTRIR.pdat", data_type="PDAT") # load
data.background_correct(tmin=-20, tmax=-5) # process
data.plot_contour(Zscale=20, Yscale="lin") # plot
ax, t, Y = data.plot_kinetics([1980, 2030], plotStyle="-") # plot + extract
```

3.13 Graphical interface — 1-D dialogs

Several interactive dialogs supplement the main 1-D tab.

3.13.1 Chirp correction dialog

The **Chirp…** button opens `chirp_dialog.py`, which guides the user through chirp correction in three sub-panels:

1. **Mode selection** — choose Automatic, Step-function or Manual.
2. **Parameter input** — set the delay window, IRF FWHM, and (for Automatic mode) which derivative terms to include.
3. **Diagnostics** — the fitted $t_0(\lambda)$ curve overlaid on the contour map; the boundary-handling strategy (crop / extrapolate) is chosen here before the correction is applied.

A progress dialog (`chirp_progress_dialog.py`) reports per-pixel fit progress for the Automatic mode, which can be parallelised via the `parallel_fitting` setting (thread pool, process pool, or sequential).

3.13.2 Solvent subtraction dialog

The **Subtract Solvent…** button opens a dialog in `dialogs.py` that loads a pure-solvent scan, previews the subtraction, and applies `fit_solvent_auto` or the manual `subtract_solvent` path according to the chosen settings.

3.13.3 Shockwave subtraction dialog

The **Subtract Shockwave…** button opens `shockwave_dialog.py`. The user selects a set of pixels (by index or probe value) whose average forms the shockwave reference trace, inspects a live preview, and confirms. The correction calls `subtract_shockwave` and updates the dataset in place.

3.13.4 Dataset Stitching: Wavelength and Timescale

Ultrafast experiments often collect transient spectra across distinct spectral windows (such as UV/Vis and NIR spectrometers) or combine ultrafast optical delay lines with electronically synchronised sub-nanosecond to microsecond delays. PyMORGAN groups these operations under the **Spectra** $\rightarrow$ **Stitch Datasets** menu, providing two dedicated graphical interfaces.

Wavelength stitching (`Stitch Wavelength Windows...`). When measurements are conducted in adjacent or overlapping wavelength ranges (for instance, a UV/Vis window spanning 350–$700\,\mathrm{nm}$ and an NIR window spanning 650–$1400\,\mathrm{nm}$):

1. **Overlap detection:** The tool automatically discovers the shared probe interval $[\lambda_{\min}^{\mathrm{ovl}}, \lambda_{\max}^{\mathrm{ovl}}]$ between Dataset 1 and Dataset 2.

2. **Amplitude scaling:** To compensate for slight differences in probe intensity or detector sensitivity, an optimal least-squares scaling coefficient α is evaluated across the shared delay and wavelength grid:
$$\alpha = \frac{\sum_{\tau,\lambda} Z_1(\tau,\lambda)\, Z_2(\tau,\lambda)}{\sum_{\tau,\lambda} Z_2(\tau,\lambda)^2}. \tag{5}$$
The user can choose to scale Dataset 2 to Dataset 1, scale Dataset 1 to Dataset 2, or maintain native unscaled amplitudes.

3. **Smooth cosine taper blending:** Across the transition window, signals are seamlessly blended using a raised cosine taper weight:
$$w(\lambda) = \frac{1}{2}\left[1 + \cos\left(\pi \frac{\lambda - \lambda_{\min}^{\mathrm{ovl}}}{\lambda_{\max}^{\mathrm{ovl}} - \lambda_{\min}^{\mathrm{ovl}}}\right)\right], \tag{6}$$
$$Z_{\mathrm{stitched}}(\tau,\lambda) = w(\lambda)\, Z_1(\tau,\lambda) + \big(1 - w(\lambda)\big)\, \alpha Z_2(\tau,\lambda). \tag{7}$$
This prevents sharp steps or derivative artefacts in the stitched data.

4. **Live inspection and export:** The dialog provides an interactive 3-panel display (Dataset 1, Dataset 2, and Stitched preview) with kinetic and spectral cut comparisons before updating the main window or exporting to a new `.pdat` file.

Timescale stitching (`Stitch Timescales...`). When merging femtosecond/picosecond optical delay data with nanosecond/microsecond flash-photolysis or electronic delay measurements:

1. **Delay offset calibration:** Compensates for small timing differences $\Delta\tau_0$ between the independent delay mechanisms so that both axes share an identical time-zero reference.

2. **Overlap interpolation and cross-scaling:** Identifies the overlapping delay window (typically 1–$5\,\mathrm{ns}$), estimates the relative signal scaling factor via linear regression over the overlapping delays, and resamples the dataset onto a unified temporal grid.

3. The combined time axis preserves fine sub-picosecond temporal resolution around time zero while transitioning smoothly to logarithmic or extended linear steps at late nanosecond and microsecond delays.

3.13.5 Singular Value Decomposition (SVD)

Accessed via **Analysis → Singular Value Decomposition (SVD)...**, this tool provides model-free factorisation of the two-dimensional transient matrix $Z \in \mathbb{R}^{N_\tau \times N_\lambda}$:

$$Z = U\Sigma V^{\mathsf{T}} = \sum_{k=1}^{\min(N_\tau, N_\lambda)} \sigma_k \, \mathbf{u}_k \, \mathbf{v}_k^{\mathsf{T}}, \tag{8}$$

where:

- $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \dots)$ contains the singular values in descending order, representing the variance/energy carried by each component.
- The columns of U ($\mathbf{u}_k \in \mathbb{R}^{N_\tau}$) represent the orthonormal left singular vectors, corresponding to the time-dependent kinetic evolution of each component.
- The rows of V^{T} ($\mathbf{v}_k \in \mathbb{R}^{N_\lambda}$) represent the orthonormal right singular vectors, corresponding to the spectral basis shapes of each component.

Diagnostic scree plot. The dialog displays a scree plot showing both the individual singular values σ_k (log-scaled) and the cumulative variance explained $\sum_{i=1}^{k} \sigma_i^2 / \sum \sigma_j^2 \times 100\%$. The transition from rapidly falling singular values to a flat, noisy plateau indicates the number of genuine photophysical species present in the sample, distinguishing genuine chemical states from random detector noise.

Low-rank noise filtering and reconstruction. Users can select a rank cutoff K (e.g. $K = 3$) to reconstruct a low-rank, noise-filtered approximation of the transient surface:

$$Z_{\mathrm{filtered}} = \sum_{k=1}^{K} \sigma_k \, \mathbf{u}_k \, \mathbf{v}_k^{\mathsf{T}}. \tag{9}$$

The dialog provides real-time contour previews of the raw data, the low-rank reconstruction, and the residual matrix $Z - Z_{\mathrm{filtered}}$. The filtered dataset can be committed directly to the main PyMORGAN session or exported to disk as a clean `.pdat` file.

3.13.6 Polarization Anisotropy and Magic Angle

Time-resolved polarization anisotropy allows isolating transition-dipole orientational relaxation (rotational diffusion) from isotropic population decay. The unified polarization suite is accessed through the top-level **Anisotropy** menu (as well as **Spectra → Calculate polarization anisotropy...**). It seamlessly handles:

- **PyMESS slow-modulation datasets (`.h5`):** Polarization states (parallel and perpendicular) acquired with slow modulation are automatically recognised from metadata, labeled, and paired.
- **Multi-detector instruments:** Hardware setups recording parallel and perpendicular signals on distinct detector channels.
- **Separate files:** Two distinct single-channel measurements (one parallel, one perpendicular) paired directly via the interactive file browser.
- **Legacy MESS folder archives:** Backwards-compatible presets (UV/IR anisotropy and calculated/experimental magic angles) accessible via the legacy submenu.

G-factor correction. To account for differences in spectrometer throughput, grating efficiency, and detector sensitivity for parallel versus perpendicular polarizations, a correction factor G is applied:

$$G = \frac{\Delta A_{\parallel}(\tau_{\mathrm{iso}}, \lambda)}{\Delta A_{\perp}(\tau_{\mathrm{iso}}, \lambda)}, \tag{10}$$

which can either be specified as a calibrated constant, read directly from acquisition metadata (e.g. in PyMESS datasets), or determined by tail-matching at late delay times where rotational diffusion has fully randomised the orientation distribution ($r \to 0$).

Anisotropy and isotropic decay. The time-resolved anisotropy $r(\tau, \lambda)$, the isotropic (magic-angle equivalent) absorbance change $\Delta A_{\mathrm{iso}}(\tau, \lambda)$, and the polarization difference $\Delta I(\tau, \lambda)$ are computed according to:

$$r(\tau, \lambda) = \frac{\Delta A_{\parallel}(\tau, \lambda) - G\,\Delta A_{\perp}(\tau, \lambda)}{\Delta A_{\parallel}(\tau, \lambda) + 2G\,\Delta A_{\perp}(\tau, \lambda)}, \tag{11}$$

$$\Delta A_{\mathrm{iso}}(\tau, \lambda) = \frac{\Delta A_{\parallel}(\tau, \lambda) + 2G\,\Delta A_{\perp}(\tau, \lambda)}{3}, \tag{12}$$

$$\Delta I(\tau, \lambda) = \Delta A_{\parallel}(\tau, \lambda) - G\,\Delta A_{\perp}(\tau, \lambda). \tag{13}$$

The calculated ΔA_{iso} is completely devoid of rotational reorientation dynamics, matching what would be measured directly at the magic angle ($54.7°$). In addition to the rich interactive dialog—featuring simultaneous intensity and $r(\tau)$ kinetic traces, denominator noise thresholding, and probe wavelength slicing—the **Anisotropy** menu provides instant 1-click execution to replace the active workspace dataset with $r(\tau, \lambda)$, ΔA_{iso}, or the polarized components for immediate downstream kinetic analysis.

3.13.7 Time Derivative of Spectra ($\mathrm{d}(\Delta A)/\mathrm{d}t$)

Accessed via **Pre-Processing → Time Derivative of Spectra…** (`time_derivative_dialog.py`) or programmatically through `pymorgan.oneD.process.time_derivative`, this tool computes the numerical derivative $\partial \Delta A(\tau, \lambda)/\partial \tau$ along the delay axis to accentuate rapid kinetic inflection points and coherent oscillations:

1. **Early-delay cutoff:** Optionally trims delays $\tau < \tau_{\min}$ prior to differentiation so that the intense time-zero coherent artefact does not dominate the derivative scale.
2. **Time-domain interpolation:** Optionally resamples the signal onto a uniform or denser temporal grid (N_{interp} points) before differentiation.
3. **Pre-smoothing:** Supports Savitzky–Golay polynomial filtering or moving-average smoothing along the delay dimension, with a live interactive preview comparing raw and differentiated kinetic traces at selected probe wavelengths.

4 Two-dimensional data (2D-IR, 2D-ES, 2D-EV, and 2D-VE)

The two-dimensional branch follows the same *load → process → plot → analyse* pipeline as the one-dimensional one (chapter 3), built around a single `Dataset2D` object. It is designed for general two-dimensional optical correlation spectroscopy across all spectral regimes:

- **2D-IR** (Two-Dimensional Infrared Spectroscopy: vibrational pump ω_1 and vibrational probe ω_3 in the mid-IR).
- **2D-ES** (Two-Dimensional Electronic Spectroscopy: electronic pump ω_1 and electronic probe ω_3 in the UV/Vis/NIR).

- **2D-EV** (Two-Dimensional Electronic–Vibrational Spectroscopy: electronic excitation ω_1 in the UV/Vis coupled to vibrational detection ω_3 in the mid-IR).
- **2D-VE** (Two-Dimensional Vibrational–Electronic Spectroscopy: vibrational excitation ω_1 in the mid-IR coupled to electronic detection ω_3 in the UV/Vis/NIR).

Because `Dataset2D` stores the pump (ω_1) and probe (ω_3) axes independently, both homonuclear (2D-IR, 2D-ES) and cross-regime heteronuclear (2D-EV, 2D-VE) measurements are handled uniformly. Table 10 maps each stage to its module.

Stage	Module	Entry point(s)
load	`pymorgan.twoD.load`	`pm.load_2D`, `Dataset2D.from_file`
process	`pymorgan.twoD.process`	`Dataset2D.process`, `background_correct`, `recalc_average`
plot	`pymorgan.twoD.plot`	`Dataset2D.plot_map`, `plot_surface`, `plot_scan_maps`
analyse	`pymorgan.twoD.analyse` / `kubo_fit`	`Dataset2D.slice_at`, `diagonal`, `doCLS`, `doKuboFit`

Table 10: The 2-D pipeline stages and where they live.

4.1 Data model

A two-dimensional measurement is a stack of 2-D frequency–frequency maps, one per population time t_2. The signal is stored as a cube

$$Z \in \mathbb{R}^{N_{\text{pump}} \times N_{\text{probe}} \times N_{t_2}}, \tag{14}$$

indexed by the pump frequency ω_1, the probe frequency ω_3 and the population time t_2. When single-scan loading (`load_single_scans=True`) or slow-modulation state datasets are present, `Dataset2D` also stores the per-scan and per-state tensors. The fields held by `Dataset2D` are listed in Table 11.

Field	Shape	Meaning
`Z`	$N_{\text{pump}} \times N_{\text{probe}} \times N_{t_2}$	signal cube (m OD)
`pump`	N_{pump}	pump axis ω_1
`probe`	N_{probe}	probe axis ω_3
`delays`	N_{t_2}	population times t_2
`units`	dict	unit labels and metadata
`raw_signal_ss`	$N_{t_1/\text{pump}} \times N_{\text{probe}} \times N_{t_2} \times N_s$	per-scan raw signal tensor (optional)
`states_data`	dict	slow-modulation / polarization state cubes (optional)
`states_signal_ss`	dict	per-state single-scan tensors (optional)
`nscans`	int	total number of individual scans N_s
`active_scan_indices`	list[int]	0-based indices of scans included in the active average

Table 11: Fields stored on `Dataset2D`.

4.2 Loading and the loader registry

As in 1-D, formats are resolved through a registry (`pymorgan.twoD.load`). The registered readers are `P2DAT` (processed maps), `MESS_2DIR`, `PyMESS_2D` (alias `PyMESS_2DIR`), `UoS_2DIR`, `RAL_RAW` (raw interferograms, phased and transformed on load) and `RAL_Proc`; `pm.available_map_loaders()` lists what the running installation recognises.

