Supplementary Information

pyglotaran: a lego-like Python framework for global and target analysis of time resolved spectra

Ivo H.M. van Stokkum[a], Jörn Weißenborn[a], Sebastian Weigand[a,b], Joris J. Snellenburg[a]

[a]Department of Physics and Astronomy and LaserLaB, Faculty of Science, Vrije Universiteit Amsterdam, De Boelelaan 1081, 1081 HV, Amsterdam, The Netherlands

[b]Institut für Festkörperphysik, Technische Universität Berlin, Hardenbergstr. 36, 10623 Berlin, Germany

The Jupyter notebooks and the preprocessed data can be downloaded from https://github.com/glotaran/pyglotaran-release-paper-supplementary-information/releases, so that the reader can reproduce all results.

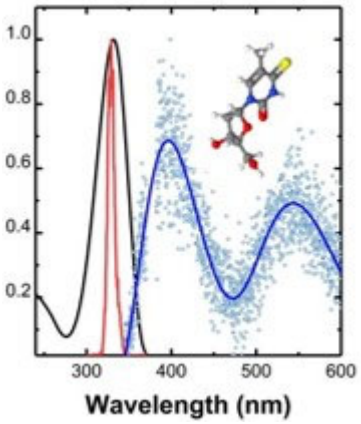


Figure S 1. The structure of 4TT (inset) with its normalized absorption spectrum (black curve), the normalized pump pulse spectrum (red curve), and normalized photoluminescence spectrum (blue curve is the fit and blue dots are the data) obtained pumping the sample at 330 nm. Figure adopted from [1].

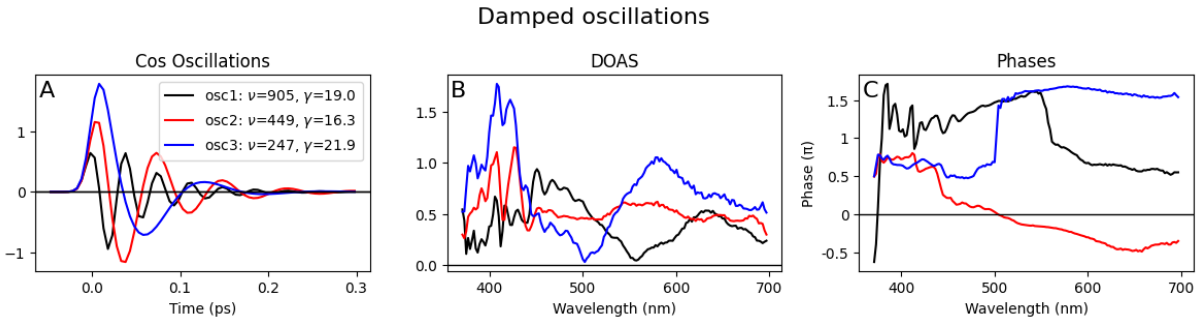


Figure S 2. Overview of the estimated DOAS and phases. (A) Cosine oscillations with frequencies $\bar{\nu}n$ (in /cm) (where n is the DOAS number) and damping rates γ (in 1/ps) written in the legend at the left, using the appropriate color. Scaling of the DOAS is such that the product of the DOAS and the damped oscillation is the contribution to the fit. (B) Estimated DOAS. (C) Estimated phase profiles of the DOAS.

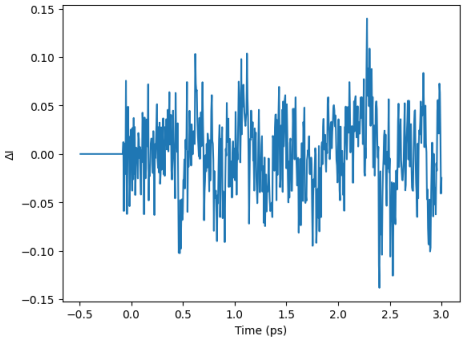


Figure S 3. Estimation of the laser intensity fluctuations responsible for the residual structure in Figure 5A by fitting the residual spectrum against the data. Further details can be found in the **4TT** Jupyter notebook.

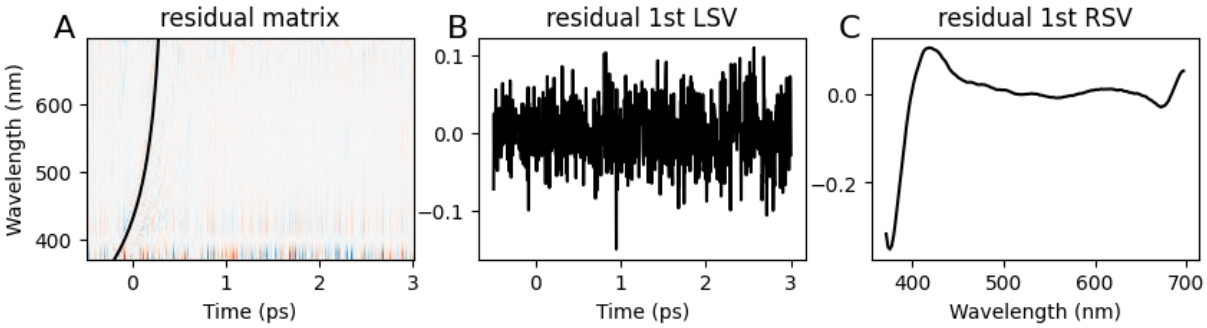
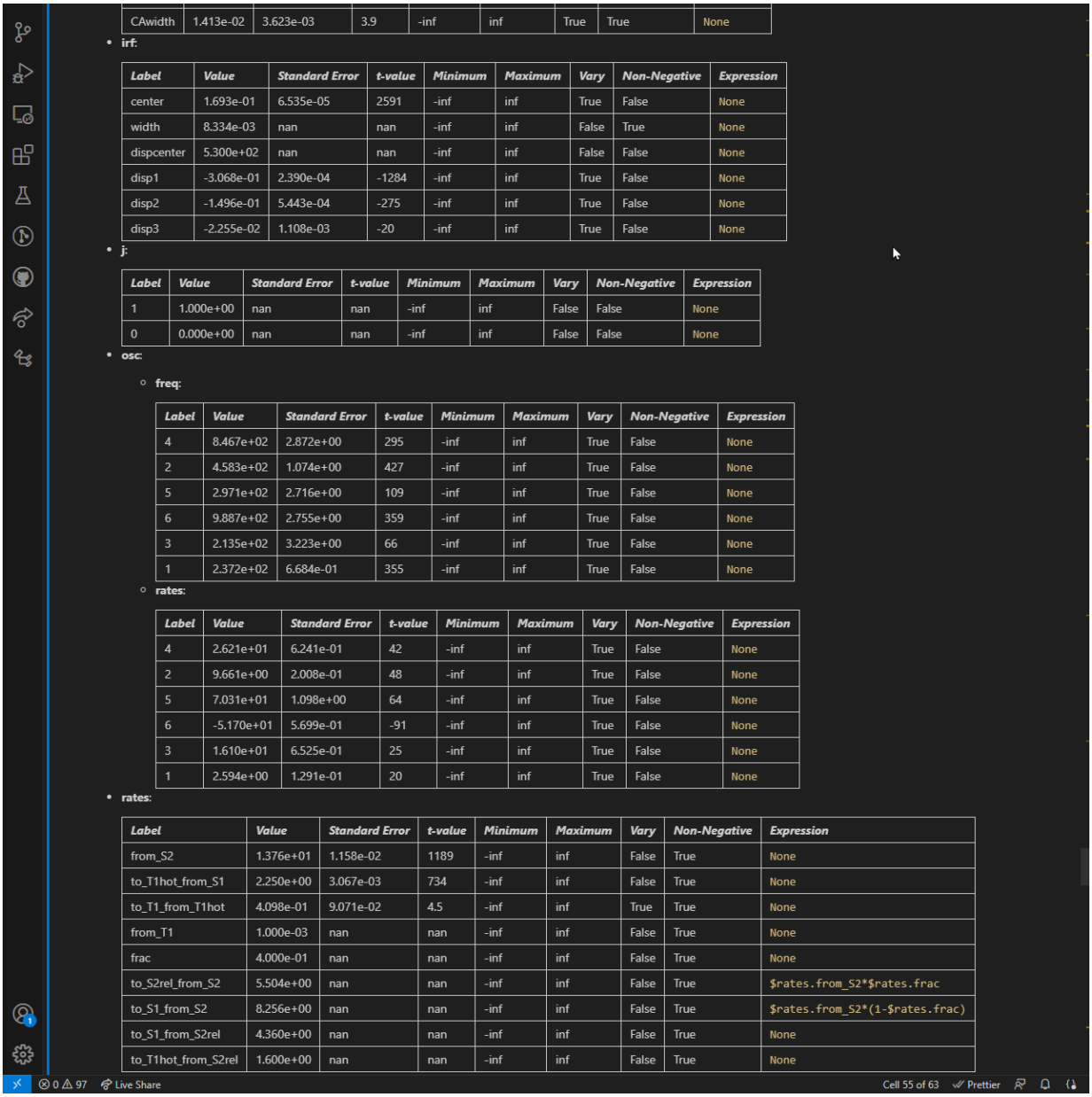


