

PyRATE-TA

Rate Analysis & Target-model Engine for Transient Absorption
Kinetic analysis and user manual

Dr. Ricardo J. Fernández-Terán

11th October 2026

Contents

1	Introduction	2
2	Citing PyRATE-TA	3
3	Installation	3
3.1	Quick install via PyPI (Recommended)	4
3.2	Installation from Source (Development)	4
3.3	Co-development with PyMORGAN	4
3.4	Console scripts	5
3.5	Verifying the installation	5
3.6	Versioning	5
4	The PyMORGAN boundary	5
4.1	A minimal session	6
4.2	Units	6
4.3	Extraction of traces	6
5	Kinetic models	6
5.1	The bilinear model	6
5.2	Model families	7
5.3	Parallel and sequential models	7
5.4	Rate matrices	7
5.5	Predefined schemes	8
5.6	Writing a scheme	8
5.7	Scheme diagrams	8
5.8	Instrument response	9
5.9	The coherent artefact	9
5.10	Chirp	9
6	Fitting	9
6.1	Variable projection	9
6.2	Engines	9
6.3	The coherent artefact	10
6.4	Constrained amplitudes and sign masks	11
6.5	Ground-state bleach and absolute species spectra	11
6.6	Lifetime density analysis	11
6.7	Watching a fit run	12

6.8	Weighting and the fit statistic	12
6.9	The fixed/free mask	13
6.10	Uncertainties	13
6.11	Convergence	13
6.12	Results	13
6.12.1	LDA result	14
6.13	Reproducibility	14
6.14	Fit session files	14
6.15	Steady-state spectra and pure positive species recovery (GSB and SE)	14
6.16	Testing	15
7	The graphical interface	15
7.1	Layout	16
7.2	Fitting on a small screen	16
7.3	The K-matrix editor	17
7.4	The LDA tab	17
7.5	Reading a fit	17
7.6	Software updates	18
7.7	Layout is declared, not built	18
7.8	Structure of the code	18
7.9	Rendering	19
7.10	Start-up cost	19
8	Settings	19
8.1	Two settings objects	19
8.2	Design	19
8.3	Sections of the file	20
8.4	The configuration file	21
8.5	The settings editor	21
8.6	Both files are loaded	21
8.7	How results are quoted	21
9	Extending PyRATE-TA	22
9.1	Where new code goes	22
9.2	Adding a model	22
9.3	Adding an export to the public API	22
9.4	Adding a GUI panel	22
9.5	Adding a plot	22
9.6	Conventions	23
10	License	23

1 Introduction

PyRATE-TA fits time-resolved spectroscopy data: multi-exponential decays, global analysis with the amplitudes solved by variable projection, target analysis with a general rate matrix, and lifetime density analysis (LDA) over a dense regularised grid — together with the decay-, evolution- and species-associated spectra that come out of them.

PyRATE-TA 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-TA reimplements none of it. A format PyRATE-TA 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, result objects, session files 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.

2 Citing PyRATE-TA

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

1. The PyRATE-TA / 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.bbambio.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).
5. P. C. Hansen, *Analysis of discrete ill-posed problems by means of the L-curve*, SIAM Review **1992**, 34, 561–580. 10.1137/1034115 — L-curve corner detection for automatic regularisation in LDA (section 6.6).
6. G. H. Golub, M. Heath, G. Wahba, *Generalized cross-validation as a method for choosing a good ridge parameter*, Technometrics **1979**, 21, 215–223. 10.1080/00401706.1979.10489751 — GCV alpha selection for LDA (section 6.6).
7. C. Slavov, H. Hartmann, J. Wachtveitl, *Implementation and Evaluation of Data Analysis Strategies for Time-Resolved Optical Spectroscopy*, Anal. Chem. **2015**, 87, 2328–2336. 10.1021/ac504348h — the OPTIMUS software and lifetime density analysis (LDA) formalism as applied to ultrafast spectroscopy (section 6.6).
8. G. F. Dorlhiac, C. Fare, J. J. van Thor, *PyLDM – An open source package for lifetime density analysis of time-resolved spectroscopic data*, PLOS Comput. Biol. **2017**, 13, e1005528. 10.1371/journal.pcbi.1005528 — open-source lifetime density analysis algorithms.

A routine implementing a published method also logs its own reference through `pyrate_ta.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-TA requires **Python 3.12 or later**. It can be installed directly from PyPI or installed from source using **uv**.

