KiMoPack

Release 7.4.9

Jens Uhlig

May 15, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

KiMoPack is a project for the handling of spectral data measure at multiple time-points. The current design is optimised for the use with optical transient absorption data, but it has been successfully adapted for the use with transient x-ray emission and spectro-electro chemistry data.

It focuses on the main tasks an experimentator has Loading and shaping of experiments, plotting of experiments, comparing of experiments, analysing experiments with fast and/or advanced fitting routines and saving/exporting/presenting the results.

The software can be used on several different levels. The simplest level packs everything into an object "TA" that contains all the parameters that are typically set. These objects also contain the typical functions that are used in an analysis. See *Main Tasks overview* for an overview of these functions. All active functions have a capital letter in the beginning.

At the lower levels a series of convenience functions for the efficient plotting of one or two dimensional data is provided. These are typical in the main module

For typical use a series of juypter notebooks are provided that guide through the a number of different use scenarios, and are suggesting the parameter that are typically set.

In addition a series of tutorial notebooks are provided that guide the user through the different functions. These Tutorials can either be downloaded or executed on a "mybinder" server via this badge.

In addition a small series of videos were produced to introduce the features and usage of KiMoPack.

1.1 Links

- Overview talk: I gave a recent overview talk at the LaserLab Europe meeting: https://youtu.be/z9QqVLFWYrs
- Tutorial videos: https://www.youtube.com/channel/UCmhiK0P9wXXjs_PJaitx8BQ
- Documentation: https://kimopack.readthedocs.io/
- PyPI Releases: https://pypi.org/project/KiMoPack/
- Source Code: https://github.com/erdzeichen/KiMoPack
- Issue Tracker: https://github.com/erdzeichen/KiMoPack/issues
- Website: https://www.chemphys.lu.se/research/projects/kimopack/
- Publication: https://pubs.acs.org/doi/10.1021/acs.jpca.2c00907
- Zenodo: https://doi.org/10.5281/zenodo.5720587

The basis of the program is a module called "plot_func.py" that contains all the necessary functions and classes. We recommend to use a package manager to install the program.

Install using "pip":

```
$ pip install KiMoPack
```

Upgrade if already installed:

```
$ pip install KiMoPack -U
```

Install and update using "conda" from the channel erdzeichen:

```
$ conda install -c erdzeichen kimopack
```

Hint: the pip version is usually more recent than the conda version The files can also be downloaded from the github directory https://github.com/erdzeichen/KiMoPack or zenodo (see below)

In general it is a good idea to create a local environment to install files in python if you are using python for many tasks. In a local environment only the packages that are needed are installed, which usually avoids that conflicts can appear. It is very easy to do that.

Under Windows: open the anaconda command prompt or power shell (type anaconda in the windows startmenu) Under Linuxs: open a console

```
$ conda create --name KiMoPack
$ conda activate KiMoPack
$ conda install pytables
```

If you are working with a very old installation it is usually a good idea to also install an updated python

```
$ conda create --name KiMoPack python=3.10 ipython jupyterlab jupyter
$ conda activate KiMoPack
$ conda install pytables
```

into this environment KiMoPack can then be installed. We also recommend (optional) to install python-pptx to create power point slides and nbopen (which allows to automatically open a local server) into the environments. If one of the installs complains (error) that the user does not has sufficient rights, this installation can be done attaching "–user" to the following commands

```
pip install kimopack

pip install python-pptx
pip install nbopen
```

Finally, while still in the environement, activate nbopen. There are different commands for Windows/Linux/Mac. By doing that in the local environment will open and activate the environment. If you left the environement already you can always go back with "conda activate KiMoPack".

```
python -m nbopen.install_win
python3 -m nbopen.install_xdg
Clone the repository and run ./osx-install.sh
```

CHAPTER

TWO

BEST USAGE

While KiMoPack is a python library, we facilitate its use with Jupyter notebooks. For the typical analysis tasks we have developed a series of Notebooks that guide through the tasks.n These notebooks can be downloaded from https://github.com/erdzeichen/KiMoPack/tree/main/Workflow_tools or by command line.

To do that start any console (under windows e.g. type "cmd" and hit enter). In the console you then start python by typing "python" and hit enter, lastly you import Kimopack and run a function that downloads the files for you by typing "import KiMoPack; KiMoPack.download_all()" This downloads the notebooks and tutorials from github for you. If you instead use "import KiMoPack; KiMoPack.download_notebooks()" then only the workflow tools are downloaded. Please copy one of these notebooks into your data analysis folder and rename them to create a analysis log of your session. For more information please see the publication https://doi.org/10.1021/acs.jpca.2c00907, the tutorial videos, or the tutorial notebooks under https://github.com/erdzeichen/KiMoPack/tree/main/Tutorial_Notebooks_for_local_use.

CHAPTER

THREE

WORKFLOW TOOLS

While KiMoPack is a python library, we facilitate its use with Jupyter notebooks. For the typical analysis tasks we have developed a series of Notebooks that guide through the tasks.n These notebooks can be downloaded from https://github.com/erdzeichen/KiMoPack/tree/main/Workflow_tools or by command line.

To do that start any console (under windows e.g. type "cmd" and hit enter). In the console you then start python by typing "python" and hit enter, lastly you import Kimopack and run a function that downloads the files for you by typing "import KiMoPack.plot_func as pf; pf.download_all()" This downloads the notebooks and tutorials from github for you. If you instead use "import KiMoPack.plot_func as pf; pf.download_notebooks()" then only the workflow tools are downloaded. Please copy one of these notebooks into your data analysis folder and rename them to create a analysis log of your session. For more information please see the publication https://doi.org/10.1021/acs.jpca.2c00907, the tutorial videos, or the tutorial notebooks under https://github.com/erdzeichen/KiMoPack/tree/main/Tutorial_Notebooks_for_local_use.

Under windows this would be:

```
$ pip install nbopen
```

$ python -m nbopen.install_win

For more information on the use of these notebooks see the publication https://doi.org/10.1021/acs.jpca.2c00907, the tutorial videos, or the tutorial notebooks under https://github.com/erdzeichen/KiMoPack/tree/main/Tutorial_Notebooks_for_local_use.

CHAPTER

FOUR

LINKS

- Overview talk: I gave a recent overview talk at the LaserLab Europe meeting: https://youtu.be/z9QqVLFWYrs
- Tutorial videos: https://www.youtube.com/channel/UCmhiK0P9wXXjs_PJaitx8BQ
- Documentation: https://kimopack.readthedocs.io/
- PyPI Releases: https://pypi.org/project/KiMoPack/
- Source Code: https://github.com/erdzeichen/KiMoPack
- Issue Tracker: https://github.com/erdzeichen/KiMoPack/issues
- Website: https://www.chemphys.lu.se/research/projects/kimopack/
- Publication: https://pubs.acs.org/doi/10.1021/acs.jpca.2c00907
- Zenodo: https://doi.org/10.5281/zenodo.5720587

CHAPTER

FIVE

MAIN TASKS OVERVIEW

This menu is a shortcut to the main function and tasks that are performed during an analysis. In general one opens one or multiple data Files and after defining a number of shaping parameter, that e.g. set the axis limits or correct the arrival time of different wavelength, plots various graphs. Different measurements or Fits can be compared and the results saved in various forms.

- *Opening of data*
  - Open single file: `pf.TA()`
  - Open many files: `pf.GUI_open()`
  - Combine many scans: `pf.Summarize_scans()`
- *Shaping of Data*
  - Background correction: `self.Background()`
  - Filter bad data: `self.Filter_data()`
  - Correct arrival time (Chirp) `self.Cor_Chirp()`
  - *Data shaping settings that affect the fits*
  - *Plot shaping options without influence on the fitting*
- *Plotting functions*
  - Plotting non Fitted data: `self.Plot_RAW()`
  - Plotting Fitted data: `self.Plot_fit_output()`
  - Adjust fonts in plots: `pf.changefonts()`
- *Fitting, Parameter optimization and Error estimation*
  - Fitting data: `self.Fit_Global()`
- *Comparative plotting*
  - Compare spectra: `self.Compare_at_time()`
  - Compare kinetics: `self.Compare_at_wave()`
  - Compare calculated spectra (SAS or DAS): `self.Compare_DAC()`
- *Data Export and Project Saving*
  - Copy project `self.Copy()`
  - Save Project as hdf5 `self.Save_project()`
  - Save Plots `self.Save_Plots()`

- Save Plots as Powerpoint *self.Save_Powerpoint()*
- Save/export data as ascii/text *self.Save_data()*

CHAPTER

SIX

OPENING OF DATA

A key challenge in using a non graphical programming software is to locate and open files. This tool provides a mixed interface to solve this challenge and in general allows three different ways to import data.

Each of the following function has a "Gui" keyword that can trigger a standard file opening dialogue. Alternatively the filenames can be written together with an (optional) path argument. If the analysis uses the from us provided workflow notebooks, then we suggest that a fresh notebook is used for each particular analysis and that the notebook is copied close to the data.

Three different pathways of importing data are offered:

1. All import functions provide a wide variety of options to adapt for data formats. If a particular option is missing and is desired, please contact the developers via email or raise an issue on github. We will consider to do so but most likely instead provide you with a function for option 2. The formats of the example file have the spectral information as the first row, the time vector as first entrance of each of the following rows and are separated by tab. Files of this type can be read without any further adaption (using the standard parameter). typical options include the transposing of columns, the conversion of time and energy vectors or the providing of external files for energy and times.
2. All import function have the option of providing an external import function. (from version 7.8.0 onwards) this function gets the filename and should return a dataframe to KiMoPack. We provide a function library that contains the formats of befriended groups. If you would like help to develop an import function then please contact the developers.
3. The two main import function have a "ds" parameter to which a Pandas DataFrame can be given. Thus the user might simply import and shape the file and then hand it over to KiMoPack.

6.1 Opening single file and creating TA object

Open single file: *pf.TA()*

Typical use of the tool is based upon an object containing all parameter, functions and the data. This object is created by importing a data file using this function.

The filename can be either provided as a string, with the (optional) path to other folders given. Using the keyword "gui" instead of a filename opens the graphical file selection interface. The function can either open a text style format (using any file ending) or the internally used "hdf5" file format. The latter is exclusively used as internal storage formate that stores the complete project including the RAW data.

After import of either filetype the missing parameter in the "TA" object are set with the *self.__make_standard_parameter()* function.

6.2 Opening multiple files

Open many files: *pf.GUI_open()*

Sometimes multiple files are to be opened. Typical use examples are the options to compare different measurements or analysis runs. This function provides a convenient way to create a list of opened projects. One can

- open a gui and select multiple saved projects, which are returned as a list
- given a list of project names to open them
- open all files in a given folder

The general behavior is selected by the first parameter (project_list) For more details see the examples in *pf.GUI_open()*

6.3 Opening and handling single scans

Combine many scans: *pf.Summarize_scans()*

Typically the experiments consists of a larger number of scans that are combined into a single experimental file. The function "Summarize_scans" reads, selects and eventually combines a series of single scans with a bunch of useful options. The essential idea is that for each scan one or two numbers are generated through integration of the intensity in a temporal and spectral window. This single number is plotted as function against the scan number. Then either a list of numbers or a GUI is used to select the scans that are removed from the average. A number of opening and selection options are given.

Observe the new automatic filter options of summarize_scans

This function could also be used to combine a number of different experiments.

CHAPTER

SEVEN

SHAPING OF DATA

In the Following sections we discuss the parameter and values that are used to filter and shape measured data. In general all loaded data read is stored in the un-altered "ta.ori_ds". A second matrix for the same data is created named ta.ds.

Only the function "Filter_data" works on the dataset ds_ori and ds. The chirp correction creates a new ds from ds_ori. The background correction is applied to ta.ds (unless a specific matrix is given´). All other parameter are only applied during a function and do not alter ta.ds. That means that in each of the plotting/fitting functions a local copy of the ta.ds is created (using the function sub_ds) to which all the shaping is applied.

The intended work flow is:

1. loading of data *Opening of data*
2. Filtering of data *Bad data Filter*
3. (optional) chirp correction *Arrival time correction*
4. (optional) Background correction *Background subtraction*
5. setting of parameter for fit *Data shaping settings that affect the fits*
6. setting of parameter for plot *Plot shaping options without influence on the fitting*
7. all the plotting/fitting. *Plotting functions* and *Fitting, Parameter optimization and Error estimation*
8. saving/exporting *Data Export and Project Saving*

The point 5 and 6 (Parameter) can be easily changed many times and a new plot/fit generated. Important to not is that the parameter are stored with the object. This means that a parameter that is explicitly set, will stay until it is overwritten or the object is fresh loaded. So if e.g. commenting out a certain parameter does not return the value to its Default "None". The Parameter needs to be set explicitly to:

```
ta.intensity_range = 3e-3
#ta.intensity_range = 3e-3 #no effect
ta.intensity_range = None
```

Often it is faster to reload the object by choosing "run all above" in the Notebook.

7.1 Bad data Filter

Filter bad data: *self.Filter_data()*

In some cases there are bad data points or other strange things. NA values will normally be replaced by 0 during import and all data is converted into floats during import. In many recording software (including Pascher instruments) a specific **value** is used to indicate that something went wrong. This function filters everything bigger than this value as error. Real "NaN" values are filtered during the import of Data. There is the option to either drop the times that contain bad values or to replace bad values with a specific value. There is the option to put a uppervalue, lowervalue or a single value that is then used for upper and (as negative) for the lower value.