```
import pymorgan as pm

data = pm.load_2D("scan.p2dat", data_type="P2DAT")
print(pm.available_map_loaders())
```

A new format is registered with `@pm.register_map_loader("Name")`; the reader must return `(Z, pump, probe, delays, units, freq_units)`.

Format Token	Instrument / Origin	Description
`P2DAT`	PyMORGAN standard	Comma-separated processed signal cube (`.p2dat`)
`MESS_2DIR`	MESS 2D-IR	Multi-file directory with raw/signal states, count, probe, and ref files
`UoS_2DIR`	Univ. of Sheffield	Sheffield 2D-IR acquisition files
`RAL_RAW`	STFC / Central Laser Facility	Raw Ultra 2D-IR interferograms
`RAL_Proc`	STFC / Central Laser Facility	Processed Ultra 2D-IR cubes
`PyMESS_2D`	PyMESS HDF5	Unified PyMESS 2D HDF5 datasets (`.h5`, alias `PyMESS_2DIR`)

Table 12: Supported two-dimensional data formats in PyMORGAN.

The P2DAT format. The first row is `0, 0,` $t_2^{(1)}$`,` $t_2^{(2)}$`, ...` (the population times); each subsequent row is `pump, probe, S(`$t_2^{(1)}$`), ...`. The flat pump/probe ordering is reshaped with zero-copy Fortran order directly into the `Z[pump, probe,` t_2`]` cube. Data is in m OD.

The MESS 2DIR and PyMESS 2D formats. A MESS 2DIR dataset is stored as a folder containing separate CSV files for each measurement component across all data states ($N_{\text{datastates}}$), spectral modulations (N_{spectra}), slow modulations (N_{slowmod}), interleaves, and population delay times (t_2). The reader automatically detects wobbler and chopper modulation modes, performs interferogram normalization (I/counts), calculates optical density $\Delta\text{OD} = -1000\log_{10}(I_{\text{probe}}/I_{\text{ref}})$, and reads the multi-file structure using parallelised, C-accelerated parsers.

For **PyMESS 2D HDF5 (`.h5`)** datasets (`PyMESS_2D` / `PyMESS_2DIR`), the loader supports both time-domain (shaper and interferometer) and pre-processed frequency-domain cubes, including slow-modulation schemas where state datasets live under `/data/slow_modulation/state_XX_<name>` (with or without `/data/interferograms_raw`). Companion `state_XX_<name>_individual_scans` datasets are loaded into `Dataset2D.states_signal_ss` and `raw_signal_ss`. When parallel and perpendicular polarization states are present, PyMORGAN automatically derives the `Magic Angle` and `Anisotropy` state cubes — as well as their corresponding per-scan tensors — and preserves the user's active scan selection (`active_scan_indices`) across state switches via `set_active_state()`.

4.3 Interferogram processing, phasing, and Fourier transformation

For raw time-domain 2D-IR measurements (such as interferometer- or pulse-shaper-based acquisitions), the data must be baseline-corrected, windowed, Fourier-transformed along the excitation axis $t_1 \to \omega_1$, and phased to produce pure absorptive 2-D correlation maps. In PyMORGAN, this is handled through `Dataset2D.process()` or the *Phasing/FT* GUI sub-tab.

4.3.1 Baseline correction and preprocessing

Optical scatter and low-frequency baseline drift are removed before spectral transformation:

- **Pulse-shaper data:** Mean subtraction across the time-domain points removes DC offsets:

$$S_{\text{corr}}(t_1, \omega_3, t_2) = S(t_1, \omega_3, t_2) - \langle S(t_1, \omega_3, t_2)\rangle_{t_1}. \tag{15}$$

- **Interferometer data:** A running median filter with an odd kernel size of 101 points is applied along t_1 to both the reference interferogram and the transient signal, isolating the oscillatory fringe signal from any slow baseline curvature.

4.3.2 Rotating-frame parameters

In rotating-frame acquisitions (such as AOTF or pulse-shaper spectrometers), the experimental time step Δt_1 and rotating-frame carrier frequency ω_0 (in cm^{-1}) define the spectral frequency axis:

$$\omega_1(k) = k \cdot \Delta\nu + \omega_0, \qquad \Delta\nu = \frac{1}{N_{\mathrm{FT}} \Delta t_1 c_0}, \tag{16}$$

where $c_0 = 2.99792458 \times 10^{-5}$ cm/fs is the speed of light and N_{FT} is the total number of Fourier transform points. If not explicitly provided, ω_0 and Δt_1 are read automatically from accompanying metadata (`w0.csv`). For interferometer systems, $\omega_0 = 0$ and $\Delta t_1 = 2.11079$ fs (the HeNe laser interference fringe period).

4.3.3 Interferogram time-zero ($t_1 = 0$, binzero) via the Fourier Shift Theorem

In Fourier transform spectroscopy, determining the exact zero-path-difference position ($t_1 = 0$, or "binzero" b_0) is essential for obtaining pure absorptive spectra without phase distortion. Physically, placing the time origin at $t_1 = 0$ means that the phase spectrum $\phi(\omega) = \arg(\mathcal{F}\{I\})(\omega)$ has zero first derivative across the optical carrier frequency ω_0:

$$\left.\frac{\mathrm{d}\phi}{\mathrm{d}\omega}\right|_{\omega_0} = 0. \tag{17}$$

Rather than iteratively rolling the time-domain signal and computing hundreds of redundant Fourier transforms, PyMORGAN determines b_0 analytically using the **Discrete Fourier Shift Theorem**. Circularly shifting a discrete sequence of length N by b bins, $I'(t_1) = \mathrm{roll}(I, -b)$, corresponds to an exact linear phase ramp in the frequency domain:

$$\mathcal{F}\{\mathrm{roll}(I, -b)\}[k] = X[k] \cdot \exp\left(+i\frac{2\pi k}{N} b\right), \tag{18}$$

where $X[k] = \mathcal{F}\{I\}[k]$ is the base Fourier transform computed once at the start. Consequently, the unwrapped phase at frequency bin k for shift b is:

$$\phi_b[k] = \phi_0[k] + \left(\frac{2\pi k}{N}\right) b. \tag{19}$$

Evaluating the phase difference across a symmetric window $[k_{\max} - \Delta k/2, k_{\max} + \Delta k/2]$ around the carrier peak gives:

$$\Delta\phi(b) = \phi_b[k_2] - \phi_b[k_1] = \underbrace{(\phi_0[k_2] - \phi_0[k_1])}_{\Delta\phi_0} + \underbrace{\left(\frac{2\pi\Delta k}{N}\right)}_{m} \cdot b. \tag{20}$$

Because $\Delta\phi(b)$ is strictly linear in b, the exact binzero position b_0 where $\Delta\phi(b_0) \equiv 0 \pmod{2\pi}$ is solved directly in closed form:

$$b_0 = \mathrm{round}\left(\frac{2\pi m_{\mathrm{wrap}} - \Delta\phi_0}{m}\right), \tag{21}$$

where $m_{\mathrm{wrap}} = \mathrm{round}((\Delta\phi_0 + m \cdot b_{\max})/2\pi)$ maps to the nearest integer phase wrap at the centerburst peak $b_{\max} = \mathrm{argmax}(I)$. This closed-form solution yields exact parity with traditional multi-pass grid searches while executing in under 1 μs per delay.

4.3.4 Apodisation (window functions)

To eliminate truncation ripples (sinc sidelobes) arising from the finite t_1 acquisition window, PyMORGAN applies an apodisation window $w(q)$ centred at binzero b_0. For $v = |q - (b_0 + 1)| \leq M = N_{\mathrm{bins}} - b_0 - 1$, the supported window functions are:

- **Box:** $w(v) = 1.0$ (uniform Heaviside weighting).
- **Cosine (Cos):** $w(v) = \cos\left(\frac{\pi v}{2M}\right)$.
- **Cos^2:** $w(v) = \cos^2\left(\frac{\pi v}{2M}\right)$.
- **Cos^3:** $w(v) = \cos^3\left(\frac{\pi v}{2M}\right)$.
- **Hanning:** $w(v) = 0.5 + 0.5\cos\left(\frac{\pi v}{M}\right)$.
- **Hamming:** $w(v) = 0.54 + 0.46\cos\left(\frac{\pi v}{M}\right)$.

Points beyond M are set to zero. For one-sided acquisition modes, the symmetric window is multiplied by a Heaviside step $H(q - b_0)$ so that negative times are excluded from the single-sided Fourier transformation.