Figure S 4. First left (B) and right (C) singular vectors resulting from the singular value decomposition (SVD) of the residual matrix (A) after correcting the data for the laser intensity fluctuations. The black line in (A) represents the location of the maximum of the IRF, which is described by a third order polynomial function of the wavenumber.



CAwidth	1.413e-02	3.623e-03	3.9	-inf	inf	True	True	None

- **irf:**

Label	*Value*	*Standard Error*	*t-value*	*Minimum*	*Maximum*	*Vary*	*Non-Negative*	*Expression*
center	1.693e-01	6.535e-05	2591	-inf	inf	True	False	None
width	8.334e-03	nan	nan	-inf	inf	False	True	None
dispcenter	5.300e+02	nan	nan	-inf	inf	False	False	None
disp1	-3.068e-01	2.390e-04	-1284	-inf	inf	True	False	None
disp2	-1.496e-01	5.443e-04	-275	-inf	inf	True	False	None
disp3	-2.255e-02	1.108e-03	-20	-inf	inf	True	False	None

- **j:**

Label	*Value*	*Standard Error*	*t-value*	*Minimum*	*Maximum*	*Vary*	*Non-Negative*	*Expression*
1	1.000e+00	nan	nan	-inf	inf	False	False	None
0	0.000e+00	nan	nan	-inf	inf	False	False	None

- **osc:**
  - **freq:**

Label	*Value*	*Standard Error*	*t-value*	*Minimum*	*Maximum*	*Vary*	*Non-Negative*	*Expression*
4	8.467e+02	2.872e+00	295	-inf	inf	True	False	None
2	4.583e+02	1.074e+00	427	-inf	inf	True	False	None
5	2.971e+02	2.716e+00	109	-inf	inf	True	False	None
6	9.887e+02	2.755e+00	359	-inf	inf	True	False	None
3	2.135e+02	3.223e+00	66	-inf	inf	True	False	None
1	2.372e+02	6.684e-01	355	-inf	inf	True	False	None

  - **rates:**

Label	*Value*	*Standard Error*	*t-value*	*Minimum*	*Maximum*	*Vary*	*Non-Negative*	*Expression*
4	2.621e+01	6.241e-01	42	-inf	inf	True	False	None
2	9.661e+00	2.008e-01	48	-inf	inf	True	False	None
5	7.031e+01	1.098e+00	64	-inf	inf	True	False	None
6	-5.170e+01	5.699e-01	-91	-inf	inf	True	False	None
3	1.610e+01	6.525e-01	25	-inf	inf	True	False	None
1	2.594e+00	1.291e-01	20	-inf	inf	True	False	None

- **rates:**

Label	*Value*	*Standard Error*	*t-value*	*Minimum*	*Maximum*	*Vary*	*Non-Negative*	*Expression*
from_S2	1.376e+01	1.158e-02	1189	-inf	inf	False	True	None
to_T1hot_from_S1	2.250e+00	3.067e-03	734	-inf	inf	False	True	None
to_T1_from_T1hot	4.098e-01	9.071e-02	4.5	-inf	inf	True	True	None
from_T1	1.000e-03	nan	nan	-inf	inf	False	True	None
frac	4.000e-01	nan	nan	-inf	inf	False	True	None
to_S2rel_from_S2	5.504e+00	nan	nan	-inf	inf	False	True	$rates.from_S2*$rates.frac
to_S1_from_S2	8.256e+00	nan	nan	-inf	inf	False	True	$rates.from_S2*(1-$rates.frac)
to_S1_from_S2rel	4.360e+00	nan	nan	-inf	inf	False	True	None
to_T1hot_from_S2rel	1.600e+00	nan	nan	-inf	inf	False	True	None

Figure S 5. Summary statistics of the optimized parameters, screenshot from the sequential_doas_4TT.ipynb Jupyter notebook.

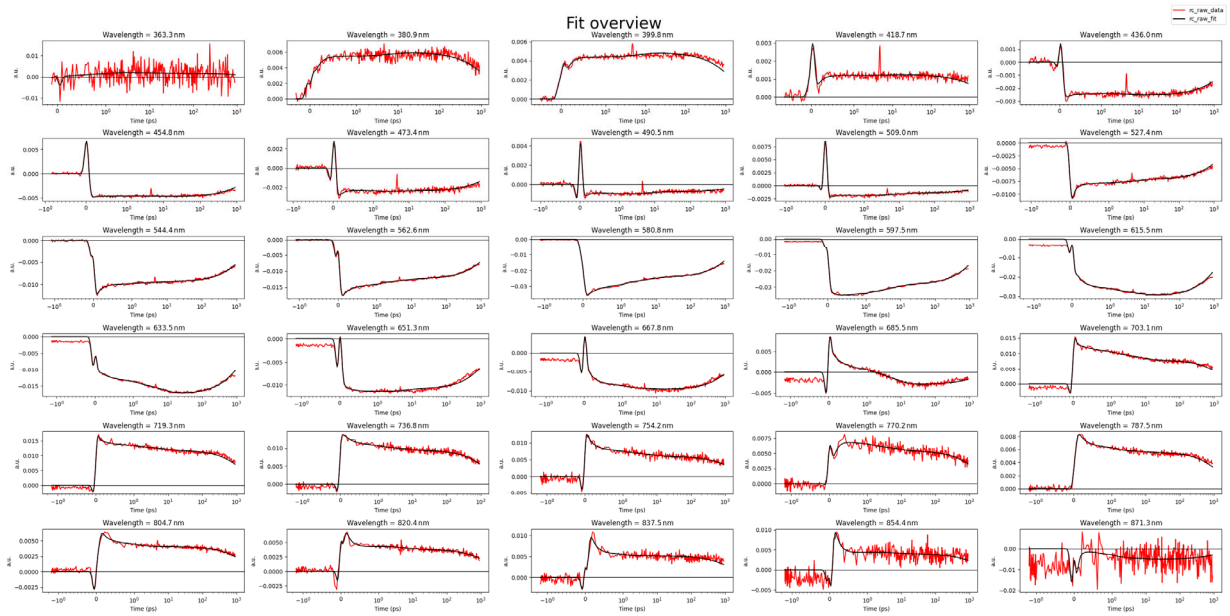


Figure S 6. Selected time traces of raw **rc** data in CH_2Cl_2 after excitation at 530 nm data (in mOD, red) and fit (black). Wavelength is indicated in the title of the panels. Note that the time axis is linear until 1 ps (after the maximum of the IRF), and logarithmic thereafter. Rms error of the fit is 0.59 mOD. Note the presence of the prezero baseline, especially with wavelengths 527, 597 nm, and higher.