If the filtering does not work, a manual way of filtering is ta.ds (the chirp corrected data) or ta.ds_ori[ta.ds_ori>20]=0 is the classical way to filter

7.2 Arrival time correction

Correct arrival time (Chirp) *self.Cor_Chirp()*

Cor_Chirp is a powerful Function to correct for a different arrival times of different wavelength (sometimes call chirp).

In general if a file is opened for the first time this function is opening a plot and allows the user to select a number of points, which are then approximated with a 4th order polynomial and finally to select a point that is declared as time zero. The observed window as well as the intensities and the colour map can be chosen to enable a good correction. Here a fast iterating colour scheme such as "prism" is often a good choice. In all of the selections a left click selects, a right click removes the last point and a middle click (sometime appreviated by clicking left and right together) finishes the selection. If no middle click exists, the process automatically ends after max_points (40 preset).

Note that scattercut, bordercut and intensity_range can be used to adjust intensity.

After the first run the polynom is stored in self.fitcoeff, a new matrix calculated from self.ds_ori that is stored as self.ds and a file stored in the same location as the original data. The second time the function *Cor_Chirp* is run the function will find the file and apply the chirp correction automatically.

If one does want to re-run the chirp correction the function *Man_Chirp* does not look for this file, but creates after finishing a new file.

Alternatively the polynom or a filename can be given that load a chirp correction (e.g. from a different run with the same sample). The function *Cor_Chirp* selects in the order:

1. "fitcoeff"
2. "other files"
3. "stored_file"
4. call Man_Chirp (clicking by hand)

Correct arrival time (Chirp) *self.Cor_Chirp()* Manual overwrite arrival time correction *self.Man_Chirp()*

7.3 Background subtraction

Background correction: *self.Background()*

This tool is one of two ways to remove a flat background from the data (typically seen before t=0). This tool averages for each measured wavelength separately the values from 'lowlimit' to 'uplimit' and subtracts it from the data. The low and uplimit can be set anywhere to substract any background. (so one could e.g. substract a product instead) It is important to note that many problems during measurements might be visible in the data before time zero. So I recommend to first plot without background correction and only after this inspection apply the background correction. The fit function has its own way to calculcate and apply a background That could be used instead (but making the fit less stable)

7.4 Data shaping settings that affect the fits

in general the data is handled in each of the plotting/fitting functions separately. In each function a copy of the matrix with the limitation below is created. A number of parameter cut and potentially rebin the raw measured data and as such affet the fit. The typical workflow would therefore be to adjust these parameter before the fitting stage using the RAW plotted fits as a feedback.

- Cut the outside limits of the spectrum: *Bordercut*
- Blank one or multiple regions in the spectrum (e.g. suppress scatter) *Scattercut*
- Cut the outside of the time axis: *timelimits*
- Blank one or multiple temporal regions (e.g. around t=0) *ignore_time_region*
- rebin the temporal axis (useful for e.g. steady state long term UV-vis data) *time_bin*
- rebin the spectral axis (useful for prism based spectrometer) *wave_nm_bin*

For further details and examples see: *self.__make_standard_parameter()* or e.g. the general plotting function *pf.plot_raw()*.

The parameter that only change the plots are discussed in *Plot shaping options without influence on the fitting*

CHAPTER

EIGHT

PLOTTING FUNCTIONS

- Plotting non Fitted data: *self.Plot_RAW()*
- Plotting Fitted data: *self.Plot_fit_output()*
- Interative Plotting RAW and Fitted: `self.Plot_Interactive()`
- Adjust fonts in plots: *pf.changefonts()*

One core function of this tool is to create plots for evaluation and publication. Internally there are a number of separate functions that create each plot type (see below). The methods Plot_RAW and Plot_fit_output wrap the parameter into the object and simplify their use. Two additional functions provide additional features. Both "Save_Plots" and "Save_Powerpoint" are calling both plot functions and dump their output into separate figure files or two slides of a power point file.

Common to both plotting function is that either a single plot can be called by giving the plotting parameter or a series of plots (Default) by giving a list of number with e.g. "range(3)".

Most of the plot defining parameter (like, what for which wavelength the kinetic is plotted or at what times the kinetics are extracted are defined by the :ref:´Plot shaping options without influence on the fitting´.

8.1 Plot_RAW

self.Plot_RAW() plots all raw figures. The different figures can be called separately or with a list of plots (standard) e.g. plotting=range(4) call plots 0-3, plotting=1 a single plot. The plots have the following numbers: 0 - Matrix, 1 - Kinetics, 2 - Spectra, 3 - SVD. The plotting can take all parameter from the "ta" object. See: *self.Plot_RAW()*

8.2 Plot_fit_output

self.Plot_fit_output() plots the fit results. For this is uses the data contained in the shaped and cut datasets that were used for the fit, including all rebinning or temporal restrictions. The figures can be called separately or with a list of plots (standard) The plotting function takes all parameter from the object.

self.Plot_fit_output()

Contents of the plots

1. DAC contains the assigned spectra for each component of the fit. For a modelling with independent exponential decays this corresponds to the "Decay Associated Spectra" (DAS). For all other models this contains the "Species Associated Spectra" (SAS). According to the model the separate spectra are labeled by time (process) or name, if a name is associated in the fitting model.
2. summed intensity. All wavelength of the spectral axis are summed for data and fit.

3. plot kinetics for selected wavelength
4. plot spectra at selected times
5. plots matrix (measured, modelled and error Matrix). The parameter are the same as used for the corresponding RAW plot with the addition of "error_matrix_amplification" which is a scaling factor multiplied onto the error matrix.
6. concentrations. In the progress of the modelling/fitting a matrix is generated that contains the relative concentrations of the species as function of time.

This function is a convenience function and is suppose to be used in conjunction with the object and the embedded parameter (see above). The use of qt as backend allows the easy customization of the plots via the GUI. If the plots are saved as "svg" they can easily be adjusted in inkscape or similar afterwards. For more details see: *self.Plot_fit_output()*

8.3 Plot shaping options without influence on the fitting

In addition to the general shaping parameter from section *Data shaping settings that affect the fits* a number of parameter only affect one or multiple of the plots but not the fitting of the data.

- The plotting of the kinetics is governed by the selection of the wavelength in the list **rel_wave** and the width of each **wavelength_bin**
- The plotting of the spectra is governed by the selection of the timepoint in the list **rel_time** and potentially a percentual binning around this time-point with **time_width_percent**. If this is set to 0 then the measured timepoint is used.
- The intensity (color) in the 2 plots as well as the height of the y-axis is determined by the **intensity_range** parameter that can be set symmetric or a-symmetric for best representation. With **log_scale** This intensity can be scaled logarithmic and **error_matrix_amplification** only amplifies the intensity of the difference matrix (measured-fitted) in the 2d plots
- The color scheme can be set very flexible using the Matplotlib palets, or a manually provided color scheme (e.g. university colors)
- The titles of all plots are chosen either by the filename or can be given flexible in each plotting functions through the title parameter. All the plots can be automatically saved if **save_figures_to_folder** is set to True, Which is useful for fast surveys, otherwise the method *self.Save_Plots()* stores all plots (see *Data Export and Project Saving*). The axis labels are accessible via the **baseunit** and the Fonts are accessible via the function *pf.changefonts()*
- The parameter **equal_energy_bin** can be set to a value which results in that the spectral plots are shown in enqual energy bins. This is useful for tracking vibrations and such. As of version 6.7.1 this is only happing for the RAW plotting.

8.4 interactive Plotting

Interactive plot function that allows the interactive slicing of both time and wavelength. The main parameter of the object apply

8.5 extended Raw plotting

`self.Plot_raw()` is an extended function. All the parameters are accessible (and need then to be set manually). This function also plots a single or multiple plots bzt setting the "plotting" parameter.

There are even more detailed manipulations possible by using the separate plot functions:

- for plotting kinetics at fixed wavelength: *pf.plot1d()*
- for plotting spectra at selected times *pf.plot_time()*
- for plotting the data matrix *pf.plot2d()*
- for plotting the 3 fit data matrix *pf.plot2d_fit()*
- for the SVD plots. *pf.SVD()*

Each of the functions allows to hand in an axis and thus plot multiple things

CHAPTER

NINE

FITTING, PARAMETER OPTIMIZATION AND ERROR ESTIMATION

Fitting data: *self.Fit_Global()*

One of the main functions of this program is to perform Global analysis of one or multiple datasets. The fitting function is in its current implementation embedded in the TA object and uses the parameter control options of an lmfit parameter object as an essential tool. (my thanks to Matthew Newville and colleagues for creating this phantastic tool) [M. Newville, T. Stensitzki, D. B. Allen, A. Ingargiola, 2014. DOI: 10.5281/ZENODO.11813.]. To excecute an Optimization/Fit these essential steps have to be followed (assuming that "ta" is your object):

9.1 Model

you have to define a model by setting ta.mod="internal model name" for one of the internal models, of by setting ta.mod=external_function to an external function. For the internal models three standards are implemented: "exponential", "consecutive" and "full_consecutive". (see below). The external function will receive a vector named "times" and parameter pandas.DataFrame named "pardf". It is expected to return a DataFrame with the times as index and in the columns an expression of the relative concentrations. The external function can reload whatever data is desired. The names of the columns are used in all plots and reports. (see below a description of the example functions defined in "plot_func_function_library.py" and "standard_functions.py")

9.2 Parameter

For handling of Parameters I am using the lmfit Parameter objects to have a flexible and fast Parameter handling toolset. The general steps are: create a Parameter object (or use an existing parameter object) set starting values and (optional) limits, relative expressions and vary = True (if the parameter is to be optimised) (see below for more details)

9.3 Trigger the fitting

To trigger the fitting the function ta.Fit_Global() is called. The fitting function will display its results on the screen and write them into the TA object. first it will create an parameter object ta.par_fit that contains the optimum parameter and can be used later, second it writes the result dictionary ta.re. In this dictionary it adds a number of results and parameters that are useful (see below for further details) A number of error catching routines are build in the fitting software. However it does sometimes (rare) come to crashes. Then please adjust the parameter and or read the error message. Important to notice is that this is a refinement of parameter using an algorithm. Limiting the region where parameter can be and choosing good starting values is essential to obtain reliable results. (see below)

The general Modelling/Fitting process happens with the same approach for internal models as well as for externally provided functions. Other fitting processes like the optimization of the arrival time correction are discussed further down in this document.

1. First a copy of the Data-Matrix ta.ds is created with the shaping parameters
2. Then a Matrix is created that represents the fractional population of each species (or process in case of the paral model). This Matrix contains one entry for each timepoint and represents the kinetic model based upon the starting parameter. (see below for a description of the models). This model formation can by done by using a build in or a user supplied function. (handled in the function "pf.build_c")
3. Then the process/species associated spectra for each of the species is calculated using the linalg.lstsq algorithm from numpy (https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html)
4. From the convoluted calculated species concentrations and spectra a calculated matrix is formed (handled in the function "pf.fill_int")
5. The difference between calculated and measured spectra is calculated, point-wise squared and summed together. (function "err_func")
6. This difference is minimized by iterating 2-4 with changing parameters using an optimization algorithm (generally nelder-mead simplex)
7. Finally in a last run of 2-5 the final spectra are calculated (using the "final" flag) and the optimized parameter, the matrixes ("A"-measured, "AC" - calculated, "AE" - linear error), spectra (always called DAS) the concentrations (called "c") are written in the dictionary "ta.re" together with a few representations and other fit outputs. The optimized parameter are also written into ta.par_fit (as an parameter object) that can be re-used as input into further optimization steps.

The choice to use lmfit parameters allows the flexible handling of constraints and freezing/releasing of parameter. (see below). Additional fitting options include the fitting of the parameter of the arrival time correction, the use of advanced optimization algorithms, and the possibility to fit multiple datasets together.

9.4 Description of models

9.4.1 internal kinetic models

The internal kinetic models are created to be highly flexible. The program recognizes all parameter that are "rates" by recognizing the names with "k" followed by one or multiple digits. The program starts at "k0" and then uses "k1","k2",... in an un-interrupted series. If one key is missing the program stops searching for more. However that offers the fast and easy possibility to add additional components to the fit by adding new parameters with an increasing number. All rates are to be given as rates and are assumed to be in the same units as is the measured matrix. So if the measured data is in picoseconds, the rates are in ps^-1. See section *Data shaping settings that affect the fits* for more information.

A number of extra parameter can be used to change some of the models. to use them add an empty parameter with this name to the fit:

'background' if this keyword is present a flat constant background is created (=1 over the whole time) **'infinite'** if this keyword is present a new non decaying component is formed with the last decay time. **'explicit_GS'** if this keyword is present thenthe ground state (including the bleach) will be added as a explicit component

ta.mod="exponential" In this model the data is represented by independent exponential decays. For each component the a symmetric response function is formed (error function) using the the parameter "resolution" as characteristic width (corresponding to 2 x σ) with with "t0" as the point of 50% rise from 0. At the time 1 x resolution the signal is 97.8%. This rise is convoluted with an exponential decay function per given decay constant. The integral signal of the error function is 1, to which all the decays are set (assuming that they do not decay faster than 2x resolution)

The extracted Spectra are commonly called decay associated spectra (DAS). Their relative intensity is corresponding with the exponential pre-factors for single exponential fits to the data (for each wavelength). Additional parameter that can be set are "background" (assuming a species that is constantly "1") and "infinite" a species that is rising with the response function and then constant "1".

ta.mod="consecutive" This fit assumes a model of consecutive exponential decays. A response function with "t0" = 50% rise is formed that rises symmetric to $2\sigma \approx 98\%$ (of 1) at 1 x the parameter "resolution" followed by A->B->C consecutive decay. This particular model uses a preudo approach to this fit to speed up the calculations. The parameter are optimised by modelling an "exponential" model (see above) followed by a single step of a "true consecutive" decay (see below). This approach is quite representative unless there are fast components of the order of the response function involved in the process and the different processes are clearly separated (each rate one of magnitude separated). Additional parameter that can be set are "background" (assuming a species that is constantly "1") and "infinite" a species that is with the last decay constant to a constant "1" and not decaying.

ta.mod="full_consecutive" This fit assumes a model of consecutive exponential decays. A response function with "t0" = 50% rise is formed that rises to $2\sigma \approx 98\%$ (of 1), at 1x the parameter "resolution" followed by A->B->C consecutive decay. This model is formed by a stepwise integrated differential equation and represents the "true" sequential model. The "rise" is simulated by sampleing a true gaussian function and adding the appropiate fraction to the excited state. Arbitrary pulse/response shapes can be sampled in the advanced functions. Additional parameter that can be set are "background" (assuming a species that is constantly "1") and "infinite" a species that is with the last decay constant to a constant "1" and not decaying.