4.3.5 Zero-padding and Fourier Transformation

To interpolate the frequency grid and avoid spectral picket-fence distortion, zero-padding appends zeros (or the asymptotic baseline value) to extend the length from N_{bins} to N_{FT}. PyMORGAN supports integer multiplication factors ($1\times$, $2\times$, $4\times$) and automatic padding to the next power of two (2^k, which accelerates the Fast Fourier Transform). The complex FFT is then computed along t_1 for each probe pixel and population delay.

4.3.6 Phasing models

The frequency-dependent phase error $\phi(\omega_1)$ is modelled by extracting the unwrapped phase spectrum around the carrier peak over a window of $\pm N$ points and fitting a polynomial by iteratively reweighted least squares (IRLS):

$$\text{Constant:} \quad \phi(\omega_1) = c_0, \tag{22}$$

$$\text{Linear:} \quad \phi(\omega_1) = c_1\omega_1 + c_0, \tag{23}$$

$$\text{Quadratic:} \quad \phi(\omega_1) = c_2\omega_1^2 + c_1\omega_1 + c_0, \tag{24}$$

$$\text{Cubic:} \quad \phi(\omega_1) = c_3\omega_1^3 + c_2\omega_1^2 + c_1\omega_1 + c_0. \tag{25}$$

The phased, purely absorptive transient signal is obtained by multiplying by the complex conjugate phase factor:

$$S_{\text{phased}}(\omega_1, \omega_3, t_2) = \text{Re}\left(S_{\text{raw}}(\omega_1, \omega_3, t_2) \cdot e^{-i\phi(\omega_1)}\right). \tag{26}$$

4.3.7 Pump correction and background subtraction

- **Pump correction:** When enabled, the signal at each excitation frequency ω_1 is normalized by the real excitation intensity profile $|I_{\text{pump}}(\omega_1)|$, compensating for laser pulse spectral shape:

$$S_{\text{norm}}(\omega_1, \omega_3, t_2) = \frac{S_{\text{phased}}(\omega_1, \omega_3, t_2)}{|I_{\text{pump}}(\omega_1)| / \max|I_{\text{pump}}|}. \tag{27}$$

- **Scatter subtraction:** A selected population delay map t_2^{scat} (typically an early or pre-zero delay containing pump scattering without sample dynamics) can be subtracted from all subsequent maps.

4.4 Single-scan contour array exploration and scan recombination

When `load_single_scans` is enabled in the settings (§6), multi-scan 2-D datasets (`MESS_2DIR`, `PyMESS_2D`, `UoS_2DIR`, and `RAL_RAW`) populate `raw_signal_ss` (shape $N_{t_1/\text{pump}} \times N_{\text{probe}} \times N_{t_2} \times N_s$) and optional per-scan interferograms `raw_interferogram_ss`.

- **Slice-level per-scan map computation** (`compute_scan_maps_at_t2`)**:** Evaluates 2-D frequency-domain maps `(label, scan_indices, Z_map)` at a target t_2 delay for each individual scan or group of N consecutive binned scans (`bin_size`). For time-domain shaper or interferometer datasets, only the requested t_2 slice (plus the scatter-subtraction reference slice when `bkg_sub=True`) is Fourier-transformed and phased per scan/bin, avoiding redundant full-cube FFTs.
- **Multi-panel contour array** (`plot_scan_maps`)**:** Calls `pymorgan.twoD.plot.plot_scan_map_array` to render a responsive grid of 2-D contour maps across all individual or binned scans alongside an **Average** comparison panel on a shared colour scale.
- **Scan recombination** (`recalc_average`)**:** Recomputes the dataset average across any subset of 0-based scan indices, updating `raw_signal`, `raw_interferogram`, `Z_R`, and `active_scan_indices`.

```
pm.update_settings(load_single_scans=True)
d2 = pm.load_2D("scan_folder", data_type="MESS_2DIR")

# Plot a grid of 2D contour maps at t2 = 0.5 ps (binning 2 scans per panel)
fig, axes, panels = d2.plot_scan_maps(0.5, bin_size=2, include_average=True)

# Recompute the dataset average excluding outlier scan index 3 (0-based)
keep = [i for i in range(d2.nscans) if i != 3]
d2.recalc_average(keep).process(apodise_method="1", phase_method="Linear")
```

4.5 Background correction

The 2-D background is a chosen t_2 map subtracted from the others,

$$Z_C(\omega_1, \omega_3, t_2) = Z(\omega_1, \omega_3, t_2) - Z(\omega_1, \omega_3, t_2^{\text{ref}}), \tag{28}$$

with the reference map itself left unchanged so it can still be inspected:

```
data.background_correct(reference=-1) # by t2 index
data.background_correct(t2=50) # by nearest t2 value
data.background_correct(do_correct=False) # passthrough
```

4.6 Plotting

`plot_map` renders the map nearest a requested t_2 as a pump-vs-probe contour and returns a `Map2DAxes(ax, top, cbar)` named tuple. It is built *from* the supplied (or freshly created) axis: the colorbar and an optional top panel are appended with a divider, so the whole composite drops into a parent layout.

```
out = data.plot_map(0.5) # one t2 map; new figure by default
out = data.plot_map(0.5, ax=my_ax, show_colorbar=False) # into a layout
```

Two behaviours are specific to 2-D:

- **Diagonal.** The $\omega_1 = \omega_3$ line is drawn only where the pump and probe axes overlap (pump and probe can be set independently); when their ranges are disjoint no diagonal is shown.

- **Top spectrum.** A steady-state or pump–probe spectrum can be drawn in a panel above the map (sharing the pump axis) via `top_spectrum`, which accepts a `Spectrum`, an `{"X","Y"}` mapping, or an `(x, y)` pair:

```
ftir = pm.load_spectrum("sample_FTIR.csv", kind="absorption")
data.plot_map(0.5, top_spectrum=ftir, top_label="FTIR")
```

The same `show_xlabel` / `show_ylabel` / `show_colorbar` / `show_colorbar_label` toggles as in 1-D apply. The frequency-axis labelling convention is the `freq_label` setting (§6): `Omega_n` (ω_1/ω_3), `Omega_PP` ($\omega_{\mathrm{pump}}/\omega_{\mathrm{probe}}$), `Pump-Probe`, or `Omega_n/2pic`; it can be overridden per call.

4.6.1 Axis units and the 10^3 rule

The maps are stored in wavenumbers, but the axes can be displayed in any of the units of Table 13, selected with the `twoD_freq_unit` setting or the `freq_unit` argument of `plot_map` and `plot_surface`. Conversions pivot through the wavelength,

$$\lambda\,[\mathrm{nm}] = \frac{P_{\mathrm{in}}}{x_{\mathrm{in}}}, \qquad x_{\mathrm{out}} = \frac{P_{\mathrm{out}}}{\lambda\,[\mathrm{nm}]}, \tag{29}$$

with the constants P listed in the table; wavenumbers in reciprocal inches follow $\tilde{\nu}\,[\mathrm{in}^{-1}] = 2.54\,\tilde{\nu}\,[\mathrm{cm}^{-1}]$.

Token	Quantity	P (unit · nm)	Typical mid-IR value
`cm-1`	wavenumber	10^7	2000
`in-1`	wavenumber	2.54×10^7	5080
`nm`	wavelength	—	5000
`THz`	frequency	2.99792458×10^5	60
`eV`	energy	1239.841984	0.25

Table 13: Display units of the 2-D spectral axes.

An axis whose values exceed 5×10^3 in the display unit is additionally divided by 1000, and the label gains a 10^3 prefix: a visible-range map reads ω_1 (10^3 cm^{-1}) with ticks at $18 \ldots 22$ rather than five-digit numbers. Axis limits and the CLS/IvCLS/NLS overlays are converted together with the axes, so a region selected in wavenumbers stays on the same features after a unit change.

4.7 Analysis

The analyse stage (`pymorgan.twoD.analyse`) offers map and diagonal extraction — the pump = probe diagonal is defined over the overlap of the two axes — together with the spectral-diffusion metrics of §4.8 and the Kubo lineshape fitting of `pymorgan.twoD.kubo_fit`.