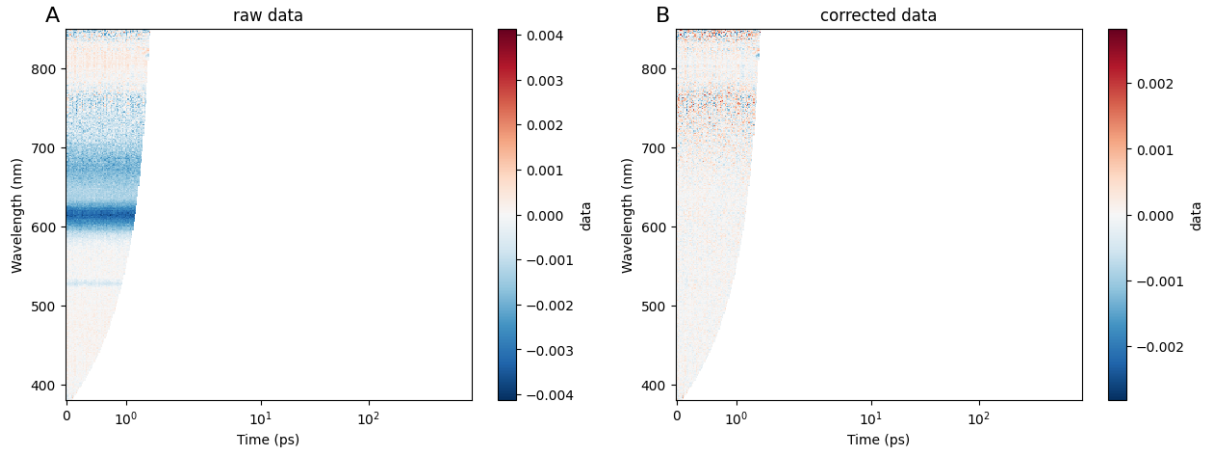


Figure S 7. Selection of the data until 0.2 ps before the center of the IRF, demonstrating the prezero baseline in the raw data (A), and the baseline corrected data (B).

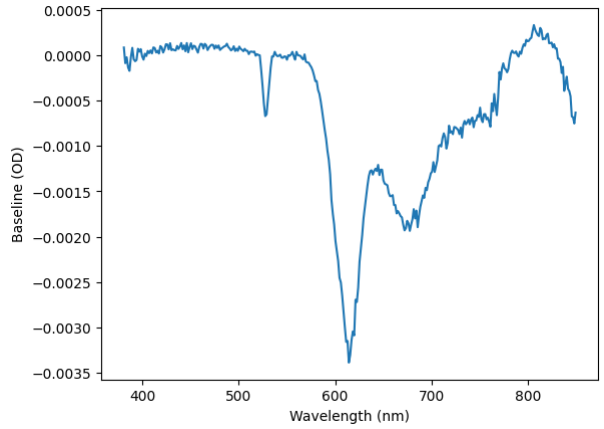


Figure S 8. Estimated prezero baseline.

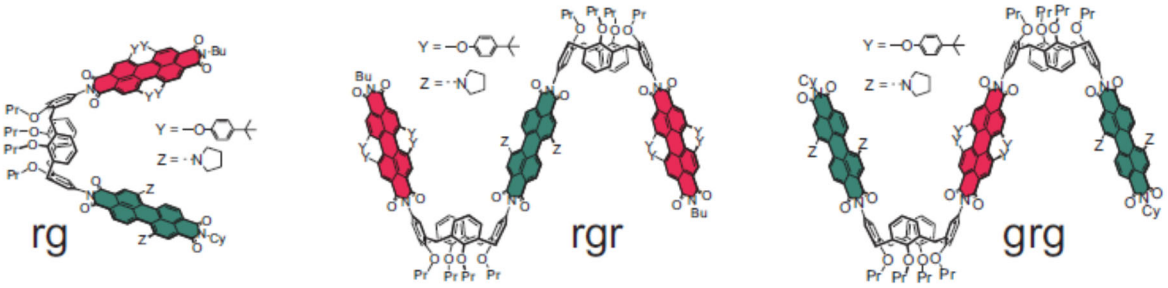


Figure S 9. Chemical structures of the supramolecular systems **rcg**, **rcgcr**, **gcrcg**. Figure adopted from [2]. Note that the **c** is omitted from the labels in the figure.

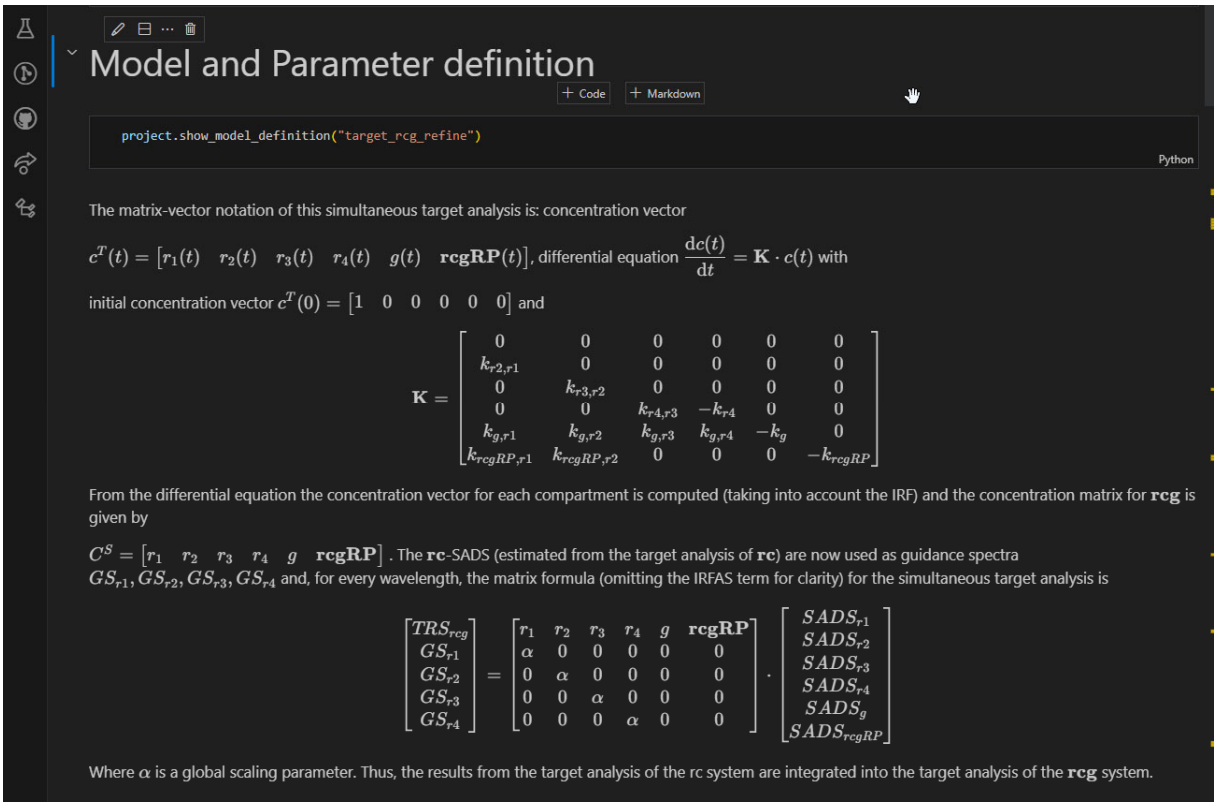


Figure S 10. Formulas for the fitting of the guidance SADS[3], screenshot from the target_rcg_compare_part2.ipynb Jupyter notebook.