9.4.2 external kinetic models as defined in example file "plot_func_function_library.py"

External model functions can be written and used to create the matrix of populations. The external function will receive a vector named "times" and a pandas.DataFrame with the columne "value" named "pardf". It is expected to return a DataFrame with the times as index and in the columns the an expression of the relative concentrations. The parameters have a name and a float value. The external function can however load whatever other data is required. We have for example modelled spectro-electro-chemistry data by reading the current from cyclic voltametry and using the value to extract a number representing the concentration of a certain species. An important feature of external functions are that columns in the DataFrame can be labeled with names. These names will be used in the plots and significantly improve the work with complex models. The parameter that are given to the functions can be name arbitrarily but must match:

```
[a-z_][a-z 0-9_]*
```

meaning, it must start with a letter and can only contain (small) letters, numbers and "_". Important is that in order allow for some of the parameter settings (see section :ref:'Setting of Fit parameter') if the first letter is a "k" and the second string is a number the parameter will be interpreted as a rate. (recognition is done by:

```
bool(re.match(re.compile('[k]'), parameter_name[:2])
```

In the file "plot_func_function_library.py" we provide a number of useful example functions that show how to model a variety of different cases. In general there is no restriction to what type of function can be modelled here, but all these examples are based upon transient absorptions spectroscopy. In these examples we model the instrument response by a gaussian pulse. In general, any pulse shape could be loaded from an external file. In these examples we sample the differential change of a concentration by writing the differential for the dynamics. The excitation is then represented by sampling the gaussian pulse and "raising" a certain fraction of the molecules into the excited state. As the gaussian used here is normalized to have the integral of "1", the total initial concentration is "1" and the numbers in this matrix representative with a "fractional population". Each step in the code is documented and the code can be adapted easily to a wide variety of problems. The example functions provided are

- **manual_consecutative** An example how a stepwise consecutative decay would look like.

- **Square_dependence** An example in which the pumping "pulse" is scaled by a parameter and a non linear decay is included (e.g. recombination)
- **gaussian_distribution** A model where a substance is excited into a state, followed by a linear decays step into a state that decays with a distribution of rates (here assumed gaussian) into a final, non decaying state. These type of complex decays are often observed in protein folding

In the file "standard_functions.py" the user can find 16 of the most used functions. For their useage refer to the pdf: "Standard_function_library_image.pdf" to choose the function. Each function has a unique name of type "P21" and is then used as described below.

Usage of external functions: To use an external function, this function needs to be handed to ta.mod. For an external function this means that it has to be imported, and then handed to ta.mod. In the example below we import an external module (the file "plot_func_function_library.py") as "func" and then use from this external module the function "Square_dependence". All the models are extensively documented in the function library. These functions can load any external file with additional information. It is highly recommended to use the versatile parameter setting (see below) to adjust models. E.g. can a certain kinetic pathway be disabled by setting its rate to "0" and using the option "vary=False" to lock it. (see below in the parameter section).

It is highly recommended to use the docstring (description) directly below the definition of the model to describe what it does. This string is stored with ta.Save_Project and should be sufficient to identify the model. Also if all the species are labeled (label the columns of the returning DataFrame) These names are used throughout the plotting functions. (please see examples for more explanation)

Remark: Importing an external function happens in python only if it has not already been imported. So if the fitting function is adapted, either the whole notebook/console needs to be restart, or (better) the function should be reloaded. I recommend to use the function "reload" from the "importlib" for this purpose (see the example below) This should happen before the function is handed to ta.mod (as shown in the workflow notebook).

9.5 Setting of Fit parameter

The fit parameter are a crucial point for achieving meaningful results from an optimization. In general three different types need to be chosen, first the model (see) then if the rate parameter (necessarily call k0, k1, k2, ..) will be handed into the fitting function as they are or in log space. (ta.log_fit) and finally the parameter themselves. The **log_fit** option can be important as it brings widely separated rats into the very similar numerical range, simplifying the function of the simplex optimizer. In this program all rates are limited to be above 0 independent if they are handled linearly or in log. This happens in the begin of the fit function, here all "rates" are identified that have the name "ki" with i =0-99 and then their lower limit is set to zero (unless they have already a lower limit >0).

KiMoPack also uses parameters without values to give specific instructions to the model building and fitting routine. Currently implemented are names like **background** that adds a new background state to be fitted for all timepoints **infinite** that lets the final state be different from the original and **explicit_GS**, which adds the ground state bleach as an explicit component.

The parameter are handled as a lmfit Parameter object. Inside the fitting function this object is converted into a pandas Dataframe that is handed to the function generating the time dependent "concentrations".

- initialize The lmfit parameter object needs to be initialized with "ta.par=lmfit.Parameters()". In the fitting function I convert the parameter object into a DataFrame and back on several places. A function par_to_pardf and pardf_to_par does this conversion. All the parameter set are available through the ta.par object and can thus be given to other fits. After the fit there is a new object calles ta.par_fit that contains the optimized fit results. So if you would like to re-use the old results ta.par=ta.par_fit accomplishes this.
- add parameter Each parameter must have a name from:

```
[a-z_][a-z0-9_]\*
```

(starting with a letter and then letters and stars and "_"). In the included models (see :ref:'Description of models') parameters like "background" and "infinite" trigger the inclusion of e.g. the background or a non decaying component. Other parameters should be initiated with a value that has to be of type float (number). Important to not is that the code automatically recognizes parameter that have the name "ki" with i and element of 0-99 as a rate. These rates are brought in and out of logspace with the switch "log_fit". All other names can be freely chosen. I highly recommend to do this for the sake of structure. e.g. a "threshold" could be named as such

- add or set New parameter are "added" to the parameter object. Existing parameter can be "set" to a certain value. If Set is used any of the additional/optional things can be set alone.
- limits min and max Optional is the settings of limits (**min** and **max**). If a limit is set the parameter will stay within the limit, even if a starting value outside the limit is given! Important to note is that after each optimization that included limits the results should be checked if the limits were reached. (the printed output states the limits). Limits are very important for the more advanced optimizers like AMPGO (see section :ref:trigger-the-fit. The parameter tunneling uses these limits as guidelines.
- Vary=True/False Very useful is the option "vary=True/False". This switch freezes the parameter, or allows it to be optimized by the algorithm. In the progress of an analysis one often freezes a parameter to develop a stable model and releases this parameter later. Particular the parameter "t0" which is in my models the starting point and "resolution", which is in my models the instrument response function are parameter that are often frozen in the beginning. Fitting with them enabled significantly extends the duration for finding a stable fit. Often I first plot the function with the starting parameter, temporarily setting all parameter to vary=False with the trick below, to then step by step enable the optimization, while the starting parameter are adapted.
- expr An advanced option is the setting of expressions. This are relations to other parameter. e.g. expr='k0' sets the value of the current parameter always the same as "k0". The values are always given as string so expr='1-k0' sets the value to 1 - the value of "k0". Please see the documentation of lmfit for further details

Very useful trick to set temporarily set all parameter to vary=False to test e.g. starting conditions and then enable the optimization of a single parameter. As here the "set" is used, the parameter can be initially added with a different value. (see workflow notebook for further examples).

9.6 storing of fit results

- ta.par always contains the initial fit parameter (parameter object)
- ta.par_fit contains the fit results and can be directly re-used with ta.par=ta.par_fit (parameter object)
- ta.re['fit_results_rates'] contains the fit results in a neatly formated DataFrame in the form of rates
- ta.re['fit_results_times'] contains the fit results in a neatly formated DataFrame in the form of decay times (1/rates)
- ta.re['fit_output'] Is the results oject of the fit routine. It can be called and then shows details like number of iterations, chi^2, fit conditions and a lot more. This object is stored after a fit but is NOT saved by ta.Save_Project!

9.7 Trigger the Fit

The Fitting process is triggered by calling the function "Fit_Global". if the parameter were set as part of the object that contains this Fit (as is usually the case with ta.par), than just calling the function without any other parameters is a good choice. Internally the Fit function is making a copy of the parameter and shapes the data, then it optimized single or multiple datasets. As standard it uses the Nelder Mead Simplex algorithm to minimize the error values defined by the function pf.err_func and pf.err_func_multi. Currently the maximum iterations are hard-coded to be max 10000. I have not needed more than 1000 for any well defined problem. The optimizer can be changes to "Ampgo" that offers

an advanced "tunneling" algorithm for checking for global minima. Important for this to work properly all optimizing parameter need "min" and "max" definitions. To use AMPGO set the parameter use_ampgo=True.

Parameter can additionally be given via the parameter and module input at this stage, but in general it is better to define them as part of the ta object.

The pf.err_func and pf.err_func_multi recognise if an internal or an external fitting model is to be used by checking if "ta.mod" (or the here given "mod") are strings or something else (in which case it assumes it is an external function).

See section *external kinetic models as defined in example file "plot_func_function_library.py"* for examples how to define those. The fitting process is in all cases the same. Advanced options include the use of fit_chirp that runs multiple iterations of chirp fitting and global fitting iterative (to a maximum of fit_chirp_iterations), or the multi_project module (see below). In general the dump_paras can be used to write into the working directory a file with the current fitting parameter and the optimum achieved fitting parameter. This is intended for long and slow optimizations to keep a record of the fits even if the fitting process did not finish.

Additional modules from https://lmfit.github.io/lmfit-py/fitting.html can be easily implemented. The string given under **other_optimizers** is handed to the lmfit minimizer and can be used to switch the optimizer. Useful choices are e.g. **least_squares** or similar words. This is particularly useful if the problem does not lend to be solved with nelder-mead. This includes e.g. osciallations.

self.Fit_Global()

9.8 Fitting multiple measured files at once

To fit multiple projects the fit function needs to get a number of projects. These can of course be opened with a hand written loop. A cleaner way is to either use the Gui function to open a list of files. *Opening multiple files* As each file needs a chirp correction and these things I recommend to use saved projects (hdf5 files) for this purpose. Please see the function documentation for further details. In general this function is fitting each of the projects separately, but using the same parameter. This means that in general a new (different) DAS is calculated for each of the measurements.:

```
# Global fitting with multiple files, using a unique parameter (that differs for the␣
↪models)::
```

ta.Fit_Global(multi_project = ta_list, unique_parameter = 'pump_power') ta.Fit_Global(multi_project = ta_list, unique_parameter = 'pump_power', weights = [1/power1,1/power2])

To work with the same DAS for the measured and calculated matrices need to be concatenated before the fitting. This is now implemented and one simply needs to use the switch same_DAS=True:

```
ta.Fit_Global(multi_project=[ta1],same_DAS=True)
```

9.8.1 If you work with the same_DAS=True

In the new version the results of the other datasets are layed into the variable ta.multi_projects (assuming that self=ta) with the current result on position 0 that means:

```
ta.re = ta.multi_projects[1]
ta.Plot_fit_output()
```

plots the other second project:

```
ta.re = ta.multi_projects[0]
```

returns the current results into the usual storage

9.8.2 If you work with the same_DAS=False

then you unfortunately have to calculate the DAS new for each of the different fits. But as the ta.par_fit contains the fits results this is not very difficult and can be done with this code snippet. We assume "project_list" is a list of projects and unique_parameter is a list of unique parameters.:

```
for ta_local in projects_list:
        local_fitted_parameter=ta.par_fit
        try:
                for key in unique_parameter:
                        local_fitted_parameter[key].value=ta_local.par.value
        except:
                pass
        ta_local.par=local_fitted_parameter
        for key in ta_local.par.keys():
                ta_local.par[key].vary=False
        ta_local.Fit_Global()
        ta_local.Plot_Fit_output()
```

9.9 Error Estimation

Estimating errors correctly is based on estimating the validity of the full set of optimized parameter for this we use the F-statistics of the single or combined datasets to define a cutoff value. At the cutoff value the combined Chi^2 is so much larger than the minimum Chi^2 that this can not be explained statistically anymore. Practically this corresponds to making the "Null hypothesis" that all parameters are zero and if the difference of Chi^2 is statistically significant, the coefficients improve the fit

the f-statistics compares the number of

"fitted parameter"=number of species*number of spectral points + number of kinetic parameter "free points"=number of datasets*number of spectral points*number of time points - fitted parameter

within the target quality, meaning, what fraction do my variances need to have, so that I'm 100% * target_quality sure that they are different from zero This is done in the function *plot_func.s2_vs_smin2()*. In this function we use the scipy function

"f_stat"=scipy.stats.f.ppf(q = target_quality, dfn = fitted_parameter, dfd = Free_points) to calculate a scaling factor:

"cut_off_factor"=1+(fitted_parameter*f_stat/Free_points)

The minimum chi^2 multiplied with this scaling factor gives the targeted cutoff at the desired confidence level:

"target_chi_2"=chi_2*cut_off_factor

For each (varied) parameter a separate optimization is performed, that attempts to find the upper and lower bound at which the total error of the re-optimized globally fitted results reaches the by F-statistics defined confidence bound. Careful, this option might run for very long time. Meaning that it typically takes 50 optimization per variable parameter (hard coded limit 200) The confidence level is to be understood that it defines the e.g. 0.65 * 100% area that the parameter with this set of values is within this bounds.