```
pump, probe, m = data.slice_at(0.5) # the 2-D map at this t2
freq, sig = data.diagonal(0.5) # signal along omega_1 = omega_3

cls = data.center_line_slope() # [t2, slope]
both = data.center_line_slope(both=True) # [t2, CLS, IvCLS]
nls = data.nodal_line_slope() # [t2, slope]
```

4.8 Spectral diffusion

The centre-line slope (CLS) is obtained by taking, for every pump frequency ω_1, the extremum of the slice along the probe axis and fitting a line through those points; the inverse CLS (IvCLS) swaps the roles of the two axes, and the nodal-line slope (NLS) tracks the zero crossing between the bleach and the excited-state absorption. The line is fitted robustly (Huber loss), so a few outlying slices do not tilt the result.

4.8.1 Sub-pixel refinement

The extremum of a slice is *not* taken at the sampled maximum: with probe pixels of a few cm^{-1} that would quantise the centre line and bias the slope. Instead the position is obtained in two stages:

1. the slice is interpolated onto a grid `sd_interpolation_factor` times finer with a cubic spline, and the extremum is located there, already resolving positions below the spectral resolution;
2. a local model — `sd_interpolation`: a quadratic, cubic or quartic polynomial, or a Gaussian or Lorentzian peak — is fitted by least squares to the *measured* points around that location, and its stationary point is the reported centre.

Fitting the measured samples (rather than applying a three-point formula to the interpolated curve) keeps the estimate continuous and averages the noise over several points. On a synthetic tilted trough sampled at $2.5\,\mathrm{cm}^{-1}$ the residual error is ≈ 0.01 pixel for the quadratic model and ≈ 0.003 pixel for the quartic one; with `sd_interpolation = "none"` the fine-grid position is returned unrefined, and its resolution is set by the interpolation factor alone.

Two further settings restrict what enters the fit: `sd_peak_range` (cm^{-1}) is the half-width of the search window around the coarse extremum, `sd_intensity_threshold` the minimum amplitude (relative to the strongest signal of the map) a slice must reach to contribute, and `sd_peak_type` selects the bleach minimum or the excited-state absorption maximum.

The decay of the extracted slopes against t_2 is plotted from the GUI with single- or bi-exponential fits. That figure always uses a *linear* t_2 axis: population times are a short, linearly sampled series, unlike the wide delay range of a 1-D experiment.

4.9 Worked example

A complete script is provided in `examples/two_d_2DIR.py`:

```
import pymorgan as pm

pm.load_settings("settings.toml")
pm.apply_style()

data = pm.load_2D("testData/test2DIR.p2dat", data_type="P2DAT") # load
data.background_correct(reference=-1) # process
data.plot_map(0.5, ShowLines=True) # plot
```

4.10 Graphical interface

The GUI (`pymorgan-gui`) exposes the full 2-D pipeline through the **2D** tab. The left column contains a hierarchical file browser, format selector, and a series of sub-tabs that drive processing and analysis. Directories shown in grey (60 % grey) are navigation placeholders (`[..] / [...]`) that contain no loadable data.

4.10.1 Phasing/FT sub-tab

Raw (interferometer) 2-D data are Fourier-transformed and phased through the *Phasing/FT* sub-tab before any spectral map is rendered. Table 14 summarises the available controls.

The **Phase coeffs.** field (read-only) displays the polynomial coefficients of the fitted phase for the currently selected delay, formatted as `%.3g`. It updates automatically when the delay changes or after a new load.

Control	Options	Default
Apodisation	Box, **Cos**, $\cos^2$, $\cos^3$, Hanning, Hamming, None	Cos
Phase fit	Constant, **Linear**, Quadratic, Cubic	Linear
Phase range	$\pm N$ frequency units around peak centre	10
Zero-pad	enabled / disabled, factor, next-2^k	disabled
Pump correction	on / off	off

Table 14: Phasing/FT controls and their defaults.

4.10.2 Other plt. sub-tab

The *Other plt.* sub-tab contains three diagnostic views selected by the TD / PH / CAL toggle buttons:

TD Time-domain interferogram and signal for the selected probe pixel.

PH Fourier transform amplitude (black), a Gaussian fit (green dashed), and — for interferometer data — the unwrapped phase (red dots) and the polynomial fit (blue line) on a twin y-axis. The x-axis is locked to $\pm 2 \times \mathrm{FWHM}$ of the Gaussian fit. The phase axis is re-centred so the fitted phase is zero at the spectral peak, wrapped to $[-\pi, \pi]$, and plotted with fixed limits $[-\pi, \pi]$.

CAL Calibrated probe axis (pixel vs. wavenumber / THz).

Clicking the **Pop** button opens an independent, resizable window that mirrors the same axes. The window is linked to the main GUI: it updates whenever the delay or plot mode changes. Closing the pop-out does not affect the embedded plot.

4.10.3 Single Scans panel & interactive scan contour array

When a multi-scan 2-D dataset is loaded with `load_single_scans` enabled, the **Single Scans** group box (`twoD_singleScan_box`) appears in the 2-D tab:

Bin Scans Sets the number of consecutive scans averaged into each panel ($1 \leq N_{\text{bin}} \leq \lfloor N_s/2 \rfloor$).

Plot Scan Contours Opens an interactive multi-panel figure displaying the 2-D contour map for each individual (or binned) scan at the currently selected t_2 delay alongside an **Average** reference panel. Clicking any scan panel toggles its `[Excluded]` status (highlighted in red), and clicking the embedded **Recalc. Avg. (Exclude Selected)** button recomputes the dataset average and refreshes the main 2-D workspace immediately.

Recalc. Avg. Opens a dialog to specify custom 1-based scan ranges (e.g. `1-4, 6-10`) to include in the dataset average.

4.10.4 Movie / GIF export dialog

The **Make Movie…** button opens `movie_dialog.py`, which exports an animated GIF, WebP, or APNG video of the 2-D contour map stepping through all population times t_2. Controls include:

Delay range Lower and upper t_2 limits to include in the animation.

Frame rate / duration Frames per second or per-frame hold time in ms.

Figure size Figure width and height in inches.

Colour scale (Z-scale) Fixed or per-frame normalised colour range.

Output format GIF, WebP, or APNG (via Pillow, with zero external `ffmpeg` dependency).

A progress bar reports frame rendering. The dialog uses the plot-controls panel settings (CLS/IvCLS overlays, quick-plot mode, etc.) that are active in the main window when the dialog is opened.

4.10.5 Direct Kubo Lineshape & Generalized FFCF Fitting

The **Kubo Fit…** dialog (`kubo_dialog.py`, backed by `pymorgan.twoD.kubo_fit`) performs full third-order response-function simulation and multi-core non-linear least-squares fitting of 2D spectra using generalized Frequency Fluctuation Correlation Functions (FFCFs):

$$C(t) = \Delta_0^2 + \sum_{m=1}^{M} \Delta_m^2 e^{-t/\tau_{c,m}}. \tag{30}$$

The corresponding line-broadening function $g(t)$ combines pure homogeneous dephasing ($\Delta\omega_{T_2}$, with $1/T_2^* = \pi c_0 \Delta\omega_{T_2}$), static inhomogeneous broadening (Δ_0), and an arbitrary number of Kubo exponential decay terms (Δ_m, correlation time $\tau_{c,m}$):

$$g(t) = \pi c_0 \Delta\omega_{T_2}\, t + \frac{1}{2}\sigma_0^2 t^2 + \sum_{m=1}^{M} \sigma_m^2 \tau_{c,m}^2 \left(e^{-t/\tau_{c,m}} + \frac{t}{\tau_{c,m}} - 1 \right), \qquad \sigma_i = \frac{2\pi c_0\, \Delta_i}{2\sqrt{2\ln 2}}. \tag{31}$$

Key capabilities of the engine and dialog include:

- **Shared or per-band independent FFCFs:** Multiple FFCF groups can be defined and assigned independently to the excitation (ω_1, `ffcf_id`) and detection (ω_3, `ffcf_id_w3`) coordinates of each oscillator band.
- **Diagonal and cross-peak bands with bath correlation:** Supports both diagonal oscillators ($\omega_1 = \omega_3$) and off-diagonal cross-peaks ($\omega_1 \neq \omega_3$) with parameter linking (`link_w1`, `link_w3`, `link_anharm`) and a tunable cross-peak bath correlation factor $\eta_{AB} \in [-1, 1]$.
- **Variable Projection (VarPro) & multi-core parallelisation:** Uses Trust-Region Reflective (`scipy.optimize.least_squares`) with analytical Variable Projection to solve the linear GSB and ESA pathway amplitudes ($A_{\text{GSB}}, A_{\text{ESA}}$) at each evaluation, alongside full-parameter TRF and `L-BFGS-B` solvers. Time-domain evaluation uses separable 1D outer products, batched 2D FFTs, separable 2D rectilinear interpolation, and multi-core parallel execution across t_2 delays (`ProcessPool` / `ThreadPool`) with parallel Jacobian column evaluation.
- **Joint 2D + 1st-order FTIR fitting & diagnostics:** Simultaneously fits 2D correlation maps across selected t_2 delays and the analytical 1st-order steady-state FTIR spectrum (weighted by `kubo_ftir_weight`), with live FFCF $C(t)$ and $g(t)$ diagnostic plots and JSON/TOML model preset import/export.

4.10.6 Interactive 2D Gaussian Peak Fitting & Cross-Peak Generator

The **2D Gaussian Fit…** dialog (`twoD_gaussian_dialog.py`, backed by `fit_2d_gaussian_map` and `fit_2d_gaussian_global`) fits a sum of 2D correlated elliptical Gaussian peaks to diagonal and off-diagonal features:

$$G(\omega_1, \omega_3) = A \exp\left(-\frac{1}{2(1-\rho^2)} \left[\frac{(\omega_1 - \omega_{1,0})^2}{\sigma_1^2} + \frac{(\omega_3 - \omega_{3,0})^2}{\sigma_3^2} - \frac{2\rho(\omega_1 - \omega_{1,0})(\omega_3 - \omega_{3,0})}{\sigma_1 \sigma_3} \right] \right). \tag{32}$$

Users can click directly on the 2D contour map to seed peaks, automatically generate linked cross-peak pairs from diagonal modes via the `CrossPeakGeneratorDialog`, link centers/anharmonicities across peaks, and fit either single t_2 slices or globally across the entire t_2 stack.

4.10.7 2D ROI Volume Integration & Dynamics

The **2D Integral Dynamics…** dialog (`twoD_integral_dialog.py`) allows interactive rectangular region-of-interest (ROI) selection on the (ω_1, ω_3) map using a draggable `RectangleSelector`. It integrates the 2D signal volume $I(t_2) = \iint_{\mathrm{ROI}} Z(\omega_1, \omega_3, t_2)\, \mathrm{d}\omega_1\, \mathrm{d}\omega_3$ across all population waiting times t_2, plots the resulting kinetic trace, and fits it with single- or multi-exponential decay models.

4.10.8 2D Spectrum Subtraction

The **Subtract 2D Spectra…** dialog (`twoD_subtraction_dialog.py`) loads a reference 2D dataset (such as pure solvent or buffer), interpolates it onto the sample dataset's $(\omega_1, \omega_3, t_2)$ grid, and performs scaled subtraction $Z_{\mathrm{corr}} = Z_{\mathrm{sample}} - k\, Z_{\mathrm{ref}}$ with a live three-panel preview (Sample, Scaled Reference, and Difference Map).

5 Steady-state spectra

The `pymorgan.steadyState` module provides lightweight wrappers for absorption, emission and excitation spectra. Its design mirrors the one-dimensional branch (chapter 3): data are held in a plain object (`Spectrum`), file formats are resolved through a loader registry, and presentation is driven by the shared `Settings`.

5.1 Data model

5.1.1 `SpectrumKind`

`SpectrumKind` is a string enum that labels the physical quantity measured:

```
class SpectrumKind(StrEnum):
    ABSORPTION = "absorption"
    EMISSION = "emission"
    EXCITATION = "excitation"
```

5.1.2 `Spectrum`

A single steady-state measurement. The spectral axis `x` and signal `y` are stored as `numpy` arrays; the remaining fields are metadata:

Attribute	Type	Meaning
`x`	`ndarray`	Spectral axis (values in `x_units`)
`y`	`ndarray`	Signal (absorbance or intensity)
`kind`	`SpectrumKind`	Measurement type
`x_units`	`str`	Spectral-axis unit token (`"nm"`, `"cm-1"`, `"eV"`)
`label`	`str | None`	Legend label
`source`	`str | None`	File path (set automatically on load)

Table 15: Attributes of a `Spectrum` object.

5.1.3 `SpectrumSeries`

An ordered list of `Spectrum` objects for overlay plots. It supports iteration, indexing and the standard sequence protocol. Construction from multiple files returns a `SpectrumSeries` directly.

5.2 Loading and the loader registry

The module exposes a loader registry analogous to the 1-D and 2-D registries. The default `"csv"` reader handles comma- and tab-delimited files with automatic delimiter detection. Custom readers are registered with the `@pm.register_spectrum_loader` decorator:

```
import pymorgan as pm

# List available loaders
print(pm.available_spectrum_loaders())

# Load a single spectrum
sp = pm.load_spectrum("sample.csv", kind="absorption")

# Specify units explicitly (overrides the reader's guess)
sp = pm.load_spectrum("sample.csv", kind="emission", x_units="nm", label="Sample A")

# Load directly via Dataset1D convenience method
sp2 = pm.Spectrum.from_file("sample.opus", kind="absorption", data_type="opus")
```

The currently registered loaders are `csv` (generic delimiter-detected text), `opus` (Bruker OPUS binary, via `load.py`), `uv_vis` (Perkin-Elmer / Shimadzu UV-Vis text export) and `fluorimeter` (Horiba FluorEssence `.fes` files).

A custom loader returns a three-element tuple:

```
@pm.register_spectrum_loader("MySpectrometer")
def read_my_spectrometer(path):
    ...
    return x, y, x_units # x_units may be None (falls back to "nm")
```

5.3 Transformations

Two per-`Spectrum` transformations return a new `Spectrum` (the original is unchanged):

`normalised()` Scales the signal so that $\max(|y|) = 1$.

`to_stimulated_emission()` Computes the stimulated-emission cross-section $F_{\text{stim}} \propto \bar{\nu}^4 \, F_{\text{spont}}$ (valid for emission spectra on a wavelength axis). Useful when overlaying on transient-absorption spectra.

`SpectrumSeries.normalised()` applies the same operation to every member and returns a new series.

5.4 Integration with the 1-D pipeline

A `Spectrum` can be passed directly to `Dataset1D.plot_spectra` as a steady-state overlay:

```
ftir = pm.load_spectrum("background_FTIR.csv", kind="absorption")
data.plot_spectra([0.5, 1, 5], ss_overlay=ftir)
```

`as_overlay_dict()` converts a `Spectrum` to the `{"X": x, "Y": y}` mapping that `plot_spectra` also accepts, for cases where you want to pass raw arrays.

5.5 Plotting

Both `Spectrum` and `SpectrumSeries` have a `plot()` method. The axis-label convention and delimiter follow the active settings (§6), and optional keyword arguments are forwarded to `matplotlib.Axes.plot`.

```python
import pymorgan as pm

pm.load_settings("settings.toml")
pm.apply_style()

# --- Single spectrum ---
sp = pm.load_spectrum("sample.csv", kind="absorption", label="Sample")
sp.plot()

# --- Normalised emission ---
em = pm.load_spectrum("emission.csv", kind="emission", label="Emission")
em.normalised().plot()

# --- Series: multiple spectra coloured along a rainbow colourmap ---
series = pm.SpectrumSeries.from_files(
    ["a.csv", "b.csv", "c.csv"],
    kind="absorption",
    labels=["10 uM", "50 uM", "100 uM"],
)
series.plot(cmap="viridis", normalise=True)

pm.show_plots()
```

The per-call keyword arguments are identical to those of the 1-D plotters: `ax=` to draw into an existing axis, `label_style=` to override the delimiter, and `normalise=` to scale the signal to unit peak.

6 Settings

Every presentation choice that is not intrinsic to the data lives in a single `Settings` object: the matplotlib style profile, label and unit conventions, colour scale, axis units and the parameters of the spectral diffusion analysis. One instance is active at a time; the plotting routines read it, and any call may override individual values.

```python
import pymorgan as pm

pm.load_settings("settings.toml") # read a file
pm.update_settings(cmap="Seismic", quick_plots=True)
s = pm.get_settings() # the active object
pm.save_settings("settings.toml") # write it back, comments preserved

with pm.use_settings(profile="poster"): # temporary override
    data.plot_contour()
```

Values are coerced to the declared type, so a string from a text field or a TOML file becomes the right enum, number, path or pair: `pm.update_settings( contour_figsize="9, 5")` and `pm.update_settings(x_axis_unit="nm")` both work. An unknown name raises `TypeError` rather than being silently stored.

6.1 The configuration file

`settings.toml` is read at start-up (the GUI looks in the working directory first, then next to the package) and written back by *Save permanently* in the settings dialog. The file is edited in place, so comments and ordering survive a round trip. Keys map one-to-one onto the tables below; a key that is absent keeps its default, and an unknown key is ignored.

6.2 Editing settings in the GUI

View → Aesthetics / Settings… opens a panel whose widgets are generated from the settings themselves and grouped in four tabs — Common, 1D, 2D, and GUI & Defaults — matching the tables that follow. Changes apply to the current session immediately (the embedded preview re-renders); *Save permanently* writes them to `settings.toml`.

6.3 Common settings

Shared by every plot: style profile, labels, colour scale and the units of the spectral axis.

Key	Label in the panel	Type	Default
`profile`	Style profile	choice: `regular`, `poster`, `inset`	`regular`
`font_scale`	Font scale	float	`1`
`label_style`	Axis-label style	choice: `()`, `[]`, `/`, (empty)	`()`
`delta_a_units`	ΔA units	choice: `mOD`, `x1E3`	`x1E3`
`cmap`	Colourmap	choice: `DkRd/Wh/DkBu`, `Rd/Wh/Bu v2`, `Seismic`, `Jet`, `vik`, `berlin`	`DkRd/Wh/DkBu`
`white_levels`	White/Zero levels	float	`0`
`asinh_pct`	arcsinh linear %	float	`5.0`
`x_axis_unit`	Spectral X-axis unit	choice: `native`, `nm`, `cm-1`, `eV`, `THz`	`native`
`secondary_axis`	Complementary X axis	bool	`false`
`round_labels`	Round trace labels	bool	`true`
`quick_plots`	Use quick plots (fast redraws)	bool	`false`
`ss_line_dashes`	Steady-state dash pattern (on, off)	text	`4, 3`

Table 16: Common settings.

6.4 1-D settings

Delay-axis scaling and the appearance of kinetic / transient-spectra traces.

6.5 2-D settings

Map appearance, axis units and the spectral-diffusion analysis.

6.6 GUI and defaults settings

Application preferences and the values pre-selected when the GUI starts.

6.7 Declared axis units

Both dataset classes report the units of their axes through `axis_units()`, which returns a `DatasetUnits` record holding one entry per axis: `x` and `y` for the plotted axes, `z` for the signal, and `t2` for the population time of a 2-D dataset (`None` in 1-D, where the delay is already a plotted axis). Each entry carries the measured quantity, the machine-readable unit token, the mathtext used on plots, and the factor the values are divided by for display.

