

PyRATE

Rate Analysis of Time-resolved Experiments
Kinetic analysis and user manual

Dr. Ricardo J. Fernández-Terán

29th July 2026

Contents

1	Introduction	2
2	Citing PyRATE	2
3	Installation	3
3.1	Environment and core installation	3
3.2	Co-development with PyMORGAN	3
3.3	Console scripts	4
3.4	Verifying the installation	4
3.5	Versioning	4
4	The PyMORGAN boundary	4
4.1	A minimal session	5
4.2	Units	5
4.3	Extraction of traces	5
5	Kinetic models	5
5.1	The bilinear model	5
5.2	Model families	6
5.3	Parallel and sequential models	6
5.4	Rate matrices	6
5.5	Predefined schemes	7
5.6	Writing a scheme	7
5.7	Scheme diagrams	7
5.8	Instrument response	7
5.9	The coherent artefact	8
5.10	Chirp	8
6	Fitting	8
6.1	Variable projection	8
6.2	Engines	8
6.3	Weighting and the fit statistic	9
6.4	Uncertainties	9
6.5	Convergence	9
6.6	Reproducibility	10
6.7	Testing	10

7	The graphical interface	10
7.1	Layout	10
7.2	The K-matrix editor	11
7.3	Layout is declared, not built	11
7.4	Structure of the code	11
7.5	Rendering	11
7.6	Start-up cost	11
8	Settings	12
8.1	Two settings objects	12
8.2	Design	12
8.3	Sections of the file	12
8.4	The configuration file	13
8.5	The settings editor	13
9	Extending PyRATE	13
9.1	Where new code goes	13
9.2	Adding a model	13
9.3	Adding an export to the public API	14
9.4	Adding a GUI panel	14
9.5	Conventions	14

1 Introduction

PyRATE fits time-resolved spectroscopy data: multi-exponential decays, global analysis with the amplitudes solved by variable projection, and target analysis with a general rate matrix, together with the decay-, evolution- and species-associated spectra that come out of them.

PyRATE is the analysis half of a two-project pair. Its sibling, **PyMORGAN**, owns loading, processing and plotting: the file-format readers and their registry, the `Dataset1D/Dataset2D` objects, the aesthetics layer and every standard plotter. The dependency is strictly one-way — `pyrate` imports `pymorgan`, never the reverse — and PyRATE reimplements none of it. A format PyRATE needs and PyMORGAN lacks is contributed upstream, not forked here. Chapter 4 makes that boundary explicit, because most confusion about where a piece of code belongs is resolved by it.

This manual documents the analysis model and the programming interface. Chapter 3 covers installation; chapter 4 the PyMORGAN boundary; chapter 5 the kinetic models and the rate-matrix formalism; chapter 6 the fitting engines, uncertainties and reproducibility requirements; chapter 7 the graphical interface; chapter 8 every setting and the configuration file; and chapter 9 how to add a model, an engine or a GUI panel.

Sections marked [\[planned\]](#) describe a design that is specified but whose implementation is still in progress; they are written down here so the interface is agreed before the code exists, and they are the parts of the manual to distrust when reading the current source.

2 Citing PyRATE

If you publish results obtained with PyRATE, please cite the following. The same list is printed by `pyrate-cite` and logged when the GUI starts.

1. The PyRATE / PyMORGAN software paper — *manuscript in preparation*.

2. R. J. Fernández-Terán, E. Sucre-Rosales, L. Echevarria, F. E. Hernández, *A Sweet Introduction to the Mathematical Analysis of Time-Resolved Spectra and Complex Kinetic Mechanisms: The Chameleon Reaction Revisited*, J. Chem. Educ. **2022**, 99, 2327–2337. 10.1021/acs.jchemed.2c00104
3. I. H. M. van Stokkum, D. S. Larsen, R. van Grondelle, *Global and target analysis of time-resolved spectra*, Biochim. Biophys. Acta Bioenerg. **2004**, 1657, 82–104. 10.1016/j.bbabbio.2004.04.011 — global and target analysis, and the variable-projection formalism (chapter 6).
4. M. N. Berberan-Santos, J. M. G. Martinho, *The integration of kinetic rate equations by matrix methods*, J. Chem. Educ. **1990**, 67, 375. 10.1021/ed067p375 — the eigenvector solution used by the propagator (section 5.4).

A routine implementing a published method also logs its own reference through `pyrate.cite` when it runs, so the log of a fit records which formalism produced it, and each reference appears only once per session.