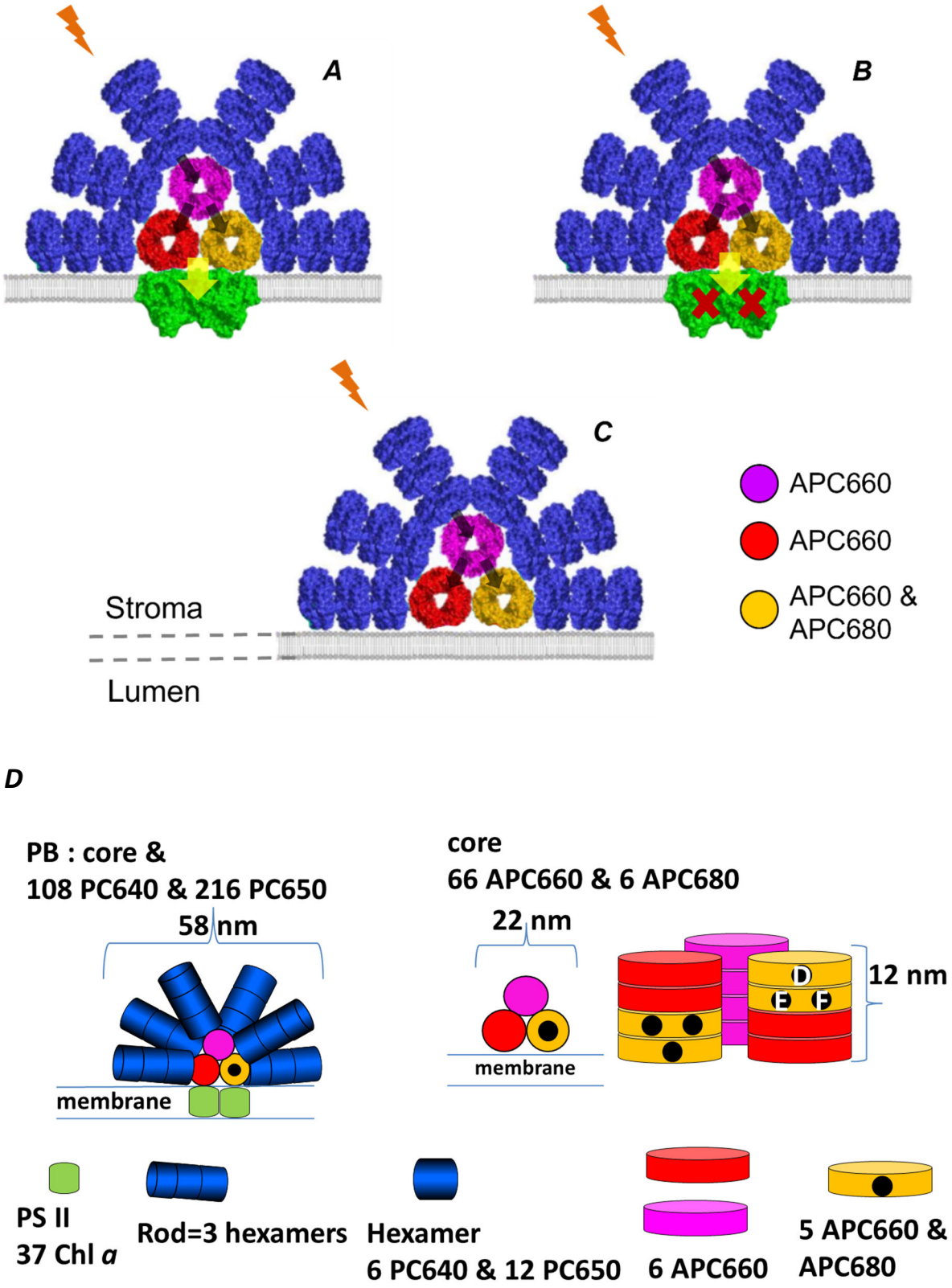


Figure S 11. Cartoon in side-view of possible fluorescent species in the *Synechocystis* ΔPSI mutant. Depicted are a PB-PSII complex with both PSII RCs open (*A*), both closed (*B*), and a PB that is not coupled to any PSII (*C*). *Key:* blue, rods consisting of three hexamers; top and basal core cylinders respectively in magenta, red and orange; green, PS II dimer. Dark arrows represent intra-PB EET; yellow arrows represent EET from the PB core to PSII. An "X" stands for a closed PSII RC. Panel D depicts the location of the different pigments in the structure. The letters D,E,F indicate the three different APC680 pigments. The approximate length for each subunit is based on [4]. Figure adopted from [5].

Modeling language

The core of pyglotaran is the modeling language which is a declarative Domain-Specific Language (DSL) that is designed to describe the behavior of systems in terms of their states and how they interact with one another, in a modular and composable manner. The DSL is split up into two parts, the parameter definitions and the model definitions referencing parameters from the parameter definitions by their name. Using the DSL pyglotaran functions as an engine that interprets the model and parameter definitions and applies them to fit the data.

Expressive user defined names

Traditionally Problem Solving Environment (PSE) like the pyglotaran predecessors Glotaran [6] and TIMP [7] use an index based description for models and parameters definitions. While this is very close to the mathematical description and implementation using matrixes and vectors, it puts a lot of mental load on the user who needs to keep track of what which index means and constantly translate back and forth between an index based description and the kinetic model describing the same physical system.

To reduce the mental load for users and simplify the translation between the kinetic model and its description pyglotaran allows and encourages to use meaningful and verbose names both in the model as well as in the parameter definitions (Figure S 12).

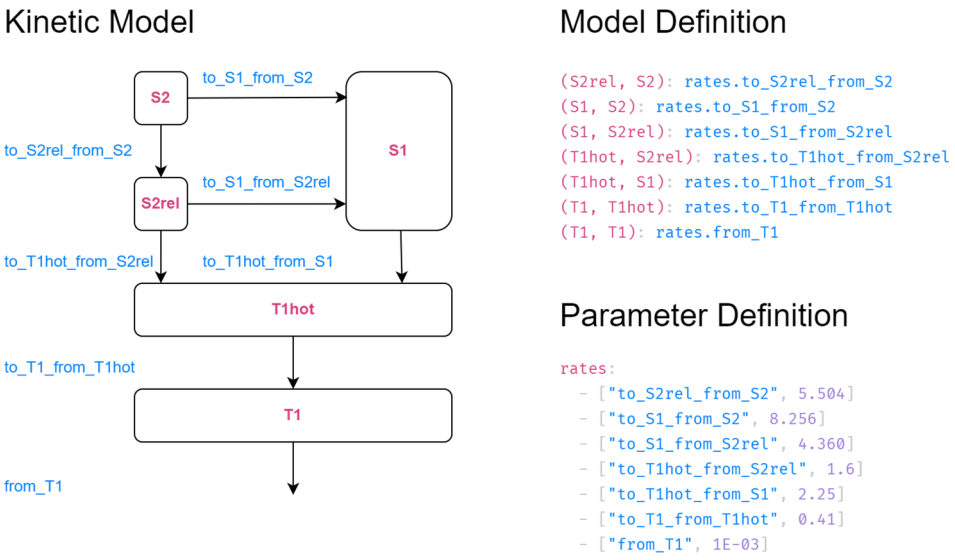


Figure S 12. Schematic illustrating the relation between the kinetic model (see also Figure 6) and the corresponding parts of pyglotaran model and parameter definitions for the 4TT case study.

Model definition structure

Since pyglotaran and its DSL are under active development and considering that the DSL can be extended via the pyglotaran plugin system it is more expedient to explain the core concepts in an abstract manner than to discuss the concrete syntax of the current model definition implementation.

In its core the model definition consists of two main parts: the model element definitions and the dataset definitions (Figure S 12). The model element definitions define reusable elements which can be combined for each dataset in the dataset definitions describing the data measured in that dataset. The parts of the model definition mostly follow a nested mapping (key-value pairs) pattern where the keys alternate between keywords defining the functionality and free user defined names which provide the meaning. This groups functionally similar parts together, with the most top-level key always being a keyword. Keywords can be categorized into required and optional keywords, where required keywords are mandatory for an element to provide its base functionality and optional keywords can further customize the functionality.