```
u = data.axis_units() # 1-D: x = probe, y = delay, z = signal
u.x.unit # 'cm-1'
u.x.label() # 'Wavenumber (cm$^{-1}$)'
u.y.quantity # 'Delay'
```

Key	Label in the panel	Type	Default
time_axis_scale	Time-axis scale	choice: symlog, log, lin	symlog
time_axis_label	Time-axis labels	choice: power, decimal, mixed	power
kinetics_marker_size	Kinetics marker size	float	6
kinetics_line_width	Kinetics line width	float	1.5
legend_label_spacing	Legend entry spacing	float	0.5
prune_legend	Prune legend	bool	true
max_legend_entries	Max legend entries	float	15
ss_abs_color	Steady-state abs. colour	text	b
ss_em_color	Steady-state em. colour	text	r
ss_line_width	Steady-state line width	float	1.5
ss_line_alpha	Steady-state line alpha	float	0.15
ss_fill_alpha	Steady-state fill alpha	float	0.05
inherit_cut_limits	Cuts inherit plot limits	bool	true
load_single_scans	Load single scans (1D & 2D)	bool	false
chirp_boundary_handling	Chirp boundary handling	choice: crop, extrapolate	crop
parallel_fitting	Chirp parallel fitting	choice: threadpool, processpool, disabled	threadpool
solvent_per_pixel_dt	Per-pixel solvent t_0 offset	bool	false
solvent_per_pixel_scale	Per-pixel solvent amplitude scale	bool	true
error_shading	Error shading ($\pm$1 std)	bool	false
error_shading_alpha	Error shading alpha	float	0.2

Table 17: 1-D settings.

```
print(u) # a readable summary of all axes
u.as_dict() # plain nested dict, e.g. for export

v = d2.axis_units() # 2-D: x = pump, y = probe, z = signal, t2 = delay
v.t2.unit # 'ps'
```

By default the *stored* units are reported. With `display=True` the axes are described as they would be plotted: converted to `x_axis_unit` (1-D) or `twoD_freq_unit` (2-D), carrying the 10^3 factor where one is applied (§4.6.1), and with `x` and `y` swapped when `pump_axis` is `Vertical`. The `label()` method renders the axis label in any of the delimiter conventions of `label_style`, so a caller can annotate a figure of its own without duplicating the unit logic.

6.8 Notes on individual settings

`quick_plots` Draws contour maps as a single image (`pcolormesh`) and skips the contour lines, which makes dragging the colour-scale slider or stepping through t_2 far quicker. It overrides the *Filled* and *Show lines* options of the plot-controls panel, so turn it off for final figures.

`x_axis_unit`, `secondary_axis` Control the spectral axis of 1-D plots: the axis is converted to the chosen unit, and a complementary axis ($\mathrm{cm}^{-1} \leftrightarrow$nm) can be added on the opposite side.

`twoD_freq_unit` The same idea for the pump/probe axes of 2-D maps, including reciprocal inches, with the 10^3 rescaling described in §4.6.1.

`sd_*` The spectral-diffusion family: search window, amplitude threshold, peak type, and the interpolation factor and model used for the sub-pixel refinement (§4.8).

`parallel_fitting` Chooses how the per-pixel chirp fit is distributed: a thread pool, a process pool (true parallelism, best for large datasets), or sequentially on the calling thread.

`gui_theme` Any Qt style available on the machine; appending *Dark* (e.g. `Fusion Dark`) switches to the dark palette, which also recolours the embedded plots.

Key	Label in the panel	Type	Default
freq_label	2-D frequency labels	choice: Omega_n, Omega_PP, Pump-Probe, Omega_n/2pic	Omega_n
pump_axis	2-D pump axis orientation	choice: Horizontal, Vertical	Horizontal
twoD_freq_unit	2-D axis unit	choice: cm-1, in-1, nm, THz, eV	cm-1
t2_label_style	t_2 delay label format	choice: t2=, plain	t2=
cls_color	CLS colour	text	b
ivcls_color	IvCLS colour	text	r
nls_color	NLS colour	text	g
overlay_draw_style	Overlay draw style	choice: Points, Lines, Both	Points
overlay_linewidth	Overlay linewidth	float	1.5
sd_intensity_threshold	Intensity threshold (top x%)	float	0.1
sd_peak_range	Peak search range (cm^{-1})	float	10
sd_peak_type	Target peak type	choice: min, max	min
sd_interpolation	Peak interpolation	choice: quadratic, cubic, quartic, gaussian, lorentzian, none	quadratic
sd_interpolation_factor	Interpolation factor (axis upsampling)	int	4
kubo_ftir_weight	Kubo joint fit FTIR vs 2D weight	float	1

Table 18: 2-D settings.

7 Spectrometer & Pulse Shaper Wavelength Calibration

Wavelength and wavenumber axis calibration maps detector pixel indices $p \in \{1, \ldots, N_{\text{pix}}\}$ to physical spectral values (wavelength λ in nm or wavenumber $\tilde{\nu}$ in cm^{-1}). PyMORGAN provides a dedicated calibration engine in the pymorgan.cal module, supporting both *spectrometer calibration* (for probe beam spectrographs) and *pulse shaper calibration* (for optical pulse shaper pump beam masks).

7.1 Pipeline Overview

The calibration workflow matches an experimentally measured absorption spectrum $I_{\text{meas}}(p)$ against a high-resolution reference FTIR spectrum $A_{\text{ref}}(\lambda)$ (such as liquid 1,4-dioxane, liquid polystyrene, or atmospheric water vapour). Non-linear least-squares (NLS) optimisation fits optical dispersion parameters so that the synthetic spectrum evaluated through the dispersion function reproduces the reference absorption features.

The workflow proceeds in six stages:

1. **Load Reference Spectrum:** Load $A_{\text{ref}}(\lambda)$ or $A_{\text{ref}}(\tilde{\nu})$ with automatic delimiter and unit detection (load_reference_spectrum).

2. **Load Experimental Data:** Parse raw detector intensity or absorbance arrays for one of twelve hardware setups (load_experimental_spectrum).

3. **Pre-alignment (Template Scanning):** Perform 1D cross-correlation scanning to estimate initial wavelength shift λ_1 at pixel 1.

4. **NLS Parameter Fitting:** Optimise dispersion coefficients and additive polynomial baseline using scipy.optimise.least_squares with 1st or 2nd derivative filtering.

5. **Axis Inversion:** Invert pixel-to-wavelength mapping $p(\lambda) \to \lambda(p)$ analytically or numerically.

6. **Export & Integration:** Save calibrated axis arrays (CalibratedProbe.csv, CalibratedPump.csv, SingleMask.txt, MultipleMask.txt).

Key	Label in the panel	Type	Default
gui_theme	GUI theme	choice: Fusion, Fusion Dark	Fusion
splash_duration	Splash-screen duration (s, 0 disables)	float	2
default_datadir	Default data directory	text	(none)
default_oneD_datatype	Default 1D data type	choice: Helios_TA, MESS_TRIR, MESS_TRUVIS, PDAT, UniGE_fsTA, UniGE_nsTA, UoS_IRpp	PDAT
default_twoD_datatype	Default 2D data type	choice: MESS_2DIR, P2DAT, RAL_Proc, RAL_RAW, UoS_2DIR	P2DAT
default_calibration_datatype	Default calibration setup	text	UniGE TRIR (Intensit
styles_dir	Style directory (blank = bundled)	text	(none)
contour_figsize	Contour figure size (w, h in)	text	8, 6
kinetics_figsize	Kinetics figure size (w, h in)	text	7.5, 4.5
spectra_figsize	Spectra figure size (w, h in)	text	7.5, 4.375

Table 19: GUI and defaults settings.

7.2 Data Model & Data Structures

The pymorgan.cal package defines four primary immutable data structures:

ReferenceSpectrum: Container for reference spectra library items.

```
class ReferenceSpectrum(NamedTuple):
    name: str # Spectrum name (e.g. 'FTIR-Dioxane')
    spectral_axis: ndarray# Spectral grid (nm or cm-1)
    absorbance: ndarray # Absorbance array A(x)
    axis_unit: str # Unit hint ('nm' or 'cm-1')
```

ExperimentalData: Container for loaded experimental spectrograph measurements.

```
class ExperimentalData(NamedTuple):
    detector_data: list[ndarray] # Raw detector intensity/absorbance arrays
    gratings: list[float] # Grating groove density (grooves/mm)
    cwl: list[float] # Central wavelength setting (nm or cm-1)
    file_path: Path # File path on disk
    min_wl: float | None = None # Lower spectral bound from metadata
    max_wl: float | None = None # Upper spectral bound from metadata
    spectral_range: str | None = None # Detected regime ('Mid-IR', 'UV/Vis', 'NIR
        ')
```

CalibrationResult: Output container returned by fit_wavelength_axis.

```
class CalibrationResult(NamedTuple):
    wavelength_nm: ndarray # Calibrated wavelength axis (nm)
    wavenumber_cm1: ndarray # Calibrated wavenumber axis (cm-1)
    fit_params: ndarray # Optimised dispersion parameters vector p
    residuals: ndarray # Fit residual array
    ref_x_cut: ndarray # Reference spectral axis in active cut region
    ref_y_cut: ndarray # Reference absorbance in active cut region
    model_y_fit: ndarray # Modelled experimental absorbance fit
    baseline_fit: ndarray # Additive polynomial baseline curve
    exp_corr_y_fit: ndarray # Baseline-corrected experimental absorbance
    ref_y_corr: ndarray # Baseline-corrected reference spectrum
```

`SpectrumFitResult`: Container for 1D Gaussian pump beam profile diagnostics.

```
class SpectrumFitResult(NamedTuple):
    amplitude: float # Peak amplitude (Counts)
    w0: float # Central frequency w0 (cm-1 or nm)
    fwhm: float # Full-Width at Half-Maximum (FWHM)
    fit_curve: ndarray # High-density fitted Gaussian profile
    axis_fit: ndarray # High-density evaluation axis
```

`ProbeFitResult`: Container returned by `fit_probe_spectrum`, which wraps up multi-detector Gaussian diagnostics for the probe beam profile.