3 Installation

PyRATE is managed with `uv` and requires Python 3.11 or later — the minimum is higher than PyMORGAN's, because PyRATE uses `tomllib` and `enum.StrEnum` from the standard library.

3.1 Environment and core installation

From the repository root:

```
uv venv # create .venv (Python >= 3.11)
uv pip install -e ".[gui,dev]" # editable install + dependencies
```

`uv` automatically uses the project's `.venv`: prefix one-off commands with `uv run` to avoid activating it manually.

3.2 Co-development with PyMORGAN

PyRATE depends on PyMORGAN. While the two are developed side by side under `Scripts/`, PyMORGAN is picked up from the sibling checkout as an editable path dependency, so an upstream edit is visible immediately without a reinstall:

```
# pyproject.toml
[tool.uv.sources]
pymorgan = { path = "../PyMORGAN", editable = true }
```

Replace that block with a plain version pin once PyMORGAN is published to an index. PyMORGAN's bundled fonts are installed once, from the PyMORGAN checkout (`uv run pymorgan-install-fonts`); PyRATE uses the same ones and deliberately does not ship a second installer.

Package	Minimum version	Purpose
pymorgan	0.6	Loading, datasets, plotting, aesthetics
numpy	2.1.0	Array operations
scipy	1.14.1	Optimisation and matrix exponentials
tomlkit	0.13	Settings file read/write
PyQt6	6.7.1	Graphical interface framework
PyQt6_sip	13.8.0	Qt bindings interface

Table 1: Core dependencies installed by `uv pip install -e .`

3.3 Console scripts

The editable install puts the following entry points on the path:

Command	Purpose
pyrate-gui	Launch the fitting interface
pyrate-edit-gui	Open main_window.ui in Qt Designer
pyrate-settings	Standalone editor for the analysis settings

Table 2: Console scripts, mirroring PyMORGAN’s. Presentation settings are edited with pymorgan-settings, not here.

3.4 Verifying the installation

```
uv run python -c "import pyrate; print(pyrate.__version__)"
uv run pytest
uv run python scripts/run_checks.py
```

run_checks.py exercises the settings round-trip, the lazy public API, the PyMORGAN boundary and the integrity of the Qt Designer layout; the Qt checks are skipped, not failed, on a headless machine. python scripts/run_changed_tests.py runs only the suites affected by the current diff, and falls back to the full suite whenever a change cannot be mapped — an unmapped change is never reported as tested.

3.5 Versioning

Every edit to the codebase must bump the version in src/pyrate/__about__.py, using python scripts/bump_version.py rather than editing it by hand. The scheme is 0.x.yymmdd.devN (PEP 440): yymmdd is the build date and N the iteration within that day, restarting at 1 on a new date. A bare letter suffix (0.1.260728d) is not valid PEP 440 and breaks the build. -check exits non-zero when the recorded version was not produced today, which suits a pre-commit hook.

4 The PyMORGAN boundary

Most questions of the form “where does this code belong?” are answered by Table 3. Read it before adding a module.

Concern	Owner	Entry point
File formats, loader registry	PyMORGAN	pm.load_1D, pm.register_loader
Dataset objects	PyMORGAN	pm.Dataset1D, pm.Dataset2D
Background, chirp, solvent	PyMORGAN	Dataset1D.background_correct, ...
Contour / spectra / kinetics plots	PyMORGAN	Dataset1D.plot_*
Species-spectra rendering	PyMORGAN	plot_species_spectra
Aesthetics, colormaps, unit labels	PyMORGAN	pm.Settings, pm.apply_style()
Kinetic models, rate matrices	PyRATE	pyrate.models
Fitting engines, global & target	PyRATE	pyrate.fit
Fit results, uncertainties	PyRATE	pyrate.results
Analysis-only plots	PyRATE	pyrate.plot

Table 3: Ownership of each concern across the two packages.

Three rules follow.

The dependency is one-way. `import pymorgan` inside PyRATE is expected; `import pyrate` inside PyMORGAN is a design error. PyMORGAN’s `plot_species_spectra` deliberately takes plain arrays (`Sfit`, `Taus`, `TauErr`, `isFixTau`, `modelType`) rather than a PyRATE result object, precisely so the arrow never has to reverse; the adapter that produces those arrays lives on the PyRATE side. `scripts/run_checks.py` asserts the direction.