9.10 Iterative Fitting

as the fit results are written into the parameter ta.par_fit the fit can be very conveniently triggered in an iterative fashion. This is particularly useful for refining the chirp. The initially achieved optimal kinetic parameters are used as starting parameter for each global fit after the chirp optimization. e.g. a 5 times iterative improvement can be achieved with:

```
for i in range(5):
        start_error=ta.re['error']
        ta.par=ta.par_fit
        ta.Fit_Global(fit_chirp=True)
        if not ta.re['error'] < start_error:break
```

9.11 Species Spectral Development

A small but often useful function is *pf.Species_Spectra()* It takes either a TA-Object or a concentration and spectral DataFrame and combines the concentration with the species associated spectrum. This leads to the matrix that is contributed by this specific species. As the concentration and the spectrum are combined, this represents the indeed measured signal. Here the ds-parameter of the Plot_RAW function offers a useful combination. assuming that we fitted: species 0,1,2, (or more) then:

```
#extract the spectra
dicten=pf.Species_Spectra(ta)
#plot the measured spectrum and substract the
#contribution of "1" and "2"
ta.Plot_RAW(ds=ta.re['A']-dicten[1]-dicten[2])
```

9.12 Ending the Fit

From version 7.2 onwards we locally import he keyboard module. This modules catches if you press q (for a while) and interrupts the fit. In this case the parameters that are in the project are still the starting parameter! otherwise the fit ends when one of the following conditions are met:

df < tol or the tolerance value that is handed to the optimizer (absolute) for nelder-mead with the name fatol number_of_function_evaluations < maxfev (default 200 * n variables) number_of_iterations < maxiter (default 200 * n variables)

9.13 External Spectra and Guidance Spectra

While the species development can be used to generate a spectra development that is then substracted from the matrix, the option ext_spectra that is available in the Fit_Global can be used to assign a specific spectrum to a species. The ext_spectra needs to be a pandas dataframe with the wavelength (or energy) as index and the name of species that is suppose to be replaced by the provided spectrum as column name. If the parameter set contains a parameter "ext_spectra_shift" this external spectrum will be moved by that parameter. As this is an external parameter, this can be optimized the usual way. Similarly the parameter "ext_spectra_scale" is multiplied to all spectra given. The parameter **'explicit_GS'** is a keyword that if present adds the ground state (including the bleach) an explicit component. If a parameter with name **ext_spectra_guide** is present the external spectra will not be used as absolute spectra but as guides. This means that the spectra is substracted. Then during the spectral fitting phase (in the function fill_int) a new spectra is fitted that is then the difference that is "missing" the returned DAC is then the sum of the two spectra and the

"real spectrum". This is very useful as it allows to e.g. provide the ground state spectrum without making it exclusive, meaning not all features need to be present. The inclusion of this feature was inspired by Glotaran, but implemented in my own way. I recommend to check the documentation to e.g. err_func for more details.

CHAPTER

TEN

COMPARATIVE PLOTTING

The comparative plotting is an important tool to compare different measurements (e.g. different conditions), different fits or steady state spectra. In general you can compare different kinetics (at one or multiple fixed wavelength) with *Compare_at_wave* , compare different spectra at one or multiple given timepoints with *Compare_at_time* and compare the extracted spectra (decay associated or species associated) with *Compare_DAC*. The essential idea is that the parameters in the project that contains the comparative plotting are used for all the plots. So the ta.bordercut or ta.intensity is used for all the plot, independent of e.g. the intensity in the other projects.

New is that the compare functions take "other" as a parameter, which can be either a single or multiple projects (TA - objects). These projects need to be loaded into the program. Loading a project can be done by having them open from prior import or analysis (e.g. when comparing different fits) and then using

self.Copy()

More usual other (saved) projects will be opened with the function

pf.GUI_open()

. See *Opening multiple files* for more information on that.

As this comparision very often contains a lot of files the images are automatically saved using the filenames and the wavelength/time points. The images are however open and if the standard approach using QT was used can be manipulated using the GUI tools. So is e.g. the conversion into a linear time-axis currently not implemented, but can easily be achieved by changing the axis in the QT GUI.

A very important function provided by this set of tools is the comparision against other spectra. So can for examples be reference spectra (e.g. UV-vis) be added to the plots.

10.1 Normalization and Scaling

An important option is the normalization in a certain window that applies for both Compare_at_time and Compare_at_wave. Very often data needs to be normalized before it can be compared to anything e.g. to the size of the ground state bleach or an excited state absorption. Here I offer the normalization in a certain window in time and space. In this window a value in the "ta" and then each of the "other" projects is calculated. The intensity of each in the other projects (but not of the "ta" project) is then mutliplied by the quotient of this value in this specific window. This means e.g. that even if the kinetics is plotted for 600nm the normalization can be performed at 1-2ps at 450-500nm. This is very useful to plot e.g. the efficiency of product formation in the study of catalytic processes. For this normalization a "window" needs to be defined in the order:

```
[Start time, End time, Start wavelength, End Wavelength]
```

Care should be take to where this normalization is performed. The region just around t=0 is dangerous due to the artifacts, that do not represent real values. If external values are suppose to be used for scaling, the individual intensities can be manipulated. For each of the loaded projects ta.ds is the matrix that contains the data. With:

```
"ta.ds*=1.1"
```

could for example the individual intensity be raised by 10%. But be aware that with this action you are changing the data in the object. The original data ta.ds_ori remains unchanged. If you save and reload the data, the intensity will revert to the originally measured value.

10.2 Compare_at_time

This function plots multiple spectra into the same figure at one or multiple given timepoints (rel_time) and allows for *Normalization and Scaling*

Very useful to compare the spectra for different solvents or quenchers, or e.g. different fits. The ta.time_width_percent parameter defines if this is a single time (if time_width_percent = 0) or an integrated window.

A normalization window can be given at which all the plotted curves are normalized to. This window does not have to be in the plotted region. See *Normalization and Scaling*

Very often one would like to compare the measured spectra at a certain time to an external spectrum (e.g. spectro-electro-chemistry or steady state absorption). This can be done by loading a specific spectrum into a DataFrame and handing this data Frame to the comparision function. The function can also be used to plot e.g. the measured spectra vs. an external spectrum without giving any "other" Projects.

For more information, details on the parameter and examples see:

self.Compare_at_time()

10.3 Compare_at_wave

This function plots multiple kinetics into the same figure at one or multiple given wavelength (rel_wave) and allows for *Normalization and Scaling* Very useful to compare the kinetics for different quencher concentrations or pump powers, or e.g. different fits. The parameter width or the general ta.wavelength_bin defines the width of the spectral window that is integrated and shown.

A normalization window can be given at which all the plotted curves are normalized to. This window does not have to be in the plotted region. See *Normalization and Scaling*

Often multiple wavelength are to be plotted, and if at the same time many projects go into the same plot, things tend to get messy. As the files are saved separately this approach proofed to be useful.

For more information, details on the parameter and examples see:

self.Compare_at_wave()

10.4 Compare_DAC

This is a convenience function to plot multiple extracted spectra (DAS or species associated) into the same figure or into a separate figure each. Other should be ta.plot_func objects (loaded or copied). By standard it plots all into the same window. If all project have the same number of components one can activate "separate_plots" and have each separated (in the order created in the projects).

The "Spectra" parameter allows as before the inclusion of an external spectrum. Others is optional and I use this function often to compare species associated spectra with one or multiple steady state spectra.

For more information, details on the parameter and examples see:

```
self.Compare_DAC()
```

CHAPTER

ELEVEN

DATA EXPORT AND PROJECT SAVING

11.1 Save_Plots

Save Plots *self.Save_Plots()*

Convenience function that calls both "Plot_RAW" and if possible "Plot_fit_output" and saves the plots. Effectively this is intented to be used at the end of a quick exploratory work to capture a status. The parameter are indentical to the two plotting functions and are handed through.

11.2 Save_Powerpoint

Save Plots as Powerpoint *self.Save_Powerpoint()*

Convenience function that calls both "Plot_RAW" and if possible "Plot_fit_output" and saves the plots as "png".

Then it creates a power point file with one slide for the RAW plots and one slide for the Fits. Effectively this is intented to be used at the end of the a quick exploratory work to capture a status and create a quick presention slide. The parameter are intentical to the plotting functions and are handed through. The additional switches save_RAW and save_Fit are convenience to allow for faster processing.

If the "savetype" contains 'png', 'svg', or 'pdf' then a summary file is created that looks close to the powerpoint file.

11.3 Saving of the project

Save Project as hdf5 *self.Save_project()*

This software allows the saving of the TA project as a HDF5 file that can be reloaded. The HDF5 file contains all the set parameter as well as the fit results (if applicable) and the raw data. To reduce the space consumption we save ta.ds_ori and the parameter that are used to create ta.ds. If manual changes were made to ta.ds, these have to be stored externally. As only the obvious errors are filtered in ta.Filter_data this is can savely replace the original data File. We are also saving the arrival-time (chirp) correction in the file and restore the chirp corrected data ta.ds during the import. The import function understands the file type and re-creates the object.

The one limitation to this method is the external fit function. If an external **ta.mod** is used, the save function stores the name and the documentation string of this function as a string. So after reloading of the analysis object the external function will have to be set with ta.mod=imported_function. The parameter of the fit are however stored. Only the filename and the path of the file can be changed during saving of the project. If left empty the path and filename of the original ASCII file is used.

11.4 Save ASCII data

Save/export data as ascii/text `self.Save_data()`

This is a convenient function to export the data for use with other plotting programs, the chirp corrected data, all the slices defined by ta.rel_wave and ta.rel_time for both the fits and a the RAW data. The external options include: save_RAW,save_Fit while there is an automatic that recognizes if for example fit data is present, this switch allows the manual selection which datasets are stored. save_slices selected if the slices defined by ta.rel_wave and ta.rel_time are saved save_binned this switch chooses if the chirp corrected and rebinned dataset (ta.ds with ta.wavelength_nm_bin) is saved. If the ta.wavelength_nm_bin is None, this saves the chirp corrected RAW data. filename sets the basis filename that is used for all the files path this can be a full path or a simple string, defining a folder relative to the folder in ta.path. If the folder does not exist, it will be created, if it exists a file with exactly the same name will be overwritten without confirmation sep defines the separator user to separate different values. Standard is a "TAP". A good choice would also be a space or a comma, unless you are located in one of the countries that uses commas for decimal points. Decimals will be separated with "dots".

This function by default also dumps a text file with the fit results

CHAPTER

TWELVE

CHANGELOG

In this changelog I will only mention the new capabilities. All other version jumps are bug fixes

12.1 7.4.9

Cleaned up previous updates.

12.2 7.3.6

Introduced that Print_results can dump the result parameter into a file.

12.3 7.3.0

introduced: sub_sample, pulse_sample into the fitting (not yet documented) but they allow to temporarily add time points to the

Pulse_sample is needed when the pulse is not in the modelled data (e.g. when the time_limit or ignore_time_regions is set sub_sample divided the times and is useful if the measure data is to sparse in time. if the parameter "sub_steps" is present this will be used to define the number of sub_steps in the iterative sampling

12.4 7.2.17

The to sparsely measured datapoints can now be sub-sampled with ta.Fit_Global(sub_sample=10) The intensity is now proper if the modelled points do not include the pump pulse

12.5 7.2.5

On Windows the fit can now be interrupted if "q" is pressed

12.6 7.2.1

Add write_paras as Fit_Global option. This will print the params continuously to the output. For now this serves as a way to interrupt the fitting and not loose the results

12.7 7.1.20

Added Check_Chirp function that allows to check (look at) a chirp vector vs the data

12.8 7.1.2

Add a description call to the TA object meaning that "ta()" will give you a mini instruction

12.9 7.1.1

- implemented the option to define the external spectrum as explicit or as guidance spectrum
- Explicit Ground state added

12.10 7.0.3

- Allows mutli data fit with sam DAS without real fitting (for parameter check)
- move from qt to tk in all notebooks

12.11 7.0.0

Change to tk version as default (no qt needed)

CHAPTER

THIRTEEN

FUNCTION INDEX

CHAPTER

FOURTEEN

KIMOPACK - FUNCTIONS

class plot_func.**TA**(*filename*, *path=None*, *sep='\t'*, *decimal='.'*, *index_is_energy=False*, *transpose=False*, *sort_indexes=False*, *divide_times_by=None*, *shift_times_by=None*, *external_time=None*, *external_wave=None*, *use_same_name=True*, *data_type=None*, *units=None*, *baseunit=None*, *ds=None*, *conversion_function=None*)

Background(*lowlimit=None*, *uplimit=- 1*, *use_median=False*, *ds=None*, *correction=None*)

This is the background correction. In general it for each measured wavelength averages the values from 'lowlimit' to 'uplimit' and subtracts it from the data. It rund on the object (global) or if given a specific ds local. The low and uplimit can be set anywhere to substract any background. It is important to note that many problems during measurements might be visible in the data before time zero. So I recommend to first plot without background correction and only after this inspection apply the background correction. The fit function has its own way to calculcate and apply a background That could be used instead (but making the fit less stable)

Parameters

- **lowlimit** (*None or float, optional*) – this is the lower limit from which the average (or median) is taken (Default) is None, in which case the lower limit of the data is used.
- **uplimit** (*None or float, optional*) – this is the upper limit until which the average (or median) is taken (Default) is -1 (usually ps), in which case the lower limit of the data is used.
- **use_median** (*bool, optional*) – the Median is a more outlier resistant metric in comparision to the Mean (Average). However the values are not quite as close to the distribution center in case of very few values. False (Default) means the Mean is used
- **ds** (*None or DataFrame, optional*) – if None (Default) the internal Dataframe self.ds is used, otherwise the pandas DataFrame ds is corrected and returned
- **correction** (*None or DataFrame, optional*) – this is the correction applied. It must be a DataFrame with the same numbers of columns (spectral points) as the used ds

Examples

if the object self has the name "ta"

typical useage:

```
>>> ta.Background()
```

specify inegrated are to - inf (Default) up to -0.5ps and use the Median for computation

```
>>> ta.Background(uplimit = -0.5, use_median = True)
```

Compare_DAC(*other=None*, *spectra=None*, *separate_plots=False*, *cmap=None*)

This is a convenience function to plot multiple extracted spectra (DAS or species associated) into the same figure or into a separate figure each. Other should be ta.plot_func objects (loaded or copied). By standard it plots all into the same window. If all project have the same number of components one can activate "separate_plots" and have each separated (in the order created in the projects).