3.1 Quick install via PyPI (Recommended)

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

```
pip install pyrate-ta
# or with uv:
uv pip install pyrate-ta

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

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

```
uvx --from pyrate-ta pyrate-ta-gui
```

3.2 Installation from Source (Development)

From the repository root:

```
uv venv # create .venv (Python >= 3.11)
uv pip install -e ".[gui,dev]" # editable install + dependencies
uv run pymorgan-install-fonts # PyMORGAN's fonts, in *this* environment
```

The font step is not optional in practice. PyRATE-TA uses PyMORGAN's font stack and ships no second installer, but Matplotlib registers fonts per environment: skip it and every figure falls back to DejaVu Sans, which is why a PyRATE-TA plot can look unlike a PyMORGAN one while both claim the same style. The GUI checks at start-up and logs the command if the preferred fonts are missing.

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

3.3 Co-development with PyMORGAN

PyRATE-TA 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. The same editable dependency puts pymorgan-install-fonts on PyRATE-TA's path, which is how the fonts are registered for this environment.

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 .

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

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

3.4 Console scripts

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

3.5 Verifying the installation

```
uv run python -c "import pyrate_ta; print(pyrate_ta.__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.

3.6 Versioning

Every edit to the codebase must bump the version in `src/pyrate_ta/\_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	<code>pm.load_1D</code> , <code>pm.register_loader</code>
Dataset objects	PyMORGAN	<code>pm.Dataset1D</code> , <code>pm.Dataset2D</code>
Background, chirp, solvent	PyMORGAN	<code>Dataset1D.background_correct</code> , ...
Contour / spectra / kinetics plots	PyMORGAN	<code>Dataset1D.plot_*</code>
Species-spectra rendering	PyMORGAN	<code>plot_species_spectra</code>
Aesthetics, colourmaps, unit labels	PyMORGAN	<code>pm.Settings</code> , <code>pm.apply_style()</code>
Kinetic models, rate matrices	PyRATE-TA	<code>pyrate_ta.models</code>
Fitting engines, global & target	PyRATE-TA	<code>pyrate_ta.fit</code>
Fit results, uncertainties	PyRATE-TA	<code>pyrate_ta.results</code>
Analysis-only plots	PyRATE-TA	<code>pyrate_ta.plot</code>

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

Three rules follow.

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

No file readers here. Loading goes through `pm.load\_1D(path, data\_type=\dots)`, 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-TA.

One aesthetics system. Call `pm.apply\_style()` and read `pm.get\_settings()` for label style, colourmap, time-axis convention and figure sizes. PyRATE-TA’s own `Settings` holds analysis-only fields (chapter 8).

4.1 A minimal session

```
import pymorgan as pm
import pyrate_ta 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-TA 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-TA 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-TA 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-TA 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-TA 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_ta.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_ta 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\_ta.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_ta.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.

Every family is laid out **horizontally**: population flows left to right and the ground state sits below, so a branched scheme spreads across a wide, short figure instead of stacking a column that wastes the width and shrinks the nodes.

The scheme is recovered from whatever the object carries. A fit records its scheme twice — as the notation (scheme_text) and as a key — and the *text* is parsed first, so a scheme written in the editor keeps its species names and rate symbols on the diagram. This matters because a typed scheme is stored under the key "custom", which is in no registry: consulting the key alone made drawing one fail outright.

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,

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

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 $t_{\min}$ 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 $t_{\min}$ is part of the result's reproducibility payload (section 6.13) and not merely a plotting choice.

5.10 Chirp

Chirp is corrected *upstream*, in PyMORGAN, before the data reach a fit. PyRATE-TA 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 **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 **C**.

6.2 Engines

```
# Global analysis
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())
```

Engine	Data	Output
Single-trace fit	one kinetic trace	lifetimes, amplitudes
Global analysis	full D , free K	lifetimes, DAS or EAS
Target analysis	full D , fixed connectivity	rates, SAS
LDA	full D , dense grid	lifetime density map

Table 5: The four fitting engines. Global, target and single-trace share the variable-projection driver; LDA is a separate regularised linear solve.