No file readers here. Loading goes through `pm.load_1D(path, data_type=...)`, which resolves the format through PyMORGAN’s registry and therefore picks up every format PyMORGAN supports, present and future. A new format is contributed upstream to `pymorgan.oneD.load`, or registered at runtime with `pm.register_loader` — never forked into PyRATE.

One aesthetics system. Call `pm.apply_style()` and read `pm.get_settings()` for label style, colormap, time-axis convention and figure sizes. PyRATE’s own `Settings` holds analysis-only fields (chapter 8).

4.1 A minimal session

```
import pymorgan as pm
import pyrate as pr

data = pm.load_1D("scan.pdat", data_type="PDAT") # PyMORGAN loads
data.background_correct(tmin=-20, tmax=-5) # PyMORGAN processes
fit = pr.fit_global(data, n_components=3) # PyRATE fits
data.plot_species_spectra(*fit.as_species_args()) # PyMORGAN plots
```

4.2 Units

Signal data is in mOD, delays are in the dataset’s own time unit and the probe axis in its native spectral unit. Fitted lifetimes are reported in the dataset’s time unit and are never silently converted: a lifetime printed without its unit, or converted behind the user’s back, is a bug.

4.3 Extraction of traces

PyMORGAN removed its own `extract_kinetic/fit_kinetics` when PyRATE took over analysis, so there is no upstream extraction helper to call: that primitive lives here. For a headless fit, slice the dataset directly (`data.Z[:, idx, detector]`) rather than re-deriving pixel-lookup logic; `Dataset1D.plot_kinetics` returns the extracted data as its third value when a figure is wanted as well.

5 Kinetic models

5.1 The bilinear model

Every model in PyRATE produces a concentration matrix $\mathbf{C}$ of shape $N_{\text{delays}} \times N_{\text{components}}$. The data matrix is then bilinear,

$$\mathbf{D} \approx \mathbf{C} \mathbf{S}^T, \quad (1)$$

with $\mathbf{S}$ of shape $N_{\text{pixels}} \times N_{\text{components}}$ holding the component spectra. The kinetics enter only through $\mathbf{C}$, and the spectra only through $\mathbf{S}$; this separation is what makes global analysis tractable (section 6.1).

5.2 Model families

ModelType	Scheme	Spectra obtained
Parallel	independent decays	DAS (decay-associated)
Sequential	$A \rightarrow B \rightarrow C \rightarrow \dots$	EAS (evolution-associated)
Target	general $\mathbf{K}$, with branching	SAS (species-associated)

Table 4: Model families and the meaning of the resulting spectra.

The distinction is not cosmetic. A parallel model makes no claim about connectivity, and its DAS are linear combinations of the true species spectra; a sequential model assumes a single irreversible chain, and its EAS are the spectra of the successive evolving states; only a target model with an explicitly justified $\mathbf{K}$ yields spectra of actual species. The `modelType` string handed to PyMORGAN's `plot_species_spectra` must reflect which of the three was used, because it drives the legend wording upstream — mislabelling a DAS as an SAS is a scientific error, not a formatting one.

5.3 Parallel and sequential models

For a parallel model with lifetimes τ_i ,

$$C_{ni} = \exp(-t_n/\tau_i), \quad (2)$$

and for a sequential model the same expression is replaced by the analytic solution of the chain, which for distinct lifetimes is a sum of exponentials with the well-known amplitude factors $\prod_{j \neq i} (\tau_i^{-1} - \tau_j^{-1})^{-1}$. Both are special cases of the rate matrix below, and are implemented as such wherever that costs nothing, so there is one propagator to test rather than three.

5.4 Rate matrices

A target model is defined by its rate matrix $\mathbf{K}$, with off-diagonal element K_{ij} the rate from compartment j to compartment i and the diagonal the negative sum of all rates leaving that compartment. The populations obey

$$\frac{d\mathbf{c}(t)}{dt} = \mathbf{K} \mathbf{c}(t), \quad \mathbf{c}(0) = \mathbf{c}_0, \quad (3)$$

so that $\mathbf{c}(t) = \exp(\mathbf{K}t) \mathbf{c}_0$. $\mathbf{K}$ is built explicitly and $\mathbf{C}$ obtained by eigendecomposition where $\mathbf{K}$ is diagonalisable, and by a matrix-exponential propagator otherwise.