The "Spectra" parameter allows as before the inclusion of an external spectrum. Others is optional and I use this function often to compare species associated spectra with one or multiple steady state spectra.

Parameters

- **other** (*TA object or list of those, optional*) – should be ta.plot_func objects (loaded or copied) and is what is plotted against the data use a list [ta1,ta2,...] or generate this list using the Gui function. See section *Opening multiple files* in the documentation
- **spectra** (*None or DataFrame, optional*) – If an DataFrame with the wavelength as index is provided, Then the spectra of each column is plotted into the differential spectra 1:1 and the column names are used in the legend Prior scaling is highly suggested. These spectra are not (in general) scaled with the norm window. (see examples)
- **separate_plots** (*bool, optional*) – True or False (Default), separate plots is the switch that decides if a axis or multiple axis are used. This option will result in a crash unless all objects have the same number of DAS/SAS components
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.

Examples

```
>>> import plot_func as pf
>>> ta = pf.TA('test1.hdf5') #open the original project,
>>> this MUST contain a fit, otherwise this will raise an error
```

Now open a bunch of other projects to compare against,

```
>>> #compare in a single window
>>> other_projects = pf.GUI_open(project_list = ['file1.hdf5', 'file2.hdf5'])
>>> ta.Compare_DAC(others = other_project)
>>> #comprare in separate windows,
>>> #the other projects must have the same number of components
>>> ta.Compare_DAC(others = other_project, separate_plots = True)
```

Compare the DAC to an external spectrum

```
>>> ext_spec = pd.read_csv('Ascii_spectrum.dat', sep = ',')
>>> ta.Compare_DAC(spectra = ext_spec) #compare just the current solution
>>> ta.Compare_DAC(spectra = ext_spec, others = other_project) #compare multiple
```

Compare_at_time(*rel_time=None*, *other=None*, *fitted=False*, *norm_window=None*, *time_width_percent=None*, *spectra=None*, *data_and_fit=False*, *cmap=None*, *print_click_position=False*, *linewidth=1*, *title=''*, *plot_second_as_energy=True*)

This function plots multiple spectra into the same figure at a given rel_time (timepoints) and allows for normalization. Very useful to compare the spectra for different solvents or quenchers, or e.g. different fits. The ta.time_width_percent parameter defines if this is a single time (if time_width_percent = 0) or an integrated window. Only "rel_time" is a mandatory, the rest can be taken from the original project (ta).

The normalization is realized by giving a norm_window at which the intensity in the triggering object is integrated (in ta.Compare_at_time(other..) "ta" is the triggering object. The in each of the other curves the same window is integrated and the curve scaled by this value. Important to note is that this window does not need to be in the plot. e.g. the normalization can be done at a different time.

Very often one would like to compare the measured spectra at a certain time to an external spectrum (e.g. spectro-electro-chemistry or steady state absorption). This can be done by loading a specific spectrum into a DataFrame and handing this data Frame to the comparision function. The function can also be used to plot e.g. the measured spectra vs. an external spectrum without giving any "other" Projects. (very useful for comparisions).

Parameters

- **rel_time** (*float or list/vector (of floats)*) – Specify the times where to plot, single value or list/vector of values. For each entry in rel_time a spectrum is plotted. If time_width_percent=0 (Default) the nearest measured timepoint is chosen. For other values see parameter "time_width_percent".
- **other** (*TA object or list of those, optional*) – should be ta.plot_func objects (loaded or copied) and is what is plotted against the data use a list [ta1,ta2,...] or generate this list using the Gui function. See section *Opening multiple files* in the documentation
- **fitted** (*bool, optional*) – True/False (Default) - use fitted data instead of raw data. If True, the fitted datapoints (without interpolation) are used. This is intended for comparing e.g. different fits
- **norm_window** (*None or list/vector (with 4 floats), optional*) – norm_window Give a list/tupel/vector with 4 entries in the order [Start - time, End

- time, Start - wavelength, End - Wavelength], see section *Normalization and Scaling* in the documentation. If None (Default) no normalization is done.

- **linewidth** (*float, optional*) – linewidth to be used for plotting
- **time_width_percent** (*None or float, optional*) – "rel_time" and "time_width_percent" work together for creating spectral plots at specific timepoints. For each entry in rel_time a spectrum is plotted. If however e.g. time_width_percent=10 the region between the timepoint closest to $timepoint + 0.1xtimepointand : math :$ is averaged and shown (and the legend adjusted accordingly). If None (Default) is given, the value is taken from the triggering object (self.time_width_percent) This is particularly useful for the densly sampled region close to t=0. Typically for a logarithmic recorded kinetics, the timepoints at later times will be further appart than 10 percent of the value, but this allows to elegantly combine values around time=0 for better statistics. This averaging is only applied for the plotting function and not for the fits.
- **spectra** (*None or DataFrame, optional*) – If an DataFrame with the wavelength as index is provided, Then the spectra of each column is plotted into the differential spectra 1-1 and the column names are used in the legend Prior scaling is highly suggested. These spectra are not (in general) scaled with the norm window. (see examples).
- **data_and_fit** (*bool, optional*) – True or False (Default), choose if for the Fitted plot the raw data of the other projects is to be plotting in addition to the fitted line. For False (Default) Only the fit is plotted.
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.
- **plot_second_as_energy** (*bool, optional*) – For (Default) True a second x-axis is plotted with "eV" as unit
- **print_click_position** (*bool, optional*) – if True then the click position is printed for the spectral plots

Examples

```
>>> import plot_func as pf
>>> ta = pf.TA("test1.hdf5") #open the original project
```

Now open a bunch of other porjects to comare against

```
>>> other_projects = pf.GUI_open(project_list = ["file1.SIA", "file2.SIA"])
```

Typical use is compare the raw data without normalization at 1ps and 6ps.

```
>>> ta.Compare_at_time(rel_time = [1,6], others = other_project)
```

Compare the fit withput normalization at 1ps and 6ps.

```
>>> ta.Compare_at_time(rel_time = [1,6], others = other_project, fitted = True)
```

Compare with normalization window between 1ps and 2ps and 400nm and 450nm.

```
>>> norm_window=[1,2,400,450]
>>> ta.Compare_at_time(rel_time = [1,6], others = other_project, norm_window =␣
↪norm_window)
```

Compare the spectrum at 1ps and 6ps with an external spectrum.

```
>>> ext_spec = pd.read_csv("Ascii_spectrum.dat", sep = ",")
>>> ta.Compare_at_time(rel_time = [1,6], spectra = ext_spec)
```

Use example - Often there are a lot of different measurements to compare at multiple time. The normlization is performed at the ground state bleach 460 nm and early in time. Then it is better to make a new plot for each timepoint. The normalization window stays fixed.

```
>>> plt.close("all") #make some space
>>> norm_window=[0.3,0.5,450,470] #define window in ground state bleach
>>> for t in [0.3,0.5,1,3,10,30]: #iterate over the wavelength
>>>     ta.Compare_at_time(rel_time = t, others = other_project, norm_window =␣
↪norm_window)
```

Compare_at_wave(*rel_wave=None*, *other=None*, *fitted=False*, *norm_window=None*, *width=None*, *cmap=None*, *data_and_fit=False*, *scale_type='symlog'*, *linewidth=1*)

This function plots multiple kinetics into the same figure at one or multiple given wavelength (rel_wave) and allows for *Normalization and Scaling* Very useful to compare the kinetics for different quencher concentrations or pump powers, or e.g. different fits. The parameter width or the general self.wavelength_bin which is used if width is None (Default) defines the width of the spectral window that is integrated and shown.

A normalization window can be given at which all the plotted curves are normalized to. This window does not have to be in the plotted region. See *Normalization and Scaling*

Parameters

- **rel_wave** (*float or list/vector (of floats)*) – Specify the wavelength where to plot the kinetics, single value or list/vector of values (only mandatory entry) For each entry in rel_wave a kinetic is plotted. 'rel_wave' and 'width' (in the object called 'wavelength_bin' work together for the creation of kinetic plots. At each selected wavelength the data between wavelength+width/2 and wavelength-width/2 is averaged for each timepoint

- **other** (*TA object or list of those, optional*) – should be ta.plot_func objects (loaded or copied) and is what is plotted against the data use a list [ta1,ta2,...] or generate this list using the Gui function. See section *Opening multiple files* in the documentation
- **fitted** (*bool, optional*) – True/False (Default) - use fitted data instead of raw data. If True, the fitted datapoints (without interpolation) are used. This is intended for comparing e.g. different fits
- **norm_window** (*None or list/vector (with 4 floats), optional*) – norm_window Give a list/tupel/vector with 4 entries in the order [Start - time, End - time, Start - wavelength, End - Wavelength], see section *Normalization and Scaling* in the documentation. If None (Default) no normalization is done.
- **width** – Specify the width above and below the given wavelength that is integrated as window. If left to (Default) "None" the value from ta is used.
- **data_and_fit** (*bool, optional*) – True or False (Default), choose if for the Fitted plot the raw data of the other projects is to be plotting in addition to the fitted line. For False (Default) Only the fit is plotted.
- **linewidth** (*float, optional*) – linewidth to be used for plotting
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.
- **Scale_type** (*None or str*) – is a general setting that can influences what time axis will be used for the plots. "symlog" (linear around zero and logarithmic otherwise) "lin" and "log" are valid options.

Examples

```
>>> import plot_func as pf
>>> ta = pf.TA('test1.hdf5') #open the original project
```

Now open a bunch of other projects to compare against

```
>>> other_projects = pf.GUI_open(project_list = ['file1.SIA', 'file2.SIA'])
```

Typical use: Compare the raw data without normalization at 400 nm and 500 nm

```
>>> ta.Compare_at_wave(rel_wave = [400, 500], others = other_project)
```

Compare the quality of the fit data without normalization at 400 nm and 500 nm

```
>>> ta.Compare_at_wave(rel_wave = [400, 500], others = other_project, fitted =␣
↪True)
```

Compare with normalization window between 1ps and 2ps and 400nm and 450nm

```
>>> norm_window=[1,2,400,450]
>>> ta.Compare_at_wave(rel_wave = [400, 500], others = other_project, norm_
↪window = norm_window)
```

Use example: Often there are a lot of different measurements to compare at multiple wavelength. The normlization is performed at the ground state bleach 460 nm and early in time. Then it is better to make a new plot for each wavelength. The normalization window stays fixed.

```
>>> plt.close('all') #make some space
>>> norm_window=[0.3,0.5,450,470] #define window in ground state bleach
>>> for wave in [300,400,500,600,700]: #iterate over the wavelength
>>>     ta.Compare_at_wave(rel_wave = wave, others = other_project, norm_window␣
↪= norm_window)
```

Copy()
returns a deep copy of the object.

Examples

>>>ta=plot_func.TA('testfile.hdf5') #open a project >>>ta1=ta.Copy() #make a copy for some tests or a differnet fit

Cor_Chirp(*chirp_file=None, path=None, shown_window=[-1, 1], fitcoeff=None, max_points=40, cmap=<matplotlib.colors.LinearSegmentedColormap object>*)
Cor_Chirp is a powerful Function to correct for a different arrival times of different wavelength (sometimes call chirp). In general if a file is opened for the first time this function is opening a plot and allows the user to select a number of points, which are then approximated with a 4th order polynomial and finally to select a point that is declared as time zero. The observed window as well as the intensities and the colour map can be chosen to enable a good correction. Here a fast iterating colour scheme such as "prism" is often a good choice. In all of the selections a left click selects, a right click removes the last point and a middle click (sometime appreviated by clicking left and right together) finishes the selection. If no middle click exists, the process automatically ends after max_points (40 preset).

The first option allows to fine select an intensity setting for this chirp correction. However sometimes spikes are making this things difficult. In this case set a guessed intensity with self.intensity_range=1e-3

Note that scattercut, bordercut and intensity_range can be used

After the first run the polynom is stored in self.fitcoeff, a new matrix calculated from self.ds_ori that is stored as self.ds and a file stored in the same location as the original data. The second time the function *Cor_Chirp* is run the function will find the file and apply the chirp correction automatically.

If one does want to re-run the chirp correction the function *Man_Chirp* does not look for this file, but creates after finishing a new file.