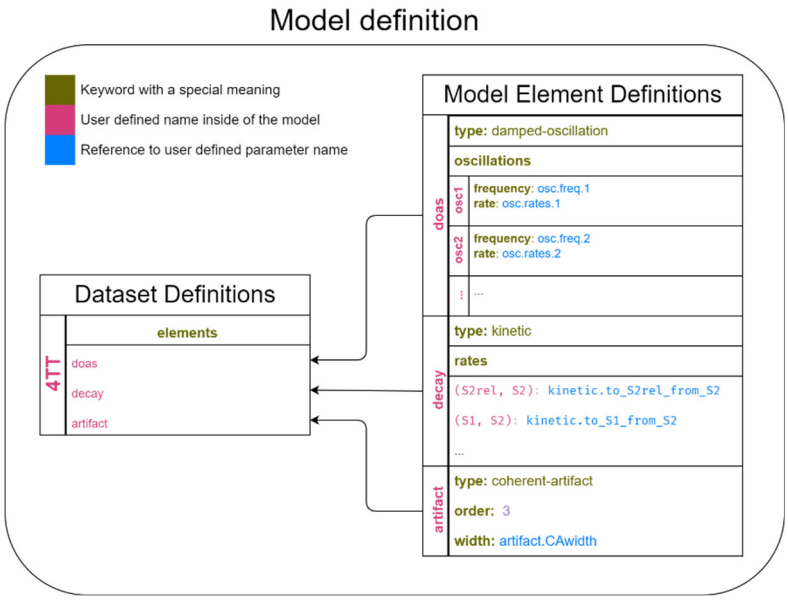


Figure S 13. Abstract schematic of the model definition structure illustrating the relation between model element definitions and dataset definitions as well as the nested alternating pattern of keywords and user defined names and references to parameter names.

The most notable required keyword of model elements is the *type* which defines the overall functionality of the element, and which other keywords can be used with it. While the value for the *type* is always a string, what values can be used for it depends on the installed plugins that define the value by which they are referred to in their implementation. Pyglotaran already comes with *built-in* model element plugins for the most common analysis needs in time-resolved spectroscopy which don't require an additional installation and can be used out-of-the-box.

What structure the value for a keyword key should have depends on the *type* of the element itself. The value of *oscillation* key in the *doas* element (Figure S 13) for example expects a nested mapping where the keys are user defined names for oscillations and the value is a mapping with the keywords *frequency* and *rates* as keys and the values being references to parameter names in the parameter definition for the corresponding values. While the value of *rates* key in the *decay* element expects a mapping where the keys have the form (*<user-defined-compartment-name>*, *<other-user-defined-compartment-name>*) in to-from-notation and the values reference the name of the corresponding transition rate parameter. Whereas the keys in the *artifact* element don't use a mapping at all but expect an integer value between 1 and 3 in case of the *order* key and a reference to the name of a parameter in case of the *width* key.

The reference implementation of the file format used for the model definition uses the YAML markup language which is designed to be intuitive and easy to read, using indentation to indicate nesting and a minimal use of punctuation. But which file formats are supported can easily be extended using the plugin system (described below) for model file reading and writing.

Parameter definition

The parameter definition connects the name a parameter is referred to with the value which will be used as the starting value in the optimization process and allows to define additional options for the parameter. The naming of the parameter is free to the user's desire with very few restrictions like that it cannot be a reserved keyword in the python programming language and that only ASCII characters are allowed. For a better structuring the parameter naming allows the creation of groups to bunch up parameters that contextually belong together making it easier to focus on those, while the group names provide the verbosity to determine what the intended usage of the parameter is. When

referring to a parameter the group it belongs to become part of its name with a dot (.) denoting the parent child relationship between parent group and child.

Two input styles for the parameter definition are supported: a nested input style using the YAML markup language and a flattened table like style input using the format CSV, xlsx or ods. The advantage in using the nested YAML syntax is that groups of parameters which belong contextually together are also visually grouped due to the indentation based YAML syntax, options can be applied to a whole group in a single line and the short name (name inside of a group) can be used which simplifies visual differentiation between parameters. The advantage of the table style is that it uses the full name of the parameter including the groups which makes it easier to look it up in the model definition and search for it.

If a parameter should not be changed during optimization the *vary* option can be set to *false*, which will exclude this parameter from the optimization. This is useful for parameters like the initial input to the compartments, to manually help the optimizer to get out of a local minimum when it gets stuck or to speed up optimization when focusing to improve a selected part of the model.

To set boundaries for the parameters that their values should not pass the *minimum* and *maximum* options can be used when using an algorithm in the optimization that supports boundaries.

The most powerful feature of the parameter definition is the usage of an *expression* allowing to define relations between parameters using equations. This can be used to enforce prior knowledge of a systems behavior, reducing the number of free parameters since this parameter is implicitly set to not be varied which consequently also improves the optimization.

Structure of the result and relation to the model definition

Optimization results generated by pyglotaran can be categorized into overall results for the whole analysis and per dataset results in the *data* attribute (Figure S 13). Each dataset result can be further categorized into a general section and one or more element dependent sections which depend upon the elements used in the dataset definition of that particular dataset inside the model definition.

The result for the overall analysis contains the optimized parameters, optimization history and the parameter history for the whole analysis, as well as optimization metrics like the degrees of freedom and root mean square error.

The per dataset results can be accessed by the same names that the datasets were referred to in the model definition. The general part consists of information which is independent of the elements used in the model definition such as the original data, the fitted data, and the residual. In addition, each model element used in the model definition for a dataset also adds information specific to the type of the element. The decay component of type kinetic for example adds information about the species concentration and species associated spectra (SAS). The names of the saved information are in general only dependent on the type of an element and not on the freely chosen name, an exception to the rule are elements whose type can occur multiple times per dataset in which case the name is suffixed with the name of the element in the model definition (e.g., a_matrix_decay for the 4TT example in Figure S 14).

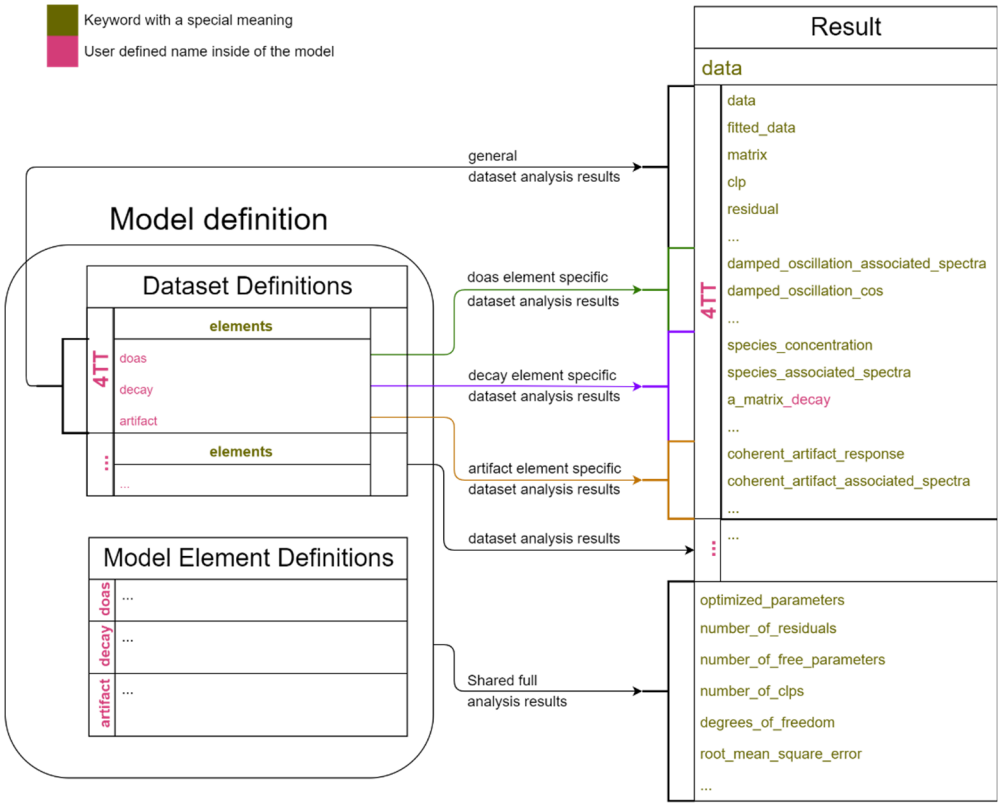


Figure S 14. Schematic of the general structure of a result object and its relation to the model definition. The information saved in the result can be categorized into overall results for the whole analysis and per dataset results. The per dataset results use the same name as the dataset in the model definition and can be categorized into general information independent of model elements and element specific information. The per dataset results include some redundant information, which could be calculated from other saved values (e.g., $fitted\_data = data - residual$). This is done for the convenience of the user, who can use the redundant information to plot the results in external plotting software without the need to compute those values and can be configured using *SavingOptions*.