```
# Preview (no optimisation)
prev = pr.preview_global(data, taus=[1.0, 10.0, 100.0])

# Target analysis
fit = pr.fit_target(data, scheme="A_eq_B_to_C", taus=[2.0, 5.0, 20.0, 200.0])

# Single-trace
fit = pr.fit_kinetics(t, y, n_components=2)

# Lifetime density analysis
lda = pr.fit_lda.solve_lda(data, n_taus=100, alpha="auto", non_negative=True)
```

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`.

Preview mode. `pr.preview\_global` evaluates a model at the given lifetimes *without* optimising. The spectra still follow from the data (they are the least-squares solution for the given **C**), so a preview shows what those lifetimes actually imply and what the residual would be. Nothing is fitted, so there is no convergence report and no uncertainties — and the result says so rather than presenting a guess as an answer.

Component ordering. Variable projection sees only the column space of **C**, so permuting the lifetimes fits equally well and the optimiser’s ordering depends on the initial guess. For parallel and sequential models the components are sorted by lifetime on output, the spectra permuted to match, and the fact recorded on the result (`sorted\_by\_lifetime`). A target model is left exactly as fitted: its compartments are defined by the scheme, and reordering them would misrepresent it.

6.3 The coherent artefact

Around time zero a transient carries signal no kinetic model describes: cross-phase modulation and two-photon absorption in the solvent and the cell windows. Left alone, the fit absorbs it into the shortest lifetime, which is then not a lifetime at all.

Tickling *Fit coherent artefact* (beside the IRF table, or `coherent\_artifact=True` on any engine) appends three columns to the design matrix: the instrument response and its first two derivatives,

$$g(t), \quad g'(t), \quad g''(t),$$

each scaled to unit maximum so the second derivative — larger than the Gaussian by $1/\sigma^2$ — does not dominate the conditioning. Their amplitudes are solved per probe pixel exactly like a species spectrum, so the artefact gets spectra of its own instead of distorting τ_1 . The description follows Kovalenko *et al.*, Phys. Rev. A **59**, 2369 (1999), which the log cites when the basis is first built.

The artefact columns are *not* species: the result records how many there are (`n\_artifact`), as `\_species\_args()` hands over only the kinetic ones — a column with no lifetime must not be labelled with someone else's — and `artefact\_spectra` exposes the rest. They are worth a look: an artefact spectrum with amplitude far from time zero is a sign it is absorbing real kinetics. The option needs a Gaussian IRF, since the basis is built from it; without one the checkbox is disabled and the engines refuse rather than inventing a width.

6.4 Constrained amplitudes and sign masks

When physical constraints are imposed on the species spectra — non-negative absorption for real excited states, non-positive for a ground-state bleach contribution — variable projection solves a bounded linear least-squares problem per probe pixel rather than an unconstrained linear solve,

$$\min_{\mathbf{S}_p} \|\mathbf{C}_w \mathbf{S}_p - \mathbf{d}_{w,p}\|^2 \quad \text{subject to} \quad \mathbf{l} \leq \mathbf{S}_p \leq \mathbf{u}. \quad (6)$$

A `sign\_mask` array of length N_c controls this: $+1$ enforces $S_p \geq 0$, -1 enforces $S_p \leq 0$, and 0 is unconstrained. When all entries are zero the problem collapses to the standard unconstrained solve. Simple non-negative least squares uses `scipy.optimize.nnls`; a mixed mask uses `scipy.optimize.lsq\_linear`. The constrained solver is in `pyrate\_ta.fit.constrained`.

6.5 Ground-state bleach and absolute species spectra

Transient absorption and vibrational data are difference spectra: $\mathbf{S}_{\text{diff}} = \mathbf{S}_{\text{excited}} - f \mathbf{GS}$. Adding back a scaled ground-state spectrum converts them to absolute absorption spectra:

$$\mathbf{S}_{\text{abs}}[:, i] = \mathbf{S}_{\text{diff}}[:, i] + f_i \cdot a \cdot \mathbf{GS}, \quad (7)$$

where a is the ground-state scale factor and f_i is the fraction of ground state bleached by species i .

`pyrate\_ta.fit.groundstate.compute\_absolute\_spectra` performs this conversion. The maximum physically allowable scale factor a_{max} — the largest a before any pixel in any absolute spectrum turns negative — is computed first, so the automatic choice (`gs\_scale="auto"`) selects the tightest non-negative solution. The result stores `S\_abs`, `gs\_scale` and `a\_max` for plotting and reporting.

6.6 Lifetime density analysis

Instead of fitting N discrete lifetimes, LDA solves for a continuous distribution of amplitudes over a dense, logarithmically spaced grid of fixed lifetimes,

$$\min_{\mathbf{S}^T} \left\| \mathbf{W} \odot \left(\mathbf{D} - \mathbf{C}(\boldsymbol{\tau}_{\text{grid}}) \mathbf{S}^T \right) \right\|^2 + \alpha^2 \left\| \mathbf{L} \mathbf{S}^T \right\|^2, \quad (8)$$

where $\mathbf{L}$ is a regularisation operator and α controls the balance between fit quality and smoothness.

Regularisation operators. Three are available: the identity (ridge, $\mathbf{L} = \mathbf{I}$), a first-difference operator ($\mathbf{L} = \mathbf{D}_1$, penalising slope), and a second-difference operator ($\mathbf{L} = \mathbf{D}_2$, penalising curvature; the default). The augmented system $[\mathbf{C}; \alpha \mathbf{L}] \mathbf{S}^T = [\mathbf{D}; \mathbf{0}]$ is solved in one least-squares call.

Automatic alpha selection. When `alpha="auto"` three methods are available:

`lcurve` The corner of the log-residual–log-norm curve — maximum curvature by the Menger formula (Hansen 1992). This is the default.

`gcv` Generalised cross-validation (Golub et al., 1979).

`morozov` Morozov discrepancy principle: match the residual norm to the estimated noise level, estimated from the pre-time-zero region.