Alternatively the polynom or a filename can be given that load a chirp correction (e.g. from a different run with the same sample). The function *Cor_Chirp* selects in the order:

“fitcoeff” # “other files” # “stored_file” # call Man_Chirp (clicking by hand)

Parameters

- **chirp-file** (*None or str, optional*) – If a raw file was read(e.g. “data.SIA”) and the chirp correction was completed, a file with the attached word “chirp” is created and stored in the same location. (“data_chirp.dat”) This file contains the 5 values of the chirp correction. By selecting such a file (e.g. from another raw data) a specific chirp is applied. If a specific name is given with **chirp_file** (and optional **path**) then this file is used.

  GUI

  The word *‘gui’* can be used instead of a filename to open a gui that allows the selection of a chrip file

- **path** (*str or path object (optional)*) – if path is a string without the operation system dependent separator, it is treated as a relative path, e.g. data will look from the working directory in the sub director data. Otherwise this has to be a full path in either strong or path object form.

- **shown_window** (*list (with two floats), optional*) – Defines the window that is shown during chirp correction. If the t=0 is not visible, adjust this parameter to suit the experiment. If problems arise, I recomment to use Plot_Raw to check where t=0 is located

- **fitcoeff** (*list or vector (5 floats), optional*) – One can give a vector/list with 5 numbers representing the parameter of a 4th order polynomial (in the order $(a4 * x^4 + a3 * x^3 + a2 * x^2 + a1 * x1 + a0)$. The chirp parameter are stored in ta.fitcoeff and can thus be used in other TA objects. This vector is also stored with the file and automatically applied during re-loading of a hdf5-object

- **max_points** (*int, optional*) – Default = 40 max numbers of points to use in Gui selection. Useful option in case no middle mouse button is available. (e.g. touchpad)

- **cmap** (*matplotlib colourmap, optional*) – Colourmap to be used for the chirp correction. While there is a large selection here I recommend to choose a different map than is used for the normal 2d plotting.

  cm.prism (Default) has proofen to be very usefull

Examples

In most cases:

```
>>> import plot_func as pf
>>> ta = pf.TA('test1.SIA') #open the original project,
>>> ta.Cor_Chirp()
```

Selecting a specific correction

```
>>> ta.Cor_Chirp('gui')
>>> ta.Cor_Chirp(chirp_file = 'older_data_chirp.dat')
>>> #use the coefficients from a different project
>>> ta.Cor_Chirp(fitcoeff = ta_old.fitcoeff) #use the coefficients from a␣
↪different project
```

Filter_data(*ds=None*, *cut_bad_times=False*, *replace_bad_values=0*, *value=20*, *uppervalue=None*, *lowervalue=None*, *upper_matrix=None*, *lower_matrix=None*)

Filteres the data by applying hard replacements. if both replace_bad_values and cut_bad_times are false or None, the times above “value” are replaced by zero

Parameters

- **ds** (*pandas Dataframe, optional*) – if this is None (default) then the self.ds and self.ds_ori wil be filtered
- **value** (*float, optional*) – all values above this (absolute) value are considered to be corrupted. (Default 20) as classically the setup reports optical DEnsity, an OD of 20 would be far above the typically expected OD 1e-3. Pascher instrument software uses a value of 21 to indicate an error.
- **uppervalue** (*float, optional*) – all values above this number are considered to be corrupted. (Default 20) as classically the setup reports optical DEnsity, an OD of 20 would be far above the typically expected OD 1e-3. Pascher instrument software uses a value of 21 to indicate an error.
- **lowervalue** (*float, optional*) – all values below this number are considered to be corrupted. (Default -20) as classically the setup reports optical DEnsity, an OD of -20 would be far above the typically expected OD 1e-3. Pascher instrument software uses a value of 21 to indicate an error.
- **replace_bad_values** (*None of float, optional*) – values above the treshold are replaced with this value. Ignored of None (Default)
- **bool** (*cut_bad_times =*) – True (Default=False) removes the whole time where this is true
- **optional** – True (Default=False) removes the whole time where this is true
- **upper_matrix** (*Pandas DataFrame, optional*) – all values above this treshold will be put N/A or replace by the value in replace_bad_values
- **DataFrame** (*lower_matrix Pandas*) – all values below this treshold will be put N/A or replace by the value in replace_bad_values
- **optional** – all values below this treshold will be put N/A or replace by the value in replace_bad_values
- **everything** (*the value is the upper bound.*) –
- **wrong** (*above will be filtered. Standard is to drop the rows(=times) where something went*) –

Examples

typical usage

```
>>> import plotfunc as pf
>>> ta=pf.TA('testfile.SIA')
>>> ta.Filter_data()
>>> ta.Filter_data(value=1) #to filter times with at least one point with OD 1
```

Fit_Global(*par=None*, *mod=None*, *confidence_level=None*, *use_ampgo=False*, *other_optimizers=None*, *fit_chirp=False*, *fit_chirp_iterations=10*, *multi_project=None*, *unique_parameter=None*, *weights=None*, *same_DAS=False*, *dump_paras=False*, *dump_shapes=False*, *filename=None*, *ext_spectra=None*, *write_paras=False*, *tol=1e-05*, *sub_sample=None*, *pulse_sample=None*)

This function is performing a global fit of the data. As embedded object it uses the parameter control options of the lmfit project as an essential tool. (my thanks to Matthew Newville and colleagues for creating this phantastic tool) [M. Newville, T. Stensitzki, D. B. Allen, A. Ingargiola, 2014. DOI: 10.5281/ZENODO.11813.]. The what type of fitting is performed is controlled by setting of the parameter here.

The general fitting follows this routine:

1. create a copy of the Data-Matrix self.ds is created with the shaping parameters
2. Then a Matrix is created that represents the fractional population of each species (or processes in case of the paral model). This Matrix contains one entry for each timepoint and represents the kinetic model based upon the starting parameter. (see below for a description of the models). This model formation can by done by using a build in or a user supplied function. (handled in the function "pf.build_c") -> If an ext_spectra is provided this its intensity is substacted from the matrix (only for external models)
3. Then the process/species associated spectra for each of the species is calculated using the linalg.lstsq algorithm from numpy (https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html)
4. From the convoluted calculated species concentrations and spectra a calculated matrix is formed (handled in the function "pf.fill_int")
5. The difference between calculated and measured spectra is calculated, point-wise squared and summed together. (function "err_func" or "err_func_multi" if multiple datasets are fitted)
6. This difference is minimized by iterating 2-4 with changing parameters using an optimization algorithm (generally nelder-mead simplex)
7. Finally in a last run of 2-5 the final spectra are calculated (using the "final" flag) and the optimized parameter, the matrixes ("A"-measured, "AC" - calculated, "AE" - linear error), spectra (always called "DAS") the concentrations (called "c") are written in the dictionary "ta.re" together with a few result representations and other fit outputs. The optimized parameter are also written into ta.par_fit (as an parameter object) that can be re-used as input into further optimization steps.
8. Under Windows we load the keyboard library and the Fit can be interrupted by pressing the "q" key. Consider using the parameter write_paras or dump_paras to observe details during the fit.

All mandatory parameters are in general taken from the internal oject (self) The optional parameter control the behaviour of the fitting function

Parameters

- **par** (*lmfit parameter oject, optional*) – Here another parameter object could be given,overwriting the (Default is self.par)
- **mod** (*str or function, optional*) – Give a extra model selection (Default uses self.mod) internal modells: 'paral','exponential','consecutive','full_consecutive' see also *plot_func.build_c()* and *plot_func.err_func()*
- **confidence_level** (*None or float (0.5-1), optional*) – If this is changed from None (Default) to a value between 0.5 and 1 the code will try to calculate the error of the parameter for the fit. For each parameter that can vary a separate optimization is performed, that attempts to find the upper and lower bound at which the total error of the re-optimized globally fitted results reaches the by F-statistics defined confidence bound. See *plot_func.s2_vs_smin2()* for details on how this level is determined. Careful, this option might run for very long time. Meaning that it typically takes 50 optimization per variable parameter (hard coded limit 200) The confidence level is to be understood that it defines the e.g. 0.65 * 100% area that the parameter with this set of values is within this bounds. Normal behaviour for this is to re-optimise the parameter during the optimization. if the parameter par['error_param_fix'] is present, this will be suppressed.
- **use_ampgo** (*bool, optional*) – (Default) is False Changes the optimizer from a pure Nelder mead to Ampgo with a local Nelder Mead. For using this powerfull tool all parameter need to have a "min" and a "max" set. Typically takes 10-40x longer than a stan-

dard optimization, but can due to its tunneling algorithm more reliably find global minima. see:https://lmfit.github.io/lmfit-py/fitting.html for further details

- **other_optimizers** (*str, optional*) – (Default) is None if this is changed from None to a string that exists in lmfit, then this optimizer will be used instead. Useful choices are e.g. **least_squares** or similar words. This is particularly useful if the problem does not lend to be solved with nelder-mead. This includes e.g. osciallations.
- **fit_chirp** (*bool, optional*) – (Default) is False a powerful optimization of the chirp parameter. For this to work the data needs to include timepoints before and after t=0 and one should have reached a decent fit of most features in the spectrum. We perform an Nelder-Mead optimisation of the parameter followed by a Nelder-Mead optimization of the chirp parameter as one iteration. After each consecutive optimization it is checked if the total error improved. If not the fit is ended, if yes the maximum number of iterations 'fit_chirp_iterations' is performed. Warning, this goes well in many cases, but can lead to very strange results in others, always carefully check the results. I recommend to make a copy of the object before runnning a chirp optimization.
- **fit_chirp_iterations** (*int, optional*) – maximum number of times the global - chirp loop is repeated. Typically this iterations run 2-5 times, (Default) is 10
- **dump_paras** (*bool, optional*) – (Default) is False, If True creates two files in the working folder, one with the currently used parameter created at the end of each optimisation step, and one with the set of parameter that up to now gave the lowest error. Intented to store the optimisation results if the fit needs to be interrupted (if e.g. Ampgo simply needs to long to optimize.) useful option if things are slow this parameter also triggers the writing of fitout to a textfile on disc
- **dump_shapes** (*bool, optional*) – this option dumps the concentratoin matrix and the DAS onto disk for each round of optimization, mostly useful for multi-project fitting that wants to use the spectral or temporal intensity
- **write_paras** (*bool, optional*) – if True(Default) writes the currently varried values to screen
- **filename** (*None or str, optional*) – Only used in conjunction with 'dump_paras'. The program uses this filename to dump the parameter to disk
- **multi_project** (*None or list (of TA projects), optional*) – This switch is triggering the simultaneous optimisation of multiple datasets. multi_project is as (Default) None. it expects an iterable (typically list) with other TA projects (like ta) that are then optimised with the same parameter. This means that all projects get the same parameter object for each iteration of the fit and return their individual error, which is summed linearly. The "weights" option allows to give each multi_project a specific weight (number) that is multiplied to the error. If the weight object has the same number of items as the multi_project it is assumed that the triggering object (the embedded project) has the weight of 1, otherwise the first weight is for the embedded project. The option 'unique_parameter' takes (a list) of parameter that are not to be shared between the projects (and that are not optimized either) The intended use of this is to give e.g. the pump power for multiple experiments to study non linear behaviour. Returned will be only the parameter set for the optimium combination of all parameter. Internally, we iterate through the projects and calculate for each project the error for each iteration. Important to note is that currently this means that each DAS/SAS is calculated independently! For performing the same calculation with a single DAS, the Matrixes need to be concatenated before the run and an external function used to create a combined model. As this is very difficult to implement reliably For general use (think e.g. different pump wavelength) this has to be done manually.
- **unique_parameter** (*None or str or list (of strings), optional*) – only

used in conjunction with 'multi_project', it takes (a list) of parameter that are not to be shared between the projects (and that are not optimized either) The intended use of this is to give e.g. the pump power for multiple experiments to study non linear behaviour. (Default) None

- **same_DAS** (*bool,optional*) – changes the fit behavior and uses the same DAS for the optimization. This means that the ds are stacked before the fill_int rounds. This option is only used in multi-project fitting
- **weights** (*list of floats, optional*) – only used in conjunction with 'multi_project'. The "weights" option allows to give each multi_project a specific weight (number) that is multiplied to the error. If the weight object has the same number of items as the 'multi_project' it is assumed that ta (the embedded project) has the weight of 1, otherwise the first weight is for the embedded object
- **ext_spectra** (*DataFrame, optional*) – (Default) is None, if given substract this spectra from the DataMatrix using the intensity given in "C(t)" this function will only work for external models. The name of the spectral column must be same as the name of the column used. If not the spectrum will be ignored. The spectrum will be interpolated to the spectral points of the model ds before the substraction. a number of parameters can be defined to aid this process. These parameter are defined as normal parameters. "ext_spectra_scale" multiplies all spectra by this value (e.g. -1 to put the steady state absorption spectra in) "ext_spectra_shift" shifts all spectra by this value to compensate for calibration differences "ext_spectra_guide" (from version 7.1.0) This is a switch, if this keyword is present, then the spectra are used as guides and not exclusively. This means the code will assume that these spectra are correct and substract them, then calulate the difference and return as DAS the provided spectra plus the difference spectra
- **tol** (*float, optional*) – the tolerance value that is handed to the optimizer (absolute) for nelder-mead the moment this means: df < tol (corresponds to fatol) number_of_function_evaluations < maxfev (default 200 * n variables) number_of_iterations < maxiter (default 200 * n variables)