Exactly degenerate lifetimes. When two rates coincide, $\mathbf{K}$ is defective: the eigenvector matrix is singular and the eigenvalue solution does not exist, although the kinetics are perfectly well behaved (for $A \rightarrow B \rightarrow$ with equal rates the closed form is $c_B = kt e^{-kt}$). PyRATE splits the degeneracy by an antisymmetric diagonal perturbation of relative size 10^{-6} and continues. This is not a cosmetic choice: an optimiser routinely passes through equal lifetimes on its way to the optimum, and refusing to evaluate there would abort otherwise sound fits. The perturbation is far below any experimental precision and is logged.

Near-degenerate lifetimes. When two eigenvalues approach each other, the eigenvector matrix becomes ill-conditioned. The symptom in a fit is a pair of amplitudes of opposite sign that grow without bound while the residual barely improves. PyRATE guards this case explicitly and warns when two fitted lifetimes come within `degeneracy_warn_ratio` of each other (chapter 8); a fit that trips this warning should be re-run with one component fewer before its lifetimes are believed.

5.5 Predefined schemes

Branched schemes are kept in `pyrate.models.TARGET_SCHEMES`, keyed by a short identifier: equilibria (`A_eq_B` and its decaying variants), chains with an equilibrium (`A_eq_B_to_C`, `A_eq_B_eq_C`), branchings into a common product (`branch_to_common_D`) and parallel chains (`two_chains_decaying_ends`). A scheme carries its own species and rate counts, and these need not agree: `A <=> B`; `A ->`; `B ->` has two compartments and four rate constants. A target model therefore takes its size from the scheme, not from the requested number of components, and the number of free lifetimes is the scheme's rate count.

```
import pyrate as pr

model = pr.make_model("Target", scheme="A_eq_B_to_C")
C = model.concentrations(t, taus=[2.0, 5.0, 20.0, 200.0], irf_fwhm=0.15)
```

5.6 Writing a scheme

A target model is written as its reactions, not as a matrix:

```
A -> B : k1 # a step
B <-> C : k2, k3 # an equilibrium (forward, backward)
C -> : k4 # decay to the ground state
init A = 1 # optional initial populations
```

`pyrate.scheme_from_text` turns this into a scheme, and the rate matrix is *derived*: every reaction contributes $-k$ to the reactant's diagonal and $+k$ to the product's row, so the columns cannot fail to balance. Three consequences are worth stating, because they are what make the notation general:

- **Species are named.** `S1`, `ICT`, `T1` appear on the diagram, in the lifetime table and in the result, so a scheme reads as chemistry rather than as indices.
- **Rates are named, so they can be shared.** The same rate name on two arrows is *one* fitted parameter used twice — the natural way to impose “these two channels have the same rate”, which a rate matrix cannot express without a hand-written builder.
- **Any topology.** Chains, branches, equilibria, cycles, disconnected pools and non-decaying products are all just lines; nothing is special-cased, and `scheme_to_text` regenerates the notation so a scheme round-trips.

A malformed scheme is refused with the offending line number, never half-built; `check_scheme_text` returns that message without raising, which is what the editor's *Update K graph* button reports.

5.7 Scheme diagrams

`pyrate.plot_scheme` draws the compartment diagram of any model, scheme or fit result: one node per compartment, one arrow per rate, the ground state as a separate node, and an equilibrium as two opposed arrows. Arrows are labelled with the *rate constants themselves* — k_1 , k_2 , and a sum such as $k_1 + k_2$ where a compartment branches — rather than with numbers, so the diagram tells you which row of the rate table an arrow corresponds to when setting values and bounds. The mapping is derived, not assumed: feeding the scheme's $k \rightarrow \mathbf{K}$ builder a unit vector shows exactly where each constant appears. Pass `rate_labels="value"` for the lifetime of each step instead.

5.8 Instrument response

The instrument response is convolved into `C`, not into the data. For exponential models this is done analytically: the convolution of a Gaussian of width σ centred at t_0 with a decaying exponential is an exponentially modified Gaussian,

$$\left(g * e^{-t/\tau}\right)(t) = \frac{1}{2} \exp\left(\frac{\sigma^2}{2\tau^2} - \frac{t-t_0}{\tau}\right) \operatorname{erfc}\left(\frac{1}{\sqrt{2}} \left(\frac{\sigma}{\tau} - \frac{t-t_0}{\sigma}\right)\right), \quad (4)$$

which is both faster and better conditioned than numerical convolution. The three handling modes are `None` (no IRF), `Gaussian` (equation 4) and `Clip` (drop the delays below $\tau_{\min}$ instead of modelling the response).