Plugin system

Pyglotaran uses a plugin system for model elements and for reading and writing files. While the builtin plugins already provide the functionality needed by most users, the plugin system provides additional modularity, flexibility, extendibility, and the possibility for seamless integration with other software. The plugins can be created by third-party developers or by the users themselves and can provide new features and capabilities that are not included in pyglotaran itself. The file io plugins are split into two categories the ProjectIo plugins for the model, parameter definitions, and results and the DataIo plugins to read and write data. Pyglotaran already allows for many different input and output data formats (e.g. plain ASCII and NetCDF). To illustrate the use case, creating a DataIo plugin allows one to read and/or write new file formats without the need for an additional conversion step. Whereas creating a result saving plugin allows for integration with other software. By registering a file io plugin with the extension of a file format, pyglotaran is able to automatically determine which plugin to use when reading or writing a given file (on https://pyglotaran.readthedocs.io/, search for: *write own plugin*). To ensure that the right plugin is used when multiple plugins are registered under the same name the plugins are also registered with their fully qualified name (python import path) and the user can override which plugin to use (on https://pyglotaran.readthedocs.io/, search for: *using plugins*). Creating a model element plugin allows researchers to add new analysis capabilities as well as the ability to share the details of their research with the wider scientific community, allowing for reproducibility, easier collaboration, and improving scientific progress.

Software development

To ensure efficient development and high-quality code, several key practices have been implemented in the development of the pyglotaran ecosystem. Firstly, version control is managed through git and GitHub, using the GitHub flow model and branch protection to manage changes and ensure code quality. This allows multiple developers to collaborate on the codebase simultaneously while maintaining version history and control over changes.

All the development happens in the glotaran organization on GitHub, that besides the new python projects also contains the legacy projects TIMP[7], and Glotaran[6], which are still maintained but not further developed. The most notable components of the pyglotaran ecosystem are pyglotaran-extras [pygta-extras], pyglotaran-examples and pyglotaran-validation which together build the basis for the pyglotaran validation framework.

Code is structured using packages and modules, making it easier to organize and navigate through the codebase. Quality assurance is ensured using linters, formatters, and type checkers, which help to catch errors and enforce consistency in the code.

Continuous Integration and -Delivery (CI/CD) are utilized to automate the building and testing of the software, ensuring that the code is always in a working state. Automated documentation, both generated and manually curated, is also provided to facilitate understanding of the codebase and to help new users get up to speed quickly.

Automated dependency updates ensure that the software remains up to date with the latest libraries/frameworks and potential problems are discovered early. Deployment is handled through PyPI and conda-forge, making it easier for users to install and use the software.

Overall, these practices ensure that pyglotaran is high-quality, well-maintained software that is efficient to develop, test, and use.

Development infrastructure and tooling

Each project in the pyglotaran ecosystem as well as pyglotaran itself are developed using professional state of the art software development technologies and practices, as well as an extensive set of different tests that need to be passed by each change as well as code reviews from the maintainer before it can be added to the project. The passing of those tests is enforced by using branch protection with requires each change to pass all CI/CD tests (see below) including the QA-tools (see below) and have at least one approving review.

Development workflow using GitHub

GitHub is a web-based platform that provides version control using git and collaboration features for software development projects. It allows developers to store their code repositories in the cloud, provides version control features to track changes to code over time, and includes collaboration features like managing issues, pull requests (PR), and code reviews. Additionally, it offers Continuous Integration and Continuous Delivery to automate building, testing, and deploying of code changes.

Development follows the GitHub branching model where each developer only works on their fork of the main repository and submits PRs to the main repository to get a change incorporated. After the changes are reviewed and all tests are passed these changes get squash merged into the main branch of the repository resulting in a clean linear history, where details of the changes and discussions can be looked up in the PR itself. The details on how to contribute to a project can be looked up in the contribution section of each project's documentation.

Quality assurance tools

Quality assurance tools (QA-tools) are an essential part of modern software development, and they play a critical role in improving the quality and maintainability of the code. In the Python development world, several quality assurance tools are widely used, such as linters, formatters, and type checkers.

Linters are tools that analyze code for potential errors, bugs, and style issues. They can identify common mistakes like syntax errors, undefined variables, or unused imports. Linters can enforce coding conventions and best practices, ensuring that code is consistent and easy to read. This type of tool can save developers significant amounts of time by catching issues early in the development cycle.

Formatters, on the other hand, are tools that help developers to enforce a consistent code style. They can help ensure that the code adheres to indentation, line length, and other formatting rules. In addition to enforcing consistency, formatters can also automatically fix common formatting issues, such as incorrect whitespace or inconsistent line breaks. The result is code that is easier to read and maintain.

Type checkers are another critical quality assurance tool in Python. They help ensure that code adheres to the specified types of variables, arguments, and return values. By catching type-related errors before runtime, type checkers can help developers avoid bugs and improve code quality. Type checkers can also improve code understanding by helping developers understand the flow of data and logic in their code.

By using quality assurance tools like linters, formatters, and type checkers, Python developers can ensure that their code is consistent, error-free, and adheres to best practices and conventions. This, in turn, can help improve the reliability, maintainability, and readability of code, and reduce the time and effort required for testing and debugging. Ultimately, the use of quality assurance tools helps to improve the overall quality of software development, making it more efficient and effective.

The pyglotaran development uses the pre-commit framework to manage the quality assurance tools and their versions. This ensures that each committed code change is up to standards by intercepting the commit if any of the checks fails, leading to a cleaner commit history and an easier review process.

Continuous-Integration and Continuous-Delivery

Continuous-Integration and -Delivery (CI/CD) is a set of software development practices that aim to improve the speed and quality of software delivery by automating the process of building, testing, and deploying software.

Continuous Integration (CI) is the process of regularly integrating code changes into a shared code repository, typically using tools like Git, and ensuring that the codebase is always in a working state. This involves running automated tests and checks on the codebase to catch errors and issues early on.

Continuous Delivery (CD) is a practice where developers automate the process of deploying code changes to production. With Continuous Delivery, every code change is automatically built, tested, and prepared for deployment. Once the code changes have passed all tests and quality checks, they are automatically deployed to a staging or testing environment. The code changes can then be manually promoted to the production environment later.

In the context of software builds and distributions, CI/CD involves setting up automated pipelines that build, test, and package software changes, and then automatically deploy them to target environments. This ensures that software changes are thoroughly tested and validated before they are released to end-users and helps to reduce the time and effort required to deliver software updates.

Software Documentation

The documentation of the projects consists of handwritten guides and API documentation generated from the source code using Sphinx. To autogenerate documentation for the source code Sphinx uses the *docstring* added by the developers which are also shown in editors to guide users on how to use each part of the software and explain what the configuration options and intended usage are. The documentation for the current development version and each release is hosted on Read the Docs, allowing online access to the documentation websites as well as downloads as PDF or epub.