Returns

- **re** (*dict*) – the dictionary "re" attached to the object containing all the matrixes and parameter. The usual keys are: "A" Shaped measured Matrix "AC" Shaped calculated Matrix "AE" Difference between A and AC = linear error "DAS" DAS or SAS, labeled after the names given in the function (the columns of c) Care must be taken that this mesured intensity is C * DAS, the product. For exponential model the concentrations are normalized "c" The Concentrations (meaning the evolution of the concentrations over time. Care must be taken that this mesured intensity is C * DAS, the product. For exponential model the concentrations are normalized "fit_results_rates" DataFrame with the fitted rates (and the confidence intervals if calculated) "fit_results_times" DataFrame with the fitted decay times (and the confidence intervals if calculated) "fit_output" The Fit object as returned from lmfit. (This is not saved with the project!) "error" is the S2, meaning AE**2.sum().sum() "r2"=1-"error"/(('A'-'A'.mean())**2).sum(), so the residuals scaled with the signal size
- **par_fit** (*lmfit parameter object*) – is written into the object as a lmfit parameter object with the optimized results (that can be use further)
- **fitcoeff** (*list, if chirpfit is done*) – The chirp parameter are updated
- **ds** (*DataFrame, if chirpfit is done*) – A new ds is calculated form ds_ori if ChripFit is done
- *The rest is mainly printed on screen.*

Examples

Non optional:

```
>>> ta=pf.TA('testfile.SIA') #load data
>>> ta.mod='exponential'    #define model
>>> ta.par=lmfit.Parameters()  #create empty parameter object
>>> ta.par.add('k0',value=1/0.1,vary=True) #add at least one parameter to␣
↪optimize
```

Trigger simple fit:

```
>>> ta.Fit_Global()
```

Trigger fit with Chrip Fit:

```
>>> ta.Fit_Global(fit_chirp=True)
```

Trigger iterative Chirp fitting with fresh refinement of the Global kinetic parametersfor i in range(5):

```
>>> for i in range(5):
>>>     start_error=ta.re['error']
>>>     ta.par=ta.par_fit
>>>     ta.Fit_Global(fit_chirp=True)
>>>     if not ta.re['error'] < start_error:break
```

Trigger fit fit error calculations

```
>>> ta.Fit_Global(confidence_level=0.66)
```

Trigger fit of multiple projects #use the GUI_open function to open a list of objects (leave empty for using the GUI)

```
>>> other_projects=pf.GUI_open(['sample_1.hdf5','sample_2.hdf5'],path='Data')
>>> ta.Fit_Global(multi_project=other_projects)
```

For more examples please see the complete documentation under *Fitting, Parameter optimization and Error estimation* or *Fitting multiple measured files at once*

Man_Chirp(*shown_window=[-1, 1], path=None, max_points=40, cmap=<matplotlib.colors.LinearSegmentedColormap object>, ds=None*)

Triggering of Manuel Fix_Chirp. usually used when Cor_Chirp has run already. Alternatively delete the chirp file. This Function opens a plot in which the user manually selects a number of points These points will then be interpolated with a 4th order polynomial The user can then select a new t=0 point. The first option allows to fine select an intensity setting for this chirp correction. However sometimes spikes are making this things difficult. In this case set a guessed intensity with self.intensity_range=1e-3

Parameters

- **path** (*str or path object (optional)*) – if path is a string without the operation system dependent separator, it is treated as a relative path, e.g. data will look from the working directory in the sub director data. Otherwise this has to be a full path in either strong or path object form.
- **shown_window** (*list (with two floats), optional*) – Defines the window that is shown during chirp correction. If the t=0 is not visible, adjust this parameter to suit the experiment. If problems arise, I recomment to use Plot_Raw to check where t=0 is located

- **max_points** (*int, optional*) – Default = 40 max numbers of points to use in Gui selection. Useful option in case no middle mouse button is available. (e.g. touchpad)
- **cmap** (*matplotlib colourmap, optional*) – Colourmap to be used for the chirp correction. While there is a large selection here I recommend to choose a different map than is used for the normal 2d plotting.

  cm.prism (Default) has proofen to be very usefull
- **ds** (*pandas dataframe,optional*) – this allows to hand in an external ds, if this is done then the on disk saved fitcoeff are the new ones only and the function returns the new fitcoeff and the combined fitcoeff, self also has a new variable called self.combined_fitcoeff the original file on dis and self.fitcoeff are NOT overwritten (are the old ones) the self.ds is the NEW one (with the correction applied) to reverse simply run Cor_Chirp() to permanently apply change self.fitcoeff with self.combined_fitcoeff and rename the file with 'filename_second_chirp' to filename_chirp

Plot_RAW(*plotting=range(0, 4)*, *title=None*, *scale_type='symlog'*, *times=None*, *cmap=None*, *filename=None*, *path='result_figures'*, *savetype='png'*, *print_click_position=False*, *plot_second_as_energy=True*, *ds=None*)

This is a wrapper function that triggers the plotting of various RAW (non fitted) plots. The shaping parameter are taken from the object and should be defined before. The parameter in this plot call are to control the general look and features of the plot. Which plots are printed is defined byt the first command (plotting) The plots are generated on the fly using self.ds and all the shaping parameter In all plots the RAW data is plotted as dots and interpolated with lines (using Savitzky-Golay window=5, order=3 interpolation). As defined by the internal parameters at selected time-points and the kinetics for selected wavelength are shaped by the object parameter. The SVD is performed using the same shaping parameter and is commonly used as an orientation for the number of components in the data. Everything is handed over to 'plot_raw' function that can be used for extended RAW plotting.

Parameters

- **plotting** (*int or iterable (of integers), optional*) – This parameter determines which figures are plotted the figures can be called separately with plotting = 1 or with a list of plots (Default) e.g. plotting=range(4) calls plots 0,1,2,3. The plots have the following numbers:

  0. Matrix
  1. Kinetics
  2. Spectra
  3. SVD

  The plotting takes all parameter from the "ta" object.
- **title** (*None or str*) – title to be used on top of each plot The (Default) None triggers self.filename to be used. Setting a specific title as string will. be used in all plots. To remove the title all together set an empty string with this command title="" .
- **Scale_type** (*None or str*) – is a general setting that can influences what time axis will be used for the plots. "symlog" (linear around zero and logarithmic otherwise) "lin" and "log" are valid options.
- **times** (*int*) – are the number of components to be used in the SVD (Default) is 6.
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it

is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.

- **filename** (*str, optional*) – offers to replace the base-name used for all plots (to e.g.~specify what sample was used). if (Default) None is used, the self.filename is used as a base name. The filename plays only a role during saving, as does the path and savetype.
- **path** (*None or str or path object, optional*) – This defines where the files are saved if the safe_figures_to_folder parameter is True, quite useful if a lot of data sets are to be printed fast. If a path is given, this is used. If a string like the (Default) "result_figures" is given, then a subfolder of this name will be used (an generated if necessary) relative to self.path. Use and empty string to use the self.path If set to None, the location of the plot_func will be used and a subfolder with title "result_figures" be generated here.
- **savetype** (*str or iterable (of str), optional*) – matplotlib allows the saving of figures in various formats. (Default) "png", typical and recommendable options are "svg" and "pdf".
- **print_click_position** (*bool, optional*) – if True then the click position is printed for the spectral plots
- **ds** (*DataFrame, optional*) – if None (Default), the program first tests self.ds and if this is not there then self.ds_ori. This option was introduced to allow plotting of other matrixes with the same parameter

Examples

Typically one would call this function empty for an overview. We name the object "ta" so with

```
>>> ta=pf.TA('testfile.SIA')
```

This would trigger the plotting of the 4 mayor plots for an overview.

```
>>> ta.Plot_RAW()
```

This would plot only the kinetics.

```
>>> ta.Plot_RAW(1)
>>> ta.Plot_RAW(plotting = 1)
```

Plot_fit_output(*plotting=range(0, 6)*, *path='result_figures'*, *savetype='png'*, *evaluation_style=False*, *title=None*, *scale_type='symlog'*, *patches=False*, *filename=None*, *cmap=None*, *print_click_position=False*, *plot_second_as_energy=True*)

plots all the fit output figures. The figures can be called separately or with a list of plots. e.g. range(6) call

plots 0-5 Manual plotting of certain type:

This is a wrapper function that triggers the plotting of all the fitted plots. The parameter in this plot call are to control the general look and features of the plot. Which plots are printed is defined by the first command (plotting) The plots are generated from the fitted Matrixes and as such only will work after a fit was actually completed (and the "re" dictionary attached to the object.) In all plots the RAW data is plotted as dots and the fit with lines

Contents of the plots

0. DAC contains the assigned spectra for each component of the fit. For a modelling with independent exponential decays this corresponds to the "Decay Associated Spectra" (DAS). For all other models this contains the "Species Associated Spectra" (SAS). According to the model the separate spectra are labeled by time (process) or name, if a name is associated in the fitting model. The spectra are shown in the extracted strength in the right pane and normalized in the left. Extracted strength means that the measured spectral strength is the intensity (concentration matrix) times this spectral strength. As the concentration maxima for all DAS are 1 this corresponds to the spectral strength for the DAS. (please see the documentation for the fitting algorithm for further details).
1. summed intensity. All wavelength of the spectral axis are summed for data and fit. The data is plotted in a number of ways vs linear and logarithmic axis. This plot is not ment for publication but very useful to evaluate the quality of a fit.
2. plot kinetics for selected wavelength (see corresponding RAW plot).
3. plot spectra at selected times (see corresponding RAW plot).
4. plots matrix (measured, modelled and error Matrix). The parameter are the same as used for the corresponding RAW plot with the addition of "error_matrix_amplification" which is a scaling factor multiplied onto the error matrix. I recommend to play with different "cmap", "log_scale" and "intensity_scale" to create a pleasing plot.
5. concentrations. In the progress of the modelling/fitting a matrix is generated that contains the relative concentrations of the species modelled. This plot is showing the temporal development of these species. Further details on how this matrix is generated can be found in the documentation of the fitting function. The modeled spectra are the convolution of these vectors (giving the time-development) and the DAS/SAS (giving the spectral development).

Parameters

- **plotting** (*int or iterable (of integers), optional*) – This parameter determines which figures are plotted the figures can be called separately with plotting = 1 or with a list of plots (Default) e.g. plotting=range(6) calls plots 0,1,2,3,4,5 The plots have the following numbers:

  0. DAS or SAS
  1. summed intensity
  2. Kinetics
  3. Spectra
  4. Matrixes
  5. Concentrations (the c-object)

  The plotting takes all parameter from the "ta" object unless otherwise specified

- **path** (*None, str or path object, optional*) – This defines where the files are saved if the safe_figures_to_folder parameter is True, quite useful if a lot of data sets are to be printed fast. If a path is given, this is used. If a string like the (Default) "result_figures"

is given, then a subfolder of this name will be used (an generated if necessary) relative to self.path. Use and empty string to use the self.path If set to None, the location of the plot_func will be used and a subfolder with title "result_figures" be generated here

- **savetype** (*str or iterable (of str), optional*) – matplotlib allows the saving of figures in various formats. (Default) "png", typical and recommendable options are "svg" and "pdf".
- **evaluation_style** (*bool, optional*) – True (Default = False) adds a lot of extra information in the plot
- **title** (*None or str, optional*) – "title=None" is in general the filename that was loaded. Setting a specific title will be used in all plots. To remove the title all together set an empty string with title=""
- **scale_type** (*str, optional*) – refers to the time-axis and takes, "symlog" (Default)(linear around zero and logarithmic otherwise) and "lin" for linear and "log" for logarithmic, switching all the time axis to this type
- **patches** (*bool, optional*) – If False (Default) the names "measured" "fitted" "difference" will be placed above the images. If True, then they will be included into the image (denser)
- **filename** (*str, optional*) – offers to replace the base-name used for all plots (to e.g.specify what sample was used). if (Default) None is used, the self.filename is used as a base name. The filename plays only a role during saving, as does the path and savetype
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.
- **print_click_position** (*bool, optional*) – if True then the click position is printed for the spectral plots

Examples

Typically one would call this function empty for an overview: After the minimum fit

```
>>> ta=pf.TA('testfile.SIA')
>>> ta.par=lmfit.Parameters()
>>> ta.par.add('k0',value=1/0.1,vary=True)
>>> ta.Fit_Global()
```

One usually plots the an overview

```
>>> ta.Plot_fit_output()
>>> ta.Plot_fit_output(plotting=range(6)) #is the same as before
>>> ta.Plot_fit_output(2) #would plot only the kinetics
>>> ta.Plot_fit_output(plotting = 2) #would plot only the kinetics
```

Save_Plots(*path='result_figures'*, *savetype=None*, *title=None*, *filename=None*, *scale_type='symlog'*, *patches=False*, *cmap=None*)

Convenience function that sets save_plots_to_folder temporarily to true and replots everything

Parameters

- **path** (*None, str or path, optional*) – (Default) None, if left on None, then a folder "result_figures" is created in the folder of the data (self.path)
- **savetype** (*str or iterable (of str), optional*) – matplotlib allows the saving of figures in various formats. (Default) "png", typical and recommendable options are "svg" and "pdf".
- **title** (*None or str, optional*) – (Default) None, Use this title on all plots. if None, use self.filename
- **filename** (*str, optional*) – (Default) None, Base name for all plots. If None, then self.filename will be used
- **scale_type** (*str, optional*) – "symlog" (Default), "linear", "log" time axis
- **patches** (*bool, optional*) – For true use white patches to label things in the 2d matrixes, to safe space for publication
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between

0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.