```
class ProbeFitResult(NamedTuple):
    fits: list[SpectrumFitResult] # Per-detector Gaussian fit results
    detector_data: list[ndarray] # Raw detector arrays that were fitted
```

7.3 Mathematical Optical Dispersion Models

PyMORGAN supports three physical models for mapping reference wavelength λ (in nm) to detector pixel index $p(\lambda) \in [1, N_{\text{pix}}]$:

7.3.1 Linear Grating Dispersion Model

For diffraction grating spectrographs in first-order paraxial approximation, the pixel position $p(\lambda)$ varies linearly with wavelength around the central wavelength λ_0:

$$p(\lambda) = (\lambda - \lambda_0) \cdot d + c_{\text{pix}} \tag{33}$$

where λ_0 is the central wavelength mapped to the central pixel $c_{\text{pix}} = (1 + N_{\text{pix}})/2$, and d is the linear dispersion in pixels/nm.

7.3.2 Polynomial Grating Dispersion Model

For wide-bandwidth spectrographs or non-paraxial grating setups (such as UniGE TRUVIS-II), higher-order geometric distortion is modelled via a 2nd- or 3rd-order polynomial expansion around central wavelength λ_0:

$$p(\lambda) = (\lambda - \lambda_0) \cdot d + 10^{-4}\, c_2\, (\lambda - \lambda_0)^2 + 10^{-7}\, c_3\, (\lambda - \lambda_0)^3 + c_{\text{pix}} \tag{34}$$

where c_2 and c_3 are quadratic and cubic dispersion curvature coefficients.

7.3.3 Prism Dispersion Model (SF10 Glass)

For prism-based spectrographs (e.g. UniGE NIR-TA using an SF10 Flint glass equilateral prism), dispersion is governed by Snell's law and the Sellmeier equation. The refractive index $n(\lambda)$ for SF10 glass is:

$$n(\lambda) = \sqrt{1 + \sum_{i=1}^{3} \frac{B_i\, \lambda_{\mu\text{m}}^2}{\lambda_{\mu\text{m}}^2 - C_i}} \tag{35}$$

with Sellmeier coefficients:

$$\begin{aligned} B_1 &= 1.62153902, \quad & B_2 &= 0.256287842, \quad & B_3 &= 1.64447552 \\ C_1 &= 0.0122241457, \quad & C_2 &= 0.0595736775, \quad & C_3 &= 147.468793 \end{aligned}$$

where $\lambda_{\mu m} = \lambda/1000$. The deflection output angle $\theta_{\text{out}}(\lambda)$ from a prism with apex angle $\alpha = 60.84^\circ$ at incidence angle θ_{in} is:

$$\theta_{\text{out}}(\lambda) = \arcsin\left[n(\lambda)\sin\left(\alpha - \arcsin\left(\frac{\sin\theta_{\text{in}}}{n(\lambda)}\right)\right)\right] \tag{36}$$

The pixel index $p(\lambda)$ is then determined by focal length f_{mm} and detector pixel shift x_{shift}:

$$p(\lambda) = f_{\text{mm}} \cdot \left(\frac{1000}{25}\right) \cdot \sin\left[\theta_{\text{out}}(\lambda) - \theta_{\text{out}}(\lambda_{\text{ref}})\right] + x_{\text{shift}} \tag{37}$$

7.4 Forward Model, Baseline Correction, and Optimisation Cost Function

7.4.1 Forward Synthetic Spectrum Model

Given measured experimental spectrum array $I_{\text{meas}}(p)$, the model synthesises an experimental absorbance curve $y_{\text{model}}(\lambda)$ matched to reference grid points λ:

$$y_{\text{model}}(\lambda) = I_{\text{meas}}\big[p(\lambda)\big] \cdot s + b_0 + 10^{-6}\, b_1\,(\lambda - \lambda_0) + 10^{-6}\, b_2\,(\lambda - \lambda_0)^2 \tag{38}$$

where s is a global scaling factor, and (b_0, b_1, b_2) parameterise a smooth background baseline polynomial.

7.4.2 Derivative Cost Function

To eliminate broad solvent/background baseline drift during fitting, optimisation is performed in derivative space using Savitzky–Golay numerical filters (window size W, polynomial order 2):

$$\mathbf{r}(\mathbf{p}) = \frac{d^k}{d\lambda^k} y_{\text{model}}(\lambda; \mathbf{p}) - \frac{d^k}{d\lambda^k} A_{\text{ref}}(\lambda) \quad (k \in \{1, 2\}) \tag{39}$$

The parameters $\mathbf{p}$ are optimised by bounded Non-Linear Least Squares:

$$\min_{\mathbf{p}} \frac{1}{2}\|\mathbf{r}(\mathbf{p})\|_2^2 \quad \text{subject to } \mathbf{lb} \le \mathbf{p} \le \mathbf{ub} \tag{40}$$

7.5 Axis Inversion Procedures

Once optimal parameters $\mathbf{p}^*$ are found, the calibrated wavelength axis $\lambda(p)$ for pixel indices $p \in \{1, \ldots, N_{\text{pix}}\}$ is calculated:

- **Linear Grating:** $\lambda(p) = \frac{p-1}{d} + \lambda_1$.
- **Polynomial Grating:** Inverted via dense 1D grid interpolation over $\lambda \in [250, 1000]$ nm.
- **Prism Model:** Inverted via KDTree spatial nearest-neighbor search against dense evaluation grid $p(\lambda)$.

Wavenumbers $\tilde{\nu}$ (in cm^{-1}) are directly converted via:

$$\tilde{\nu}(p) = \frac{10^7}{\lambda(p)} \tag{41}$$

7.6 1D Gaussian Pump Spectrum Profile Diagnostics

Pump beam profiles in optical pulse shaper calibration are fitted with a 3-parameter Gaussian distribution:

$$G(x) = A \cdot \exp\left[-4\ln 2\left(\frac{x - \omega_0}{\text{FWHM}}\right)^2\right] \tag{42}$$

where A is peak amplitude, ω_0 is central frequency (cm^{-1}), and FWHM is Full-Width at Half-Maximum.

7.7 Supported Hardware Setup Configurations

The calibration engine pre-configures twelve spectrograph hardware setups:

Table 20: Pre-configured calibration setup codes and default parameter estimates.

Code	Setup Name	Default λ_0	Dispersion Guess	Degree	Default Bounds ($\Delta\lambda_{\text{rel}}$)
1	UoS TRIR	2000 cm^{-1}	0.30 px/cm^{-1}	2	[-1000, +1666] nm
2	UniGE TA	530 nm	1.10 px/nm	2	[-180, +210] nm
3	UniGE nsTA	530 nm	1.10 px/nm	2	[-180, +210] nm
4	UZH Lab 2	2000 cm^{-1}	0.30 px/cm^{-1}	2	[-455, +1250] nm
5	UniGE NIR-TA	1000 nm	1.00 px/nm	Prism	[-200, +600] nm
6	RAL Absorbance	2000 cm^{-1}	0.30 px/cm^{-1}	2	[-653, +882] nm
7	RAL Intensity	2000 cm^{-1}	0.30 px/cm^{-1}	2	[-653, +882] nm
8	UniGE TRIR (Int.)	2000 cm^{-1}	0.30 px/cm^{-1}	2	[-200, +200] nm
9	UniGE TRIR (Abs.)	2000 cm^{-1}	0.30 px/cm^{-1}	2	[-200, +200] nm
10	UniGE TRUVIS-II	530 nm	1.20 px/nm	3	[-150, +220] nm
11	Light Conversion HARPIA-TA (Int.)	515 / 440 nm	1.00 / 1.15 px/nm	2	[-165,+235] / [-120,+160] nm
12	PyMESS (`.h5`)	Auto-routed	Auto-routed	Auto	Routed by spectral range

HARPIA Range Presets (Setup 11). Setup 11 (`Light Conversion HARPIA-TA (Int.)`) provides two dedicated optical range presets selectable in the GUI via `harpia_preset`:

- **Vis Range** (`"Vis"`): $\lambda_0 = 515\,\text{nm}$, dispersion guess $d_0 = 1.0\,\text{px/nm}$, relative fit bounds $[-165, +235]$ nm (spanning 350–750 nm).
- **UV Range** (`"UV"`): $\lambda_0 = 440\,\text{nm}$, dispersion guess $d_0 = 1.15\,\text{px/nm}$, relative fit bounds $[-120, +160]$ nm (spanning 320–600 nm).

When loading a HARPIA `.dat` file containing a calibrated wavelength header, PyMORGAN automatically inspects the minimum wavelength ($\lambda_{\min} < 365\,\text{nm}$) to select the appropriate preset.

PyMESS HDF5 Automatic Pathway Routing (Setup 12). Setup 12 (`PyMESS`) reads `.h5` calibration datasets produced by PyMESS (supporting both spectrometer `probe_*` and pulse shaper `pump_*` measurements). PyMORGAN inspects the HDF5 metadata (`/meta/spectrometer/range`, `type`, central wavelength `cwl`, and grating density) to classify the spectral regime into `"Mid-IR"`, `"UV/Vis"`, or `"NIR"`, and routes the dispersion model and parameter bounds automatically via `resolve_effective_cal_type(12, spectral_range)`:

- **Mid-IR** ($\lambda_0 \geq 2500\,\text{nm}$ or $1000 < \lambda_0 < 2500$ with grating $< 200\,\text{l/mm}$): Follows the **UniGE TRIR** pathway (Setup 8, 2nd-order polynomial in wavenumber space).
- **UV/Vis** ($\lambda_0 < 850\,\text{nm}$): Follows the **UniGE TRUVIS-II** pathway (Setup 10, 3rd-order polynomial grating model in wavelength space).
- **NIR** ($850 \leq \lambda_0 < 2500\,\text{nm}$): Follows the **UniGE NIR-TA** pathway (Setup 5, SF10 glass prism Sellmeier dispersion model).

The initial parameter bounds and dispersion guesses for each mode can be queried programmatically with `get_default_calibration_params(cal_type_code, spectral_range=..., harpia_preset=...)`, which returns a dictionary with keys `p0`, `lb`, `ub`, `cwl_guess` and `degree`.

7.8 Graphical User Interface (Calibration Tab)

The GUI incorporates a dedicated **Calibration Tab** featuring a 3-Column Inspector Panel at the bottom that keeps all file selectors, dispersion parameters, and fit controls simultaneously visible alongside 4 synchronized subplots:

1. **Raw Measurements:** Overlay of Pump (Air) black curve, Pump (Solvent) red curve, Single Mask cyan shaded area ($\alpha = 0.4$), and Multiple Mask blue curve.
2. **Calculated Absorbance & Physical Baseline:** Purple absorbance spectrum with dashed orange background polynomial baseline.
3. **Reference Overlay:** Reference FTIR spectrum overlaid against experimental absorbance, with zero-levels aligned via `mpl_axes_aligner`.
4. **Calibration Fit:** Calibrated Wavelength (nm) vs. Pixel index dispersion curve with linear reference line and central crosshairs.

The bottom **3-Column Inspector Layout** is organised into:

- **Column 1 (Calibration Files & Type):** Hardware setup selector (`combo_cal_type`), contextual sub-selectors for `HARPIA Range` (*Vis Range* / *UV Range*) or `PyMESS Regime` (*Auto*, *Mid-IR*, *UV/Vis*, *NIR*), Reference FTIR dropdown, and file browsers for Pump (Air), Pump (Solvent), Single Mask, and Multiple Mask.
- **Column 2 (Calibration Parameters):** Side-by-side cards for **Spectrometer Setup & Bounds** (Detector array, Grating, CWL, Fit Axis Unit, and Relative Cut Range $[\Delta\lambda_{\min}, \Delta\lambda_{\max}]$) and **Dispersion Model & Fit Options** (Dispersion model radio buttons, Derivative mode, Baseline correction checkbox, Smoothing window, and Initial/Fitted Dispersion px/x).
- **Column 3 (Fit Control):** One-click ↻ **Fit Spec.** and ↻ **Fit Shaper** action buttons, calibration status banner, and export buttons (*Save Spec. Cal.*, *Save Shaper Cal.*, *Pump Fit*).

7.9 Programmatic Usage Example