5.9 The coherent artefact

The signal around $t = 0$ contains contributions — cross-phase modulation, two-photon absorption, stimulated Raman — that no kinetic model describes. Two treatments are supported: clipping the early delays, or adding artefact components (the IRF and its first derivatives) to the model. Clipping changes which data the fit saw, so the applied $\tau_{\min}$ is part of the result’s reproducibility payload (section 6.6) and not merely a plotting choice.

5.10 Chirp

Chirp is corrected *upstream*, in PyMORGAN, before the data reach a fit. PyRATE does not implement a second chirp correction, and a fit of visibly chirped data is a data-processing mistake rather than a modelling one.

6 Fitting

6.1 Variable projection

In equation 1 the spectra $\mathbf{S}$ enter linearly. For any fixed set of non-linear parameters (the lifetimes, t_0 and the IRF width) the best-fit spectra are therefore available in closed form,

$$\mathbf{S}^T = \left(\mathbf{C}^T \mathbf{C}\right)^{-1} \mathbf{C}^T \mathbf{D}, \quad (5)$$

and only the non-linear parameters need to go to the optimiser. This is *variable projection*: for a 500-pixel, three-component fit it reduces the parameter vector from about 1500 entries to three, and it conditions the problem far better than fitting amplitudes and lifetimes jointly. Amplitudes must not be placed in the optimiser’s parameter vector.

In practice equation 5 is evaluated through a least-squares solver rather than by forming the normal equations, which squares the condition number of $\mathbf{C}$.

6.2 Engines

Engine	Data	Output
Single-trace fit	one kinetic trace	lifetimes, amplitudes
Global analysis	full $\mathbf{D}$, free $\mathbf{K}$	lifetimes, DAS or EAS
Target analysis	full $\mathbf{D}$, fixed connectivity	rates, SAS

Table 5: The three fitting engines. One module each; a shared driver handles parameter packing, fixed/free masks and convergence reporting.

[planned] The engines themselves are under construction; the interface below is fixed, the implementations are not yet complete.


```
fit = pr.fit_global(data, n_components=3, model_type="Sequential")
print(fit.taus, fit.tau_err)
data.plot_species_spectra(*fit.as_species_args())
```

Every fit routine is callable headlessly: a fit must be reachable and testable from a plain script with no `QApplication`. The GUI is a caller, never a prerequisite. Where a long fit puts up a progress dialog, the optimiser reports through a plain callback and the GUI adapts it; the engine itself does not touch widgets, and guards on `QApplication.instance()` being `None`.

6.3 Weighting and the fit statistic

If the dataset carries a per-point noise estimate — a sibling `.pdats` file loaded alongside the `.pdats`, or the spread over single scans — the residuals can be weighted by it. PyMORGAN supplies the array through `Dataset1D.noise_array()`; PyRATE never estimates noise itself.

With weights $w = 1/\sigma$, and N the number of points actually fitted and p the number of free parameters, the statistic is a true reduced chi-squared,

$$\chi^2_\nu = \frac{1}{N-p} \sum_{n,k} \left(\frac{D_{nk} - (\mathbf{CS}^\top)_{nk}}{\sigma_{nk}} \right)^2, \quad (6)$$

so that $\chi^2_\nu \approx 1$ means the residuals sit at the noise level, $\gg 1$ that the model is too simple (or the noise underestimated), and $\ll 1$ that the data are being overfitted (or the noise overestimated). Without weights the reported statistic is the plain sum of squared residuals, which carries no such interpretation and is not comparable between datasets. The read-out in the GUI is therefore labelled *SSR* or *red. chi2* according to which one is in force, and never simply “ χ^2 ”.

The parameter count p includes the $N_p N_c$ linear amplitudes: variable projection solves them in closed form, but it still spends those degrees of freedom, so counting only the non-linear parameters would flatter the fit. The value obtained with the non-linear-only denominator is reported alongside (`FitStatistic.value_nonlinear_dof`), because the two differ by a large factor and the choice should never be implicit.

Two practical points. Noise values are clipped from below at `noise_floor_fraction` of the median, because a single near-zero σ would otherwise dominate the cost function entirely; and points with non-finite or non-positive σ are dropped from the fit along with their data, with the count recorded on the result. Weights are diagonal: correlation of the noise between pixels is ignored, so χ^2_ν should not be read as more rigorous than that assumption allows.