Dependency updates

In contrast to classical compiled desktop applications that come with a fixed set of dependencies, python packages typically only define a minimal version requirement for dependencies to allow interoperability of multiple packages in an environment. On the one hand this allows users to use different packages without their requirements conflicting and breaking functionality. On the other hand, this makes it nondeterministic for developers which specific versions of dependencies the users have installed, and new releases of a dependency can possibly break functionality. Therefore, the development is done with the dependencies pinned to specific versions creating a deterministic and reproducible reference environment. It also allows to automate dependency updates using the update service *dependabot* which is integrated into github. Dependabot will create a pull request for each new release of a dependency which triggers the automated tests and ensures that the software works properly with the new version of the dependency or allows developers to disallow this specific version when installing the software and file a bug report/make create a fix for that dependency.

Release and Deployment

Releases are done using GitHub releases on the corresponding project which allow to have a rich description of the release itself and its changes, as well as automatically creating a git tag for the release which allows to download the source code of the project for that particular release. Creating the git tag then triggers the CI/CD workflow, runs all tests again and in the case of a package builds source and binary distributions of the package that are uploaded to Python Package Index (PyPI). The release on PyPI then gets picked up by the conda-forge bot which will create a PR to the corresponding conda-forge feedstock repository, which contains the build recipe to create a release on conda-forge. This allows for additional manual adjustments of the build recipe (e.g., changed dependencies) and ensures that the conda build is also tested before it is released.

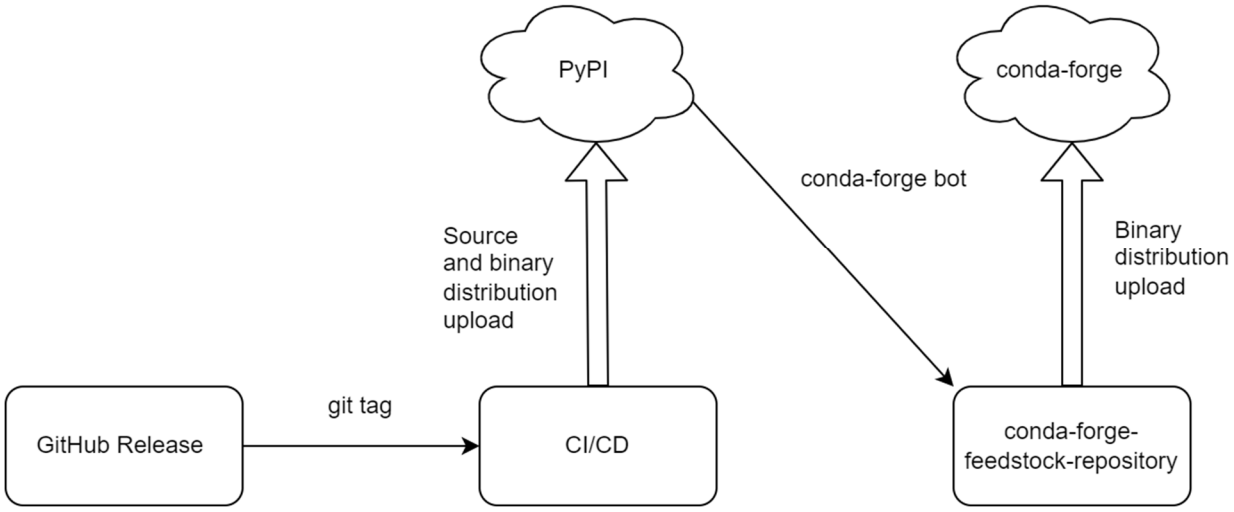


Figure S 15. Schematic of the automated release process of python packages. When creating a release on github a git tag is created which triggers the CI/CD workflow to run test and upload source and binary distributions to PyPI. The release on PyPI is then used to create a pull request to the corresponding feedstock repository updating the conda build recipe, test the created build and create a release to conda-forge.

Deprecation warnings

To allow improvements of pyglotaran as well as its DSL without breaking functionality for users pyglotaran uses deprecation warnings. Those warnings provide users with guidance on what changes they need to make to their code and/or model definition to ensure compatibility with the latest version. Users can take proactive steps to update their code, avoiding costly and time-consuming issues that may arise from using deprecated features. The clear and concise explanations of the changes required to ensure compatibility with future versions also allow for a gradual and incremental upgrade path of existing case studies done with older versions of pyglotaran by doing incremental

updates. The deprecation framework of pyglotaran was built for maximum development flexibility and its capabilities range from the renaming of functions, methods, and classes to moving source files to different locations with minimal development effort (see the documentation https://pyglotaran.readthedocs.io/en/latest/contributing.html#deprecations). To ensure that the provided deprecation warning is correct the existence of the new and old usage is verified in self-tests of the deprecation function. When a new release is prepared the deprecation functions raise errors if a deprecation is overdue, which ensures that the deprecated functionality is removed, and the source code is kept clean. A list of new deprecations and removal of deprecated functionality for each version can be found on the release-page (https://github.com/glotaran/pyglotaran/releases) for that version as well as the changelog in the documentation (https://pyglotaran.readthedocs.io/en/latest/changelog.html).

Ecosystem

While pyglotaran functions as the engine allowing to fit complex case studies in a way which wasn't possible before, its ecosystem provides convenience functions for plotting, documentation, and validation of the analysis result.

Pyglotaran-Extras

The pyglotaran-extras [**pygta-extras**] are a collection of very high-level helper functions for plotting and inspecting results from a parameter estimation using pyglotaran with a strong focus on time-resolved spectroscopy. One main purpose is to provide users with a quick and easy way to get feedback on the quality of their analysis without the need of detailed knowledge about the structure of the generated result dataset(s) and the usage of the xarray API [8]. The other purpose is to provide cross version compatibility to inspect result datasets. The plotting functionality focusses on providing a starting point for publication ready figures while allowing for the full customization and flexibility provided by matplotlib [9].

Pyglotaran-Examples

The pyglotaran-examples were historically grown as a development tool to validate results from pyglotaran manually against published and simulated results of its pre-decessors, which was mostly done by comparing plots. It still serves the purpose of validation using pyglotaran-validation and the comparison-results branch which is validated manually each time it is updated (e.g., to accommodate improvements or fix bugs). But now it also serves as usage documentation, demonstrating different features of the pyglotaran model language depending on which system is analyzed.

Validation

In the 0.5.0 release of pyglotaran the manual validation of results and testing of their consistency was extended by automatic tests which were later moved into its own repository pyglotaran-validation. This change allows for a better extendability in the future to cross validate pyglotaran against a wider range of other of software operating in the same domain. As well as decoupling the validation of generated results from the main project reducing the barrier for contributions. The validation itself does not use pyglotaran but operates on the data directly to prevent overlooking or even misinterpreting issues with the data it is supposed to validate due to circular logic.

Validation framework

The pyglotaran projects use GitHub Actions as their CI-CD platform. The automation pipelines are called workflows and are defined as plaintext files in yaml syntax inside the special folder *.github/workflows* inside of the repositories allowing to put them under version control and easily review changes to the infrastructure. Workflows consist of a definition of events on which the workflows should be executed and a definition of jobs that should be run. The jobs themselves define a list of steps to setup reproducible testing/build environments and run the necessary commands. Steps can either run commands in the shell or use predefined *Actions* to simplify more complex tasks and improve the readability of the workflow. Using the so-called *matrix strategy* (Figure S 17) allows to parametrize jobs and reduce duplications to for example run tests across different operating systems and python versions. For a

more dynamic creation of the matrix a custom step that generates the matrix can be used (Figure S 18). Having reproducible environments in which tests and builds are executed is crucial to guarantee that the software behavior is also reproducible and consistent.