Additional options.

`non\_negative` Enforce $S \geq 0$ per probe pixel via NNLS after the alpha scan. Physically motivated for some spectral regions.

`svd\_components` Pre-filter the data matrix with the top K SVD components before constructing $\mathbf{C}$, suppressing noise-dominated singular values.

`n\_bootstraps` Monte Carlo resampling of the residuals: refits the LDA `n\_bootstraps` times and records the standard deviation of the integrated amplitude $A(\tau) = \sum_p |S_{p,i}|$ as a confidence estimate.

`find\_peaks` Automatic peak centroid detection from $A(\tau)$, using `scipy.signal.find\_peaks` with a 5% prominence threshold.

`coherent\_artifact` Appends the same IRF-derivative basis as the discrete engines (section 6.3) to absorb time-zero contributions, leaving the regularised columns for the kinetics only.

The result is an `LDAResult` object (section 6.12.1) carrying the density map, the alpha scan diagnostics, residuals, and optional bootstrap errors. It exports the map to PyMORGAN's PDAT format via `.to\_pdat(path)`.

6.7 Watching a fit run

A global fit blocks for seconds to minutes, and the cost alone says little about what is going wrong. Setting `[gui] fit\_monitor\_every = N` opens a window showing one bar per parameter, its current value above it and its name below, refreshed every N residual evaluations; $N = 0$ (the default) never opens it. The scale is symlog by default (`fit\_monitor\_scale`), which is the only one that shows a lifetime of 3000 and a time zero of ≈ 0 on the same axis. Fixed parameters are drawn greyed, so a bar that never moves is not mistaken for one that has converged.

The rate is a setting because each update costs a redraw *and* an event-loop pump: at $N = 1$ a fast fit spends more time drawing than fitting, while $N = 5\text{--}10$ is enough to watch a lifetime run away to the edge of the delay window. The drawing lives in `pyrate_ta.plot.monitor` and is Qt-free, so the same bars — and `plot\_parameter\_history`, the trajectory against evaluation number — can be produced from a script.

6.8 Weighting and the fit statistic

If the dataset carries a per-point noise estimate — a sibling `.pdatn` file loaded alongside the `.pdat`, or the spread over single scans — the residuals can be weighted by it. PyMORGAN supplies the array through `Dataset1D.noise\_array()`; PyRATE-TA 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,

$$\chi_\nu^2 = \frac{1}{N-p} \sum_{n,k} \left(\frac{D_{nk} - (\mathbf{CS}^\top)_{nk}}{\sigma_{nk}} \right)^2, \quad (9)$$

so that $\chi_\nu^2 \approx 1$ 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.

6.9 The fixed/free mask

A caller that knows only about the lifetimes — the GUI reads its mask from the lifetime table — may pass a mask of “`n_lifetimes`” entries while the parameter vector is longer. Time zero and the IRF width are appended to that vector *only when they are being fitted*, so the entries that follow are free by construction and the mask is padded rather than refused. A mask of any *other* length is still an error, naming the parameters expected: padding must complete an incomplete answer, not reinterpret a wrong one.

A fixed mask may also be a sequence of parameter *names* (e.g. `["tau1", "t0"]`), in which case the driver matches by name and the rest are free.

6.10 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.

6.11 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.

Parameters resting on a finite bound at termination are reported separately (`FitReport.at\_bounds`); a bounded optimum is a constraint, not an optimum, and downstream code should treat it as such.

6.12 Results

Three result classes share the common fields of `KineticFit`:

`KineticFit` Single-trace or multi-trace result: lifetimes, amplitudes, residuals, reproducibility payload.

`GlobalFit` Adds nothing structurally (the spectra live in `S`), but is typed distinctly so a function can assert it received a global result rather than a single-trace one.

`TargetFit` Adds `scheme\_key`, `scheme\_label`, `scheme\_text`, the rate matrix `K` and eigenvalues. The eigenvalues are stored because they, not the rates, are what the data actually determine for a branched scheme.

Every result exposes `as\_species\_args()` (the adapter for PyMORGAN’s `plot\_species\_spectra`), `spectra\_kind` (`DAS / EAS / SAS`), `artefact\_spectra` (the coherent-artefact columns, if any), `summary()` and `log\_summary()`.

6.12.1 LDA result

LDAResult carries the density map S_map of shape $(N_p \times M)$, the lifetime grid tau_grid of length M , the chosen α , the L-curve diagnostics, optional bootstrap errors, and a list of detected peaks. It also carries `citations()` and `format\_citations()`, which enumerate the references for whichever algorithm combination was used, and `to\_pdat(path)`, which exports the 2D map to PyMORGAN's PDAT format for further plotting.

6.13 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_ta.io`, which handles fit sessions — parameters, masks, results — and not raw spectroscopy formats.

6.14 Fit session files

`pr.save\_fit(path, fit)` writes a `.prfit` file: a compressed `.npz` archive holding the arrays (spectra, concentration profiles, residuals, axes) and the reproducibility payload as a JSON string inside it. Keeping them together means a saved fit cannot lose its provenance.

`pr.load\_fit(path)` reopens a session as a `LoadedFit`: the arrays as attributes, the JSON payload as `.meta`. It is deliberately *not* a live `KineticFit` — a reopened session has no dataset behind it and cannot be refitted as it stands, and returning something that looks like a fresh result would invite exactly that. Everything needed to set the fit up again is in `.meta`. A target fit reopens with its `scheme\_text`, so the scheme is editable without re-entering it.