Requesting weights for a dataset that has no noise array raises. Falling back silently to an unweighted fit while still calling the result a reduced chi-squared would be a misreported statistic, which is worse than a refusal.

6.4 Uncertainties

One-sigma uncertainties are reported from the linearised covariance estimate at the optimum. They are conditional on the model being correct, and they are not confidence intervals for the choice of model: a three-component fit of two-component data can return small uncertainties on three meaningless lifetimes. Parameters held fixed receive no uncertainty and are flagged as fixed, so that a fixed lifetime can never be mistaken for a well-determined one.

6.5 Convergence

A non-converged fit raises, or is flagged on the result object; it is never returned as though it had converged. The optimiser’s own termination report (the reason for stopping, the iteration count, the final cost) travels with the result. Hitting `max_iterations` is a non-convergence, not a result.

6.6 Reproducibility

A result object carries enough to reproduce itself:

- the model identity (family, number of components, the **K** where applicable);
- the initial guesses and the bounds;
- which parameters were held fixed;
- the delay and probe ranges actually fitted, including any clipping applied for the coherent artefact;
- the solver's convergence report.

A lifetime reported without its uncertainty and its fixed/free flag is not a result. Results serialise through `pyrate.io`, which handles fit sessions — parameters, masks, results — and not raw spectroscopy formats.

6.7 Testing

Synthetic-recovery tests are the backbone of the test suite: build data from known parameters, fit it, and assert that the parameters come back within tolerance. A fitting project without these is untested regardless of its line coverage. The failure modes are tested too — a non-converged fit must be flagged rather than silently returned, and near-degenerate lifetimes must not produce exploding amplitudes without a warning.

7 The graphical interface

Launch with `uv run pyrate-gui` (or `python -m pyrate.gui`). The window reuses PyMORGAN's plot widget, plot-controls panel and theme so the two applications look identical side by side.

7.1 Layout

The window is a single narrow control column on the left and a plot area on the right; nothing is scattered across the width, so the default geometry stays compact.

Left column Four stacked groups: *Data* (load, clear, the data-type selector and the loaded-state lamp), *Model Selection* (*Parallel* / *Sequential* / *Target* / *ODE-defined*), *Components* (the lifetime table with its bounds and fixed flags, and the add/remove buttons), *IRF and time-zero*, and *Fit options* (algorithm, restrict-to-display, noise weighting).

Fit control A single strip above the plots: the data / fit / residuals view selector, *Preview*, *FIT DATA*, *Reset*, and the statistic read-out.

Plot area One figure, not three, so the window carries a single navigation toolbar. Its axes form a 3×3 grid: the contour takes the top-left 2×2 block (twice the size of the others), the kinetics panel the 2×1 column beside it and the transient spectra the 1×2 row below. The bottom-right cell holds no axis — the *Additional Plots and cuts* group sits there.

The kinetics panel is drawn *rotated* (PyMORGAN's `swap_axes`), so its delay axis runs vertically like the contour's. Both shared axes then line up — probe between the contour and the spectra, delay between the contour and the kinetics — and the duplicated tick labels are dropped. Measured data are drawn as translucent points and the fit as a thin line over them, both configurable in the `[plots]` section of `settings.toml`.

Bottom The plot-controls panel: axis limits, colour scale, contour levels, time-axis convention and probe units.

Buttons are coloured by function, following the same palette as PyMORGAN — a contour button looks like a contour button in both applications — and the mapping lives in `gui/mw_common.py` rather than in the `.ui`, so it stays theme-responsive. Controls whose feature is not implemented yet are disabled and carry the reason in their tooltip, rather than being present and silently doing nothing.

7.2 The K-matrix editor

Define model... opens the scheme editor: the scheme as text on the left (section 5.6), its graph and the derived rate matrix on the right. *Update K graph* redraws and checks — a scheme that does not parse produces its error, with the line number, instead of a picture, so the button answers “would this scheme work?” before a fit is started, and *OK* stays disabled until it does. The matrix is shown with every rate set to 1, since what is being checked there is the structure, not the values. Accepting the scheme sets the lifetime table to one row per rate constant, named after it.

7.3 Layout is declared, not built

All widgets, layouts, containers and controls are declared in the Qt Designer file `src/pyrate/gui/main_window.ui`. Python code binds signals, slots, dynamic styles and data states to widgets defined there; it does not construct them. Edit the layout with

```
uv run pyrate-edit-gui # or: python -m pyrate.gui.designer
```

The plot area is promoted to PyMORGAN’s `WidgetPlot`, and the whole plot-controls group (`PC_box`) is the one from PyMORGAN’s layout, name for name, so that `pymorgan.gui.plot_controls.PlotControlsPanel` drives it unmodified. Renaming a `PC_*` widget breaks that contract; a test asserts the full set is present.