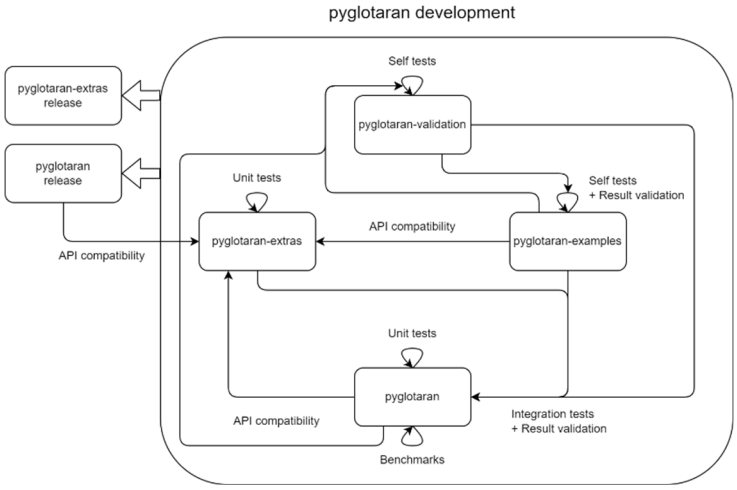


Figure S 16 Schematic overview how the different projects in the pyglotaran ecosystem interact with another using the CI. For example, if a change is to be made to pyglotaran the unit tests and benchmarks contained inside the pyglotaran repository as well as integration tests and result validation using pyglotaran-extras, pyglotaran-examples and pyglotaran-validation are triggered.

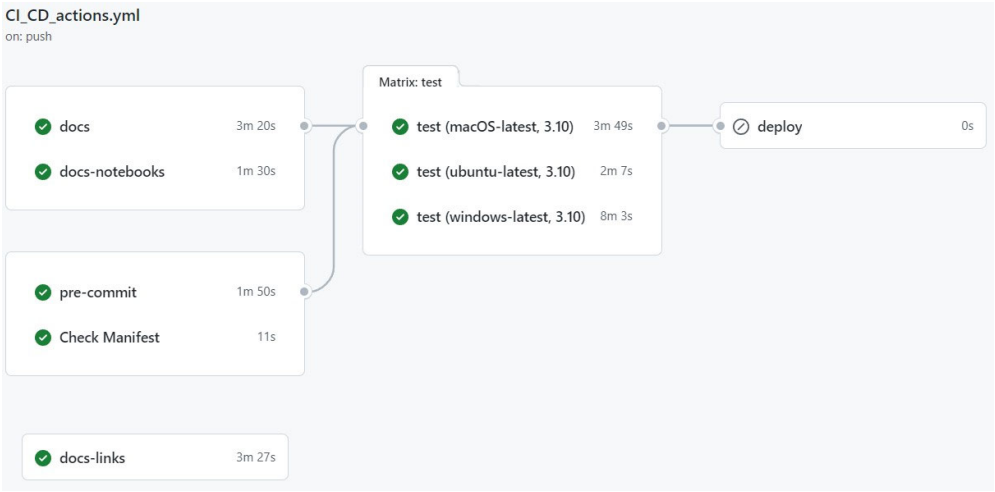


Figure S 17 Schematic illustration of the pyglotaran CI-CD workflow. First the QA-tools and manifest-check are run to ensure that the code is up to standards and all files that should be part of a release are in fact included, as well as tests that the documentation builds properly and does not contain dead links. Only after those tests have passed the unit tests are run for all supported python versions and operating systems. In case of a release all tests but the doc-links tests need to pass before a release on PyPI is created.

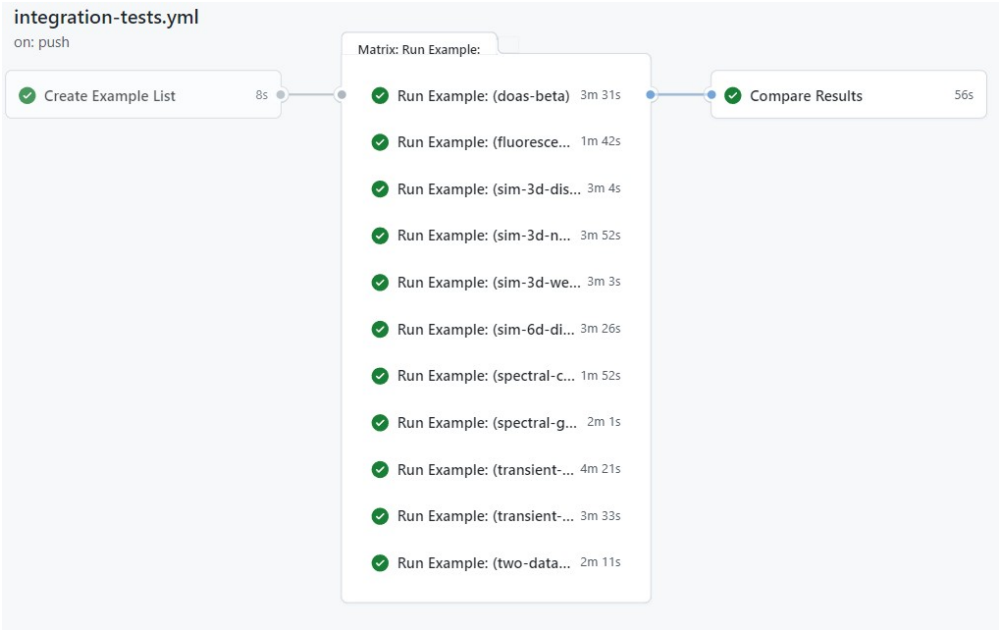


Figure S 18 Schematic illustration of the pyglotaran integration test workflow using github actions defined in the pyglotaran-examples and pyglotaran-validation repositories. The github action of the examples first generates a list of all available examples and then runs them in parallel and uploads the results to an artifact cache, those results are then downloaded and validated against the established and manually validated standards.

References

1. D. C. Teles-Ferreira, I. H. M. van Stokkum, I. Conti, L. Ganzer, C. Manzoni, M. Garavelli, . . . A. M. de Paula, Coherent vibrational modes promote the ultrafast internal conversion and intersystem crossing in thiobases, *Physical Chemistry Chemical Physics*, 2022, **24**, 21750-21758.
2. C. Hippius, PhD Thesis, Universität Würzburg, Fakultät für Chemie und Pharmazie, 2007.
3. I. H. M. van Stokkum, C. Wohlmuth, F. Würthner and R. M. Williams, Energy transfer in supramolecular calix[4]arene—Perylene bisimide dye light harvesting building blocks: Resolving loss processes with simultaneous target analysis, *Journal of Photochemistry and Photobiology*, 2022, **12**, 100154.
4. A. A. Arteni, G. Ajlani and E. J. Boekema, Structural organisation of phycobilisomes from Synechocystis sp strain PCC6803 and their interaction with the membrane, *Biochim. Biophys. Acta*, 2009, **1787**, 272-279.
5. A. M. Acuña, P. Van Alphen, R. Van Grondelle and I. H. M. Van Stokkum, The phycobilisome terminal emitter transfers its energy with a rate of (20 ps)–1 to photosystem II, *Photosynthetica*, 2018, **56**, 265-274.
6. J. J. Snellenburg, S. P. Laptenok, R. Seger, K. M. Mullen and I. H. M. van Stokkum, Glotaran: a Java-based Graphical User Interface for the R-package TIMP, *Journal of Statistical Software*, 2012, **49**, 1-22.
7. K. M. Mullen and I. H. M. van Stokkum, TIMP: An R Package for Modeling Multi-way Spectroscopic Measurements, *Journal of Statistical Software*, 2007, **18**, 1 - 46.
8. S. Hoyer and J. Hamman, xarray: N-D labeled Arrays and Datasets in Python, *Journal of Open Research Software*, 2017, DOI: 10.5334/jors.148.
9. J. D. Hunter, Matplotlib: A 2D graphics environment, *Computing in Science and Engineering*, 2007, **9**, 90--95.