```
path = pr.save_fit("scan_fit", fit) # -> scan_fit.prfit
prev = pr.load_fit("scan_fit.prfit")
print(prev.summary()) # same format as a live result
print(prev.taus) # numpy arrays as attributes
```

The format version is embedded in every file and checked on load: a file written by a newer PyRATE-TA that the current build cannot interpret raises rather than silently mis-reading it.

6.15 Steady-state spectra and pure positive species recovery (GSB and SE)

In time-resolved transient absorption, the observed differential absorbance $\Delta A(t, \lambda)$ is a linear superposition of three distinct photophysical contributions:

$$\Delta A(t, \lambda) = S_{\text{ESA}}(t, \lambda) - a(t) \sigma_A(\lambda) - b(t) F(\lambda), \quad (10)$$

where:

- $S_{\text{ESA}}(t, \lambda) \geq 0$ is the genuine excited-state absorption of transient intermediates, which is strictly non-negative.

- $\sigma_A(\lambda) \geq 0$ is the steady-state ground-state absorption spectrum. When molecules are promoted to the excited state, ground-state depletion produces a negative *Ground State Bleach* (GSB) feature with an amplitude $a(t) \geq 0$ proportional to the fraction of depleted ground state.
- $F(\lambda) \geq 0$ is the steady-state fluorescence emission spectrum. Optical de-excitation induced by the probe pulse produces negative *Stimulated Emission* (SE) with an amplitude $b(t) \geq 0$ matching the fluorescence profile.

Because GSB and SE introduce negative dips that overlap with positive ESA bands, raw evolution-associated spectra (EAS) or species-associated spectra (SAS) often appear distorted or drop below zero, complicating spectral identification and molar extinction coefficient estimation.

Mathematical recovery of pure positive spectra. When steady-state absorption (σ_A) and/or emission (F) spectra are provided, PyRATE-TA extracts pure positive species absorption spectra $S_{\text{abs}}(\lambda) \geq 0$ using constrained non-negative least squares (NNLS) implemented in `pyrate_ta.fit.groundstate`:

$$\min_{a,b \geq 0} \int_{\lambda \in \Lambda_{\text{neg}}} |S_{\text{diff}}(\lambda) + a \sigma_A(\lambda) + b F(\lambda)|^2 d\lambda, \quad (11)$$

subject to the physical constraint:

$$S_{\text{abs}}(\lambda) = S_{\text{diff}}(\lambda) + a \sigma_A(\lambda) + b F(\lambda) \geq 0 \quad \forall \lambda. \quad (12)$$

Here Λ_{neg} denotes the spectral regions where the uncorrected difference spectrum S_{diff} drops below zero or exhibits prominent bleach valleys. Active-region quantile scaling guarantees that the recovery cancels negative bleach bands without artificially elevating or distorting positive ESA peaks where GSB is absent.

Loading reference spectra. Steady-state absorption and emission files can be loaded programmatically through `pyrate_ta.io.reference` or interactively via the GUI. Reference spectra are accepted in standard two-column ASCII formats (`.csv`, `.txt`, `.dat`), where lines beginning with `#` or `%` are treated as comments, column 1 contains wavelength coordinates in nanometres, and column 2 contains absorbance or emission intensity. Reference spectra are automatically resampled onto the transient dataset’s probe wavelength axis using monotonic cubic spline interpolation.

Interactive visualization. Both the embedded species preview panel and the detached **Plot DAS/EAS/SAS** popout window include a toggle checkbox:

☒ **Recover pure positive spectra (GSB / SE)**

When enabled, the plots switch instantly from differential spectra to recovered positive absorption spectra $S_{\text{abs}}(\lambda)$. In addition, the fitted ground-state bleach contribution ($-a \sigma_A$) and stimulated emission contribution ($-b F$) are plotted as dashed reference baselines, providing immediate visual confirmation of the de-bleaching quality.

6.16 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-ta-gui` (or `python -m pyrate_ta.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 and two lamps — one for the dataset, one for the per-point noise; there is no format selector, since a fit needs a PDAT and its `.pdats` sibling is picked up automatically), *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). The algorithm defaults to Trust Region Reflective: lifetimes are bounded below by zero, and MINPACK's Levenberg-Marquardt cannot honour bounds at all, so choosing it with finite bounds is refused rather than silently ignoring them.

Fit control At the foot of the left column: the data / fit / residuals view selector, *Preview*, *FIT DATA*, *Reset*, the statistic read-out, *Copy taus*, and the two post-fit figures — *Plot DAS* (named after the model: DAS, EAS or SAS) and *Plot Conc. Profile*. Both open in their own window, and both are opened automatically when a fit finishes; each has its own switch (`plot\_species\_after\_fit`, `plot\_profiles\_after\_fit`) in the `[gui]` section of `settings.toml`. A preview does not trigger them: it is a look at a guess, not a result.