7.4 Structure of the code

`MainWindow` is assembled from per-tab mixins in `gui/tabs/` (`DataTabMixin` for loading and the embedded previews, `ModelTabMixin` for the model tables). A mixin is a plain method container with no state of its own, so `self` is always the main window. `main_window.py` keeps construction, wiring, menus and theming only; shared constants live in `gui/mw_common.py` and dialogs in `gui/dialogs.py`.

Wiring is defensive: a connection is made only if the named widget exists, so the window still loads while the layout is being edited in Designer, and the missing widget is logged rather than raising at start-up.

7.5 Rendering

A contour re-plot cannot be updated artist by artist — the level set itself changes — so rapid control changes, such as a colour-scale slider drag that emits one signal per step, are coalesced into a single render through a short-interval timer. Limit changes that do not alter the levels re-apply the axis limits without re-plotting.

7.6 Start-up cost

`gui/app.py` imports neither `pyrate` nor `pymorgan` at module level, so the window (and, where enabled, the splash screen) can appear before `scipy`, `matplotlib` and the loader registries are paid for. The same discipline applies to the package itself: `pyrate/__init__.py` binds its exports through a PEP 562 `__getattr__`, so `import pyrate` is nearly free and each module is imported on first use of a name it provides.

8 Settings

8.1 Two settings objects

Presentation — label style, colormap, time-axis convention, figure sizes — belongs to `pymorgan.Settings` and is read through `pm.get_settings()` after `pm.apply_style()`. PyRATE’s own `Settings` holds *analysis-only* fields, listed in Table 6. There is deliberately no second aesthetics system.

Field	Default	Group	Meaning
<code>n_components</code>	2	fit	Components proposed for a new fit
<code>model_type</code>	Sequential	fit	Default model family
<code>offset</code>	False	fit	Add an infinite-lifetime component
<code>irf_mode</code>	Gaussian	fit	None / Gaussian / Clip
<code>irf_fwhm</code>	None	fit	Fixed IRF width; None fits it
<code>t_min</code>	None	fit	Clip delays below this (mode Clip)
<code>ftol</code>	10^{-8}	solver	Termination tolerance on the cost
<code>xtol</code>	10^{-8}	solver	Termination tolerance on the parameters
<code>max_iterations</code>	2000	solver	Cap on iterations; exceeding it is a failure
<code>variable_projection</code>	True	solver	Solve amplitudes in closed form
<code>report_uncertainties</code>	True	solver	Report linearised $1\text{-}\sigma$ errors
<code>use_noise_weights</code>	False	solver	Weight residuals by the per-point noise
<code>noise_floor_fraction</code>	10^{-3}	solver	Clip σ below this fraction of the median
<code>degeneracy_warn_ratio</code>	1.2	solver	Warn when two lifetimes come this close
<code>default_datadir</code>	None	paths	Directory offered by the file dialog

Table 6: The analysis settings. Times are in the dataset’s own time unit (section 4.2).

Turning `variable_projection` off is almost always the wrong choice (section 6.1); it is exposed for diagnostics only.

8.2 Design

`Settings` is a dataclass and **the dataclass field is the single source of truth**. `__post_init__` (which coerces enums), `to_dict`, `from_dict` and `update_settings` all derive from `dataclasses.fields()`, so adding a setting needs only the field. Enums are `enum.StrEnum`: `str(member)` is the value, so they serialise and compare as plain strings while staying type-checked.

`Settings.field_specs()` is hand-written and drives the GUI settings panel: label, kind (choice / float / int / bool / text), optional choices / min / max / step / tooltip, and tab. *A setting without an entry there does not appear in the panel* — which is the intended way to hide an internal knob, and the usual explanation for a new setting not showing up.

8.3 Sections of the file

`settings.toml` is grouped by concern, with a comment above every key: `[fit]` (defaults for a new fit), `[solver]` (optimiser limits, weighting, reporting), `[irf]`, `[lda]` (the lifetime-density module), `[gui]` and `[paths]`. The mapping from field to section is `Settings.field_sections()`, and it is exhaustive: a field with no section raises on save rather than being written where the loader would not find it. A flat file — the older layout — is still read, so an existing configuration is never orphaned.