```
import pymorgan.cal as cal

# 1. Load reference FTIR spectrum
ref = cal.load_reference_spectrum("FTIR-Dioxane.csv")

# 2. Load experimental probe spectrum (e.g. UniGE TRIR setup 9)
exp = cal.load_experimental_spectrum("pump_air.csv", cal_type_code=9)

# 3. Perform wavelength calibration fit
res = cal.fit_wavelength_axis(
    meas_y=exp.detector_data[0],
    ref_x=ref.spectral_axis,
    ref_y=ref.absorbance,
    cal_type_code=9,
    cwl=2000.0,
    use_derivative="1st",
)

print(f"Calibrated span: {res.wavelength_nm[0]:.2f} nm to {res.wavelength_nm[-1]:.2f}
    nm")

# 4. Save calibrated vector
```

```
cal.save_calibration_file(res.wavenumber_cm1, "output_dir", filename="CalibratedProbe.
    csv")

# 5. Fit probe beam Gaussian profiles (multi-detector)
probe_fit = cal.fit_probe_spectrum(exp.detector_data, exp.cwl)
for det_idx, fit in enumerate(probe_fit.fits):
    print(f"Detector {det_idx}: w0={fit.w0:.1f} cm-1, FWHM={fit.fwhm:.1f} cm-1")

# 6. Query default parameters for a given setup
params = cal.get_default_calibration_params(cal_type_code=9)
print(params["degree"]) # polynomial degree
print(params["cwl_guess"]) # initial central-wavelength guess
```

8 Extending PyMORGAN (optional)

This optional chapter explains how to (i) modify the GUI layout and behaviour and (ii) add a reader for a new instrument file format. Neither is required for day-to-day use.

8.1 Editing the GUI

The window layout lives in a single Qt Designer file, `pymorgan/gui/main_window.ui`; the controller `pymorgan/gui/main_window.py` binds to the widgets *by their object name*. Editing the layout therefore means editing the `.ui` file, and only *new* widgets need code.

8.1.1 Opening the layout

Qt Designer edits the `.ui` regardless of the Qt binding. The easiest way to launch it is using the provided command:

```
pymorgan-edit-gui
```

Alternatively, you can launch it manually using `uvx`:

```
uvx --from pyside6-essentials pyside6-designer src/pymorgan/gui/main_window.ui
```

8.1.2 Moving or resizing widgets

Drag, resize and regroup freely. Because the controller looks widgets up by object name (`getattr(self, "PP_CtrPlot_btn", ...)`), rearranging the layout breaks nothing as long as the object names are unchanged. Save the `.ui` and re-run `pymorgan-gui`; no reinstall is needed because the editable install reads the file at run time.

8.1.3 Adding a widget

1. In Designer, add the widget and give it a clear **object name** (e.g. `PP_exportFig_btn`). After loading it is available as `self.PP_exportFig_btn`.
2. In `main_window.py`, inside `_wire()`, connect it with the defensive helpers and write the slot:

   ```
   self._connect("PP_exportFig_btn", self.export_figure) # buttons (clicked)
   self._connect_text("SomeField", self.on_text_changed) # QLineEdit (
       editingFinished)
   ```

   `_connect`/`_connect_text` only attach if the widget exists, so adding-without-wiring (or removing a wired widget) never crashes the window.

3. Read the widget's state in the slot (`self.<name>.currentText()`, `.value()`, `.isChecked()`, ...).

8.1.4 Plot areas and icons

- **A new plot area:** promote a `QWidget` to the class `WidgetPlot` with header `pymorgan.gui.widgetplot` (right-click → *Promote to...*), exactly like the existing `PPaxes`. In code it then exposes `.figure`, `.ax` and `.canvas`, so any pipeline plotter can draw into it via `data.plot_contour(ax=widget.ax)`.
- **Icons:** do *not* assign icons in Designer — that embeds a compiled `.qrc` resource dependency that breaks `loadUi`. Instead drop the image in `pymorgan/gui/icons/` and add an entry to `_BUTTON_ICONS` in `main_window.py` (applied by `_apply_icons`).
- Keep signal/slot wiring in Python (in `_wire`), not in Designer's connection editor; the handlers live in `main_window.py`.

8.1.5 Do not rename

The controller references these object names; renaming them in Designer requires updating the matching strings in `main_window.py`: `PPaxes`, `MainTabs`, `PP_datatype_cbx`, `PP_datafolderlist_lst`, `RootDir_field`, the `PP_*Plot_btn` buttons, `PP_SubtactBkg_chk`, `PP_bkg_tmin/tmax` and `PP_Normalise_chk`. After any change, confirm the window still builds:

```
QT_QPA_PLATFORM=offscreen uv run pytest tests/test_gui.py
```

8.2 Adding a new data-format parser

A *loader* maps a file path to the fields a dataset needs. Registering it with the matching decorator makes its data-type name appear in `pm.available_*_loaders()` and in the GUI's data-type combo. Table 21 lists the three contracts.

Branch	Decorator	Returns
1-D	`@pm.register_loader`	`(Zavg_R, delays, probe, Units, Nscans, Zss_R, Zstdv`
2-D	`@pm.register_map_loader`	`(Z, pump, probe, delays, units, freq_units)`
steady-state	`@pm.register_spectrum_loader`	`(x, y, x_units)`

Table 21: Loader return contracts (see chapters 3, 4).

8.2.1 A 1-D reader

The signal is stored as `[Ndelays × Npixels × Ndetectors]`; the unit dictionary is built with `helpers.units2dic(unitsL, unitsT, unitsZ)`. A minimal comma-separated reader:

```
import numpy as np
import helpers as hlp
import pymorgan as pm

@pm.register_loader("MyTA")
def read_my_ta(path):
    raw = np.loadtxt(path, delimiter=",")
    delays = raw[1:, 0] # first column = delays
    probe = raw[0, 1:] # first row = probe axis
    Zavg = raw[1:, 1:][..., np.newaxis] # add a detector axis
```

```
    Units = hlp.units2dic("nm", "ps", "mOD") # (probe, time, signal) units
    Nscans = np.nan # no single-scan information
    Zss = np.nan * np.zeros_like(Zavg[..., np.newaxis])
    Zstdv = np.zeros_like(Zavg)
    return Zavg, delays, probe, Units, Nscans, Zss, Zstdv
```

The data type is then available everywhere:

```
data = pm.load_1D("scan.myta", data_type="MyTA")
```

Where to register. The decorator runs when its module is imported. For a permanent format, place the reader in `pymorgan/oneD/load.py` (imported with the package). For a one-off, defining it anywhere before calling `pm.load_1D` is enough.

Exposing it in the GUI. Add the file extension to `_DATATYPE_GLOB` in `pymorgan/gui/main_window.py` so the dataset browser lists matching files, for example `"MyTA": "*.myta"` (unlisted types fall back to `"*"`).

8.2.2 2-D and steady-state readers

The pattern is identical, only the returned tuple differs (Table 21). A 2-D reader returns the cube `Z[pump, probe, t2]` with its pump/probe/t2 axes, a unit dictionary and the spectral-axis unit string; a steady-state reader returns the x/y arrays and a best-guess spectral unit (or `None`). Register them with `@pm.register_map_loader("Name")` and `@pm.register_spectrum_loader("Name")` respectively.

8.2.3 The unit dictionary

`helpers.units2dic(unitsL, unitsT, unitsZ)` accepts the spectral unit `unitsL` (`"nm"`, `"cm^{-1}"` or `"eV"`), the time unit `unitsT` (e.g. `"ps"`), and the signal unit `unitsZ` (`"mOD"`, `"x1E3"` or `"int"` for intensity). The ΔA display convention (m OD vs $\times 10^3$) is then a presentation choice handled by the settings (§6).

9 License

PyMORGAN is free software released under the **GNU Affero General Public License version 3.0** (AGPLv3). You are free to run, inspect, modify and redistribute this software under the terms of the license. Any derivative works or modifications made available over a network or server must also make their complete corresponding source code available under the same license terms.