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`.

The legends of these panels are always drawn *inside* their axes, whatever `[plot] legend\_location` says in PyMORGAN's settings: three panels share one figure, so a legend parked beside an axis lands on its neighbour or off the canvas. The setting still governs every stand-alone figure PyRATE-TA opens — pop-out cuts, species spectra, concentration profiles — where beside the axis is PyMORGAN's default and usual look.

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

The embedded figure sets its margins explicitly (`[plots] panel\_left`, `panel\_right`, `panel\_top`, `panel\_bottom` and the two gaps) rather than calling `tight\_layout`: it shares a canvas with the controls panel, and the automatic layout leaves the axes noticeably smaller than the window allows.

The four *Additional plots and cuts* buttons open stand-alone PyMORGAN figures. The two cut plots ask which cuts to draw, pre-filled with the crosshair's position: the crosshair picks one cut, which is what the embedded panels show, whereas a separate figure is where several are compared. The prompt is PyMORGAN's (`gui.dialogs.ask\_values`), so the syntax — numbers, ranges such as 1900:5:1950, or `all` — is the same as in PyMORGAN's own cut plots.

7.2 Fitting on a small screen

The window opens at 940×900 and is clipped at start-up to the *available* screen area — which already excludes the menu bar and the dock — then centred, so a geometry saved on a large monitor cannot reopen half under the dock on a laptop.

Being able to *ask* for a small window is not enough: a layout whose minimum size hint exceeds the screen simply refuses to shrink. The left column therefore lives in a scroll area, so the window's minimum height is set by the plot canvas alone and the controls stay reachable by scrolling when there is not room for all of them at once.

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.

7.3 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.

Show kinetic graph, beside it, draws the scheme of whatever model is currently selected — compartments, arrows and rate *symbols* ($k_1, k_2, \dots$, not their values) — in its own window. It works before any fit, from the lifetimes as typed, so a scheme can be checked while it is being built; after a fit it draws the fitted one.

7.4 The LDA tab

A separate *LDA* tab gives access to Lifetime Density Analysis (section 6.6) without switching away from the same dataset. Its controls map directly to the keyword arguments of `solve\_lda`:

Grid Number of log-spaced lifetime points and the $[\tau_{\min}, \tau_{\max}]$ range (auto-filled from the delay axis).

Regularisation Penalty type (`ridge` / `d1` / `d2`) and the alpha method (`lcurve` / `gcv` / `morozov` / `fixed`). When *fixed* is selected an alpha entry box appears.

Options Non-negative constraint, SVD pre-filtering (number of components), coherent-artefact suppression, and bootstrap iterations.

Run LDA Executes `solve\_lda` headlessly and populates the results table and the lifetime-density map.

After a successful run the tab shows: the integrated amplitude spectrum $A(\tau) = \sum_p |S_p(\tau)|$, the L-curve or GCV score plot (for diagnosing the alpha choice), and a 2D heatmap $S(\tau, \lambda)$. Detected peaks are listed in a table with their centroid lifetimes. The result can be exported to a `.pdat` file with *Export map...* for further plotting in PyMORGAN.

7.5 Reading a fit

Two figures carry the result, and both are opened automatically when the fit converges (buttons reopen them at will):

Species spectra The amplitude spectrum of each component, drawn by PyMORGAN's `plot\_species\_spectra` through the result's `as\_species\_args()`. Their name follows the model, and so does the button: DAS for parallel, EAS for sequential, SAS for target. The distinction is not cosmetic — it is what the spectra may be interpreted as.

When steady-state absorption or emission reference files are loaded, both the embedded preview and the detached species popout provide a **Recover pure positive spectra (GSB / SE)** checkbox. Enabling it de-bleaches the species spectra in real time via constrained NNLS, displaying the

reconstructed positive absorption bands $S_{\text{abs}}(\lambda) \geq 0$ alongside the fitted bleach and stimulated emission reference baselines (§6.15).