The `[gui]` block holds the interface defaults: `gui_theme`, `splash_duration`, the default window size and whether to restore it, the bounds and precision a new row of the lifetime table starts with, and what happens after a fit (auto-plot, auto-open the scheme diagram, auto-save the session). Plot aesthetics are *not* here; they belong to PyMORGAN.

The [lda] block is read by the lifetime-density module: grid size and limits, the regularisation method (L-curve / GCV / Fixed), the fixed strength, and whether the distribution is constrained non-negative.

8.4 The configuration file

Settings are read from and written to `settings.toml` with `tomlkit`, so user comments survive a save. The file is looked up in the working directory first, then at the project root.

```
import pyrate as pr

pr.load_settings("settings.toml")
pr.update_settings(n_components=3, irf_mode="Clip", t_min=0.3)
pr.save_settings("settings.toml")
```

`update_settings` rejects unknown names rather than storing them, so a typo fails immediately instead of silently doing nothing.

8.5 The settings editor

`uv run pyrate-settings` opens a standalone editor whose pages and fields are generated from `field_specs()`; *Save permanently* writes them back to `settings.toml`. A value the settings layer rejects is logged and reverted in the widget rather than silently dropped. `settings.py` itself never imports Qt.

9 Extending PyRATE

9.1 Where new code goes

`models/` Kinetic model definitions. A model owns its parameter vector layout, its bounds, and the mapping from parameters to `C`. Models are pure NumPy and must be importable without Qt.

`fit/` The solvers, one module per engine, plus the shared driver that handles parameter packing, fixed/free masks and convergence reporting.

`results/` Result dataclasses and their serialisation, carrying the reproducibility payload of section 6.6.

`plot/` Only what PyMORGAN cannot express: residual matrices, concentration profiles $c(t)$, **K** / compartment diagrams, lifetime-density maps. Everything else is delegated upstream.

`io/` Import and export of fit sessions — not raw spectroscopy formats, which are PyMORGAN's.

`gui/` The PyQt6 application (chapter 7).

9.2 Adding a model

A model supplies a concentration matrix and the metadata the fit driver and the result object need: the parameter names, their bounds, and the `modelType` string that will reach `plot_species_spectra`. Build **K** explicitly and reuse the shared propagator (section 5.4) rather than deriving a closed-form solution per scheme — one propagator is easier to test, and the degeneracy guard then applies automatically.

Every new model needs a synthetic-recovery test: generate data from known parameters, fit it, assert the parameters come back within tolerance.

9.3 Adding an export to the public API

The public API is lazy (PEP 562). A new export needs an entry in `_EXPORTS` (name $\rightarrow$ module) *and* in `__all__`, and should be added to the `TYPE_CHECKING` block as well so that IDEs and type checkers still resolve it. Heavy third-party imports used by a single function (`scipy.optimize`, `scipy.linalg`, `h5py`) belong *inside* that function.

9.4 Adding a GUI panel

Declare the widgets in `main_window.ui` with Qt Designer (`uv run pyrate-edit-gui`), then bind them from a mixin in `gui/tabs/`. Do not build widgets in Python: the layout must stay editable in Designer. Long operations are wrapped with `busy_guard` so a second fit cannot start inside the first, and progress is reported through a callback the GUI adapts, keeping the engine headless.

9.5 Conventions

Logging, not print. Library code uses `logger = get_logger(__name__)`. `logger.info` is for messages the user should see — fit summaries, recovered lifetimes, convergence status, method citations; `logger.debug(..., exc_info=True)` goes inside an `except` block whose failure is recoverable, so that nothing fails silently. `print` is reserved for console entry points and explicit `*_console` helpers. To debug a swallowed failure: `logging.getLogger("pyrate").setLevel(logging.DEBUG)`.

Method citations. A routine implementing a published method — a global-analysis formalism, a target-analysis convention, a regularisation scheme — logs its reference through `logger.info` when it finishes. An inherited habit from PyMORGAN; keep it.

No placeholders. Write fully functional code or raise `NotImplementedError`. Deprecated code moves to `old_code/` (gitignored, and excluded from lint and packaging) rather than being deleted.

Matplotlib. Use standard `mathtext`; never enable `text.settex=True`.

Style. Target Python 3.11, 100-character lines, double quotes: `uvx ruff@latest format src tests` and `uvx ruff@latest check src tests scripts examples`.