Concentration profiles The populations $C(t)$ of the model itself: one of the few views PyMORGAN has no concept of, since they are never measured. Component i carries the same colour as its spectrum, and the delay axis comes from PyMORGAN's `apply\_delay\_axis`, so the profiles can be laid beside the contour and read on the same time axis. For a sequential or target model this is where the plot earns its keep: it shows *when* each species is present, and therefore which spectrum can be trusted where.

7.6 Software updates

PyRATE-TA shares PyMORGAN's automated update checking and deployment architecture:

- **Passive background notification:** At launch, PyRATE-TA queries GitHub releases in the background (at most once every 24 hours). When a new release is available, a discreet status bar notification appears.
- **On-demand check:** The **About** dialog (**Help** → **About PyRATE-TA**) provides a **Check for Updates...** button to inspect release notes and query the repository directly.
- **Direct installation:** When launched from a Windows standalone suite installation, users can download the latest installer directly within the application and execute the upgrade seamlessly.

7.7 Layout is declared, not built

All widgets, layouts, containers and controls are declared in the Qt Designer file `src/pyrate_ta/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-ta-edit-gui # or: python -m pyrate_ta.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.

7.8 Structure of the code

`MainWindow` is assembled from per-tab mixins in `gui/tabs/`:

DataTabMixin Loading, dataset display, the embedded data / fit / residuals view and the additional cut plots.

ModelTabMixin Model selection, the lifetime table, bounds, fixed flags, IRF and time-zero controls.

FittingTabMixin Fit control: run, preview, reset, statistic read-out, copy taus, auto-plots.

LDATabMixin The LDA controls and result display.

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`.

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.

7.9 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.

7.10 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_ta` 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, colourmap, time-axis convention, figure sizes — belongs to `pymorgan.Settings` and is read through `pm.get\_settings()` after `pm.apply\_style()`. PyRATE-TA'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>time_limits</code>	[0.2, None]	fit	Default fitting window [min, max] delays
<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- σ 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 that appear in the settings dialog. LDA parameters are not listed here; they are set via the GUI LDA tab or directly in `settings.toml` for headless use (Table 7).

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

Field	Default	Meaning
lda_n_lifetimes	100	Grid points in the lifetime axis
lda_tau_min	None	Grid lower bound; None = auto
lda_tau_max	None	Grid upper bound; None = auto
lda_regularisation	L-curve	Alpha selection: L-curve / GCV / Fixed
lda_alpha	10^{-3}	Strength used when lda_regularisation is Fixed
lda_non_negative	False	Constrain distribution to ≥ 0

Table 7: LDA default settings, stored in the [lda] section of `settings.toml`. These do *not* appear in the settings dialog; the GUI LDA tab reads them at start-up to pre-populate its widgets, which are then the single control surface for an interactive session.

`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()` drives the GUI settings panel: label, kind (choice / float / int / bool / text), optional choices / min / max / step / tooltip, and tab. The method is *derived* rather than hand-written: every dataclass field gets an entry automatically, with the widget kind read from its annotation, the page from `field\_sections()` and the tooltip from `field\_comments()`. 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. A small override table refines labels and ranges, and can hide a deprecated field.

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] (LDA defaults for headless and scripted use), [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.

The [lda] block and the two control surfaces. LDA has two ways to set its parameters:

- **Interactive:** the GUI LDA tab — dropdowns, spinboxes and checkboxes. These are the canonical control surface during a session.
- **Headless / scripted:** `settings.toml` under [lda]. Keyword arguments to `solve\_lda` override everything for a single call.

The [lda] block does *not* appear in the settings dialog, because that would create two places to set the same parameters. Instead, the GUI reads `lda\_*` fields from the active `Settings` once at start-up and loads them into the tab’s widgets (`LDATabMixin.\_init\_lda\_tab\_from\_settings`), so editing `settings.toml` changes the defaults the GUI sees the next time it opens.

The same precedence applies for scripted use:

```
# defaults from settings.toml
result = pr.fit.lda.solve_lda(data)

# or override explicitly --- keyword args always win
result = pr.fit.lda.solve_lda(
    data,
    n_taus=200,
    alpha="auto",
    alpha_method="gcv",
    non_negative=True,
```

)

A flat file — the older layout — is still read, so an existing configuration is never orphaned.

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_ta 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-ta-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.

The same editor is reachable from the main window under *View* → *Settings...*, in a dialog with two halves: *Analysis* (PyRATE-TA) and *Aesthetics* (PyMORGAN). Both panels edit their own package's settings live — the embedded plots re-render as values change — and *Save permanently* writes each half back to *its own* `settings.toml`. PyMORGAN's file is located from the installed package rather than from the working directory: when PyRATE-TA is the host, the `settings.toml` in the working directory is PyRATE-TA's, and writing PyMORGAN's fields into it would overwrite the wrong file.

8.6 Both files are loaded

The GUI loads PyRATE-TA's `settings.toml` and PyMORGAN's. Without the second, PyMORGAN's settings stayed at their dataclass defaults, so a profile or a `font\_scale` chosen in PyMORGAN was ignored here and the same data was drawn with a different font stack — which is what makes a PyRATE-TA figure look unlike a PyMORGAN one while both claim the same style. PyMORGAN's file is located from the installed package, never from the working directory, for the reason given above.

Fonts themselves are PyMORGAN's (Helvetica, TeX Gyre Heros, Arial, from its `.mplstyle`) and PyRATE-TA applies them unchanged, but the *installation* is per environment: a PyRATE-TA virtual environment that has never run `pymorgan-install-fonts` falls back to DejaVu Sans. The GUI therefore checks at start-up and says so, with the command to run.

8.7 How results are quoted

Two settings in `[plots]` govern every lifetime PyRATE-TA prints:

`show\_uncertainties` Whether the $\pm$ appears at all, in the plot legends and the logged fit summary.

`round\_uncertainties` Round a value *together with* its uncertainty: the uncertainty to one significant figure and the value to *that same decimal place*, so 325.3 ± 2.5 is quoted as 325 ± 3 , 297.5 ± 1.38 as 298 ± 1 and 8.036 ± 0.0234 as 8.04 ± 0.02 . This keeps a fitted constant from being presented with more precision than its own uncertainty supports. Halves round *up* ($2.5 \rightarrow 3$): an uncertainty should never be rounded down.

Both are applied at the point of display only; the result object keeps the full precision, so a saved session and the copied lifetimes are unaffected.

9 Extending PyRATE-TA

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. `lda.py` is the regularised LDA solver; `groundstate.py` converts difference spectra to absolute ones; `constrained.py` provides the sign-mask amplitude solver.

`results/` Result dataclasses and their serialisation, carrying the reproducibility payload of section 6.13. `lda\_result.py` is the companion to `fit/lda.py`.

`plot/` Only what PyMORGAN cannot express: residual matrices, concentration profiles $c(t)$, K / compartment diagrams, lifetime-density maps, fit-monitor bars, kinetic traces with residuals, and spectra with residuals. 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-ta-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 Adding a plot

New analysis-only figures go in `plot/` as a new module, exported through `plot/\_\_init\_\_.py`. Every plotter reads aesthetics from `pymorgan.get\_settings()` so the figures match PyMORGAN's. The module must not import Qt; the GUI imports the plotter, not the other way around.

9.6 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(\dots, 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.setex=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`.

10 License

PyRATE-TA 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.