Examples

```
>>> ta.Save_Plots()
>>> ta.Save_Plots(patches = True)
```

Save_Powerpoint(*save_RAW=True*, *save_Fit=True*, *filename=None*, *path='result_figures'*, *scale_type='symlog'*, *title=None*, *patches=False*, *cmap=None*, *savetype='pptx'*)

This function creates two power point slides. On the first it summarizes the RAW plots and on the second (if existent) it summarizes the fitted results

Parameters

- **save_RAW** (*bool, optional*) – (Default) True then the first slide with the RAW data is created
- **save_Fit** (*bool, optional*) – (Default) True then the second slide with the Fitted data is created
- **path** (*None, str or path, optional*) – (Default) None, if left on None, then a folder "result_figures" is created in the folder of the data (self.path)
- **savetype** (*str or iterable (of str), optional*) – triggers the additional creation of a composite file in this format. matplotlib allows the saving of figures in various formats. (Default) "png", typical and recommendable options are "svg" and "pdf".
- **title** (*None or str, optional*) – (Default) None, Use this title on all plots. if None, use self.filename
- **filename** (*str, optional*) – (Default) None, Base name for all plots. If None, then self.filename will be used
- **scale_type** (*str, optional*) – 'symlog' (Default), 'linear', 'log' time axis
- **patches** (*bool, optional*) – For true use white patches to label things in the 2d matrixes, to safe space for publication
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.

Examples

```
>>> ta.Save_Powerpoint()
>>> ta.Save_Powerpoint(patches = True)
```

Save_data(*save_RAW=True*, *save_Fit=True*, *save_slices=True*, *save_binned=False*, *filename=None*, *save_fit_results=True*, *path='Data_export'*, *sep='\t'*)

handy function to save the data on disk as dat files. The RAW labeled files contain the chirp corrected values (self.ds)

the save_slices switch turns on the dump of the separate sliced figures (time and spectral)

Parameters

- **save_binned** (*bool, optional*) – is also the re-binned matrix to be saved.
- **save_slices** (*bool, optional*) – save the kinetics and spectra from the fitted data (with the fits)
- **sep** (*str, optional*) – what symbol is used to separate different number. (typical either 'tab' or comma
- **save_RAW** (*bool, optional*) – (Default) True then the first slide with the RAW data is created
- **save_Fit** (*bool, optional*) – (Default) True then the second slide with the Fitted data is created
- **path** (*None, str or path, optional*) – (Default) None, if left on None, then a folder "result_figures" is created in the folder of the data (self.path)
- **save_fit_results** (*bool, optional*) – if True (Default) a neatly formated file with the fit results is created and stored with the data
- **filename** (*str, optional*) – (Default) None, Base name for all plots. If None, then self.filename will be used

Examples

```
>>> ta.Save_Data
```

Save_project(*filename=None*, *path=None*)

function to dump all the parameter of an analysis into an hdf5 file. This file contains the ds_ori and all the parameter, including fitting parameter and results. One limitation is the fitting model. If the model is build in, so the model is 'exponential' or 'parallel' then the safing works. If an external model is used then the dostring of the external function is stored, but not the function itself.

Parameters

- **path** (*None, str or path, optional*) – (Default) None, if left on None, then a folder "Data" is created in the folder of the project (self.path)
- **filename** (*str, optional*) – (Default) None, Base name for all plots. If None, then self.filename will be used

Examples

```
>>> ta.Save_project()
```

__init__(*filename*, *path=None*, *sep='\t'*, *decimal='.'*, *index_is_energy=False*, *transpose=False*, *sort_indexes=False*, *divide_times_by=None*, *shift_times_by=None*, *external_time=None*, *external_wave=None*, *use_same_name=True*, *data_type=None*, *units=None*, *baseunit=None*, *ds=None*, *conversion_function=None*)

Function that opens and imports data into an TA object it is designed to open combined files that contain both the wavelength and the time. (e.g. SIA files as recorded by Pascher instruments software) or hdf5 projects saved by this software There are however a lot of additional options to open other ascii type files and adapt their format internally Attention times with Nan will be completely removed during the import

Parameters

- **filename** (*str*) –
  - expects a filename in string form for opening a single file.
  - alternatively 'gui' can be set as filename, then a TKinter gui is opened for select.
  - alternatively 'recent' can given as key word. in this case it tries to find a text file named "recent.dat" that should contain the path to the last file opened with the GUI. this file is then opened. if this file is not found the GUI is opened instead
- **path** (*str or path object (optional)*) – if path is a string without the operation system dependent separator, it is treated as a relative path, e.g. data will look from the working directory in the sub director data. Otherwise this has to be a full path in either strong or path object form.
- **sep** (*str (optional)*) – is the separator between different numbers, typical are tap (Backslash t) (Default) ,one or multiple white spaces 'backslash s+' or comma ','.
- **decimal** (*str (optional)*) – sets the ascii symbol that is used for the decimal sign. In most countries this is '.'(Default) but it can be ',' in countries like Sweden or Germany
- **index_is_energy** (*bool (optional)*) – switches if the wavelength is given in nm (Default) or in eV (if True), currently everything is handled as wavelength in nm internally
- **data_type** (*str, None*) – data_type is the string that represents the intensity measurements. Usually this contains if absolute of differential data. This is used for the color intensity in the 2d plots and the y-axis for the 1d plots
- **units** (*str (optional)*) – this is used to identify the units on the energy axis and to label the slices, recognized is 'nm', 'eV' and 'keV' but if another unit like 'cm^-1' is used it will state energy in 'cm^-1'. Pleas observe that if you use the index_is_energy switch the program tries to convert this energy into wavelength.
- **baseunit** (*str (optional)*) – this is used to identify the units on the developing/time axis. This is name that is attached to the index of the dataframe. setting this during import is equivalent to ta.baseunit
- **transpose** (*bool (optional)*) – if this switch is False (Default) the wavelength are the columns and the rows the times.
- **sort_indexes** (*bool (optional)*) – For False (Default) I assume that the times and energies are already in a rising order. with this switch, both are sorted again.
- **divide_times_by** (*None or float (optional)*) – here a number can be given that scales the time by an arbitary factor. This is actually dividing the times by this value.

Alternatively there is the variable self.baseunit. The latter only affects what is written on the axis, while this value is actually used to scale the times. None (Default) ignores this

- **shift_times_by** (*None, float (optional)*) – This a value by which the time axis is shifted during import. This is a useful option of e.g. the recording software does not compensate for t0 and the data is always shifted. None (Default) ignores this setting
- **data_type** – this is the datatype and effectively the unit put on the intensity axis (Default)'differential Absorption in $mathregular{Delta OD}$
- **external_time** (*None or str (optional)*) – Here a filename extension (string) can be given that contains the time vector. The file is assumed to be at the same path as the data and to contain a single type of separated data without header. If use_same_name = True (default) It assumes that this is the ending for the file. The filename itself is taken from the filename. e.g. if samp1.txt is the filename and external_time='.tid' the program searches samp1.tid for the times. The transpose setting is applied and sets where the times are to be inserted (row or column indexes) If use_same_name = False this should be the file containing the vector for the time (in the same format as the main file)
- **external_wave** (*None or str (optional)*) – Here a filename extension (string) can be given that contains the wavelength vector. If use_same_name = True (default) The file is assumed to be at the same path as the data and to contain a single type of separated data without header. This is the ending for the file. The filename itself is taken from the filename. e.g. if samp1.txt is the filename and external_wave='.wav' then the program searches samp1.wav for the wavelength. The transpose setting is applied and sets where the wavelength are to be inserted (columns or row indexes) If use_same_name = False this should be a full filename that contains the vector
- **use_same_name** (*bool, optional*) – this switches if the external filename included the loaded filename or is a separate file True(default)
- **ds** (*pandas.DataFrame (optional)*) – feed in an external dataframe instead of opening a file
- **conversion_function** (*function(optional)*) – function that receives should have the shape: return pandas Dataframe with time/frames in rows and wavelength/energy in columns, The function is tested to accept (in that order) a my_function(filename, external_time,external_wave), my_function(filename, external_time), my_function(filename,external_wave), my_function(filename) and return: the dataframe ds with the time_axis as rows and spectral axis as columns if the ds.index.name ia not empty the "time axis" is in to that name the spectral axis is in ds.columns.name the return is investigated if it is one, two, or three things. if two are returned then the second must be the name of what the intensity axis is. This value will then be set to data_type if three are returned the third is the baseunit (for the time axis) this allows to use the automatic naming in ps or nanosecond If the values units, data_type or baseunit are (manually) set in the import function the corresponding entries in datafram will be overwritten shift_times_by and divide_times_by will be applied if not None (useful to adjust for offset before chirp correction)

Returns

Return type A TA object with all parameter initialized

Examples

Typical useage:

```
>>> import plot_func as pf #import the module and give it a shorter name
>>> ta=pf.TA('gui') #use a GUI to open
>>> ta=pf.TA('sample_1.SIA') #use a filename in the same folder
>>> ta=pf.TA('sample_1.hdf5',path='Data') #use a filename in the folder 'Data'
```

Opening a list of files with external time vector (of the same name) so it looks for a data file "fite1.txt" and a file with the time information "file1.tid"

>>>ta=pf.TA('file1.txt', external_time = 'tid')

__make_standard_parameter()

function that sets the standard parameter. The function takes no input, but we use this docstring to explain the parameter.

Parameters

- **log_scale** (*bool, optional*) – If False (Default), The 2D plots (Matrix) is plotted with a pseudo logarithmic intensity scale. This usually does not give good results unless the intensity scale is symmetric
- **self.cmap** (*matplotlib.cm*) – (Default) standard_map - global parameter cmap is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.
- **self.lintresh** (*float*) – The pseudo logratihmic range "symlog" is used for most time axis. Symlog plots a range around time zero linear and beyond this linear treshold 'lintresh' on a logarithmic scale. (Default) 0.3
- **self.log_fit** – (Default) False

  Transfer all the time-fitting parameters into log-space before the fit
- **self.ignore_time_region** (*None or list (of two floats or of lists)*) – (Default) None cut set a time range with a low and high limit from the fits. (Default) None nothing happens The region will be removed during the fitting process (and will be missing in the fit-result plots)

  Usage single region: [lower region limit,upper region limit]

  use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]

- **self.error_matrix_amplification** – (Default) 10
- **self.rel_wave** (*float or list (of floats)*) – (Default) np.arange(300,1000,100)

  'rel_wave' and 'width' (in the object called 'wavelength_bin' work together for the creation of kinetic plots. When plotting kinetic spectra one line will be plotted for each entrance in the list/vector rel_wave. During object generation the vector np.arange(300,1000,100) is set as standard. Another typical using style would be to define a list of interesting wavelength at which a kinetic development is to be plotted. At each selected wavelength the data between wavelength+ta.wavelength_bin and wavelength-ta.wavelength_bin is averaged for each timepoint returned
- **self.rel_time** (*float or list/vector (of floats)*) – (Default) [0.2,0.3,0.5,1,3,10,30,100,300,1000,3000,9000]

  For each entry in rel_time a spectrum is plotted. If time_width_percent=0 (Default) the nearest measured timepoint is chosen. For other values see 'time_width_percent'
- **self.time_width_percent** (*float*) – (Default) 0 "rel_time" and "time_width_percent" work together for creating spectral plots at specific timepoints. For each entry in rel_time a spectrum is plotted. If however e.g. time_width_percent=10 the region between the timepoint closest to the 1.1 x timepoint and 0.9 x timepoint is averaged and shown (and the legend adjusted accordingly). This is particularly useful for the densly sampled region close to t=0. Typically for a logarithmic recorded kinetics, the timepoints at later times will be further appart than 10 percent of the value, but this allows to elegantly combine values around time=0 for better statistics. This averaging is only applied for the plotting function and not for the fits.
- **self.baseunit** (*str*) – (Default) 'ps'

  baseunit is a neat way to change the unit on the time axis of the plots. (Default) 'ps', but they can be frames or something similarly. This is changing only the label of the axis. During the import there is the option to divide the numbers by a factor. I have also used frames or fs as units. Important is that all time units will be labeled with this unit.
- **self.mod** – (Default) 'exponential'

  This is the default fitting function, in general this is discussed in the fitting section
- **self.scattercut** (*None or iterable (of floats or other iterable, always pairs!)*) – (Default) None

  intented to "cut" one or multiple scatter regions. (if (Default) None nothing happens) If it is set the spectral region between the limits is set to zero. Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **self.bordercut** (*None or iterable (with two floats)*) – (Default) None

  cut spectra at the low and high wavelength limit. (Default) None uses the limits of measurement
- **self.time_bin** (*None or int*) – (Default) None is dividing the points on the time-axis in even bins and averages the found values in between. This is a hard approach that also affects the fits. I do recommend to use this carefully, it is most useful for modulated data. A better choice for transient absorption that only affects the kinetics is 'time_width_percent'
- **self.timelimits** (*None or list (of 2 floats)*) – (Default) None

  cut times at the low and high time limit. (Default) None uses the limits of measurement Important: If either the background or the chirp is to be fit this must include the time before

zero! Useful: It is useful to work on different regions, starting with the longest (then use the ta.Backgound function prior to fit) and expand from there

- **data_type** (*str*) – this is the datatype and effectively the unit put on the intensity axis (Default)'differential Absorption in $mathregular{Delta OD}$
- **self.wave_nm_bin** (*None or float*) – (Default) None

  rebins the original data into even intervals. If set to None the original data will be used. If set to a width (e.g. 2nm), the wavelength axis will be divided into steps of this size and the mean of all measurements in the interval is taken. The re-binning stops as soon as the measured stepsize is wider than given here, then the original bins are used. This function is particularly useful for spectrometer with non-linear dispersion, like a prism in the infrared.
- **self.wavelength_bin** (*float, optional*) – (Default) 10nm the width used in kinetics, see below
- **self.intensity_range** (*None, float or list [of two floats]*) – (Default) None - intensity_range is a general switch that governs what intensity range the plots show. For the 1d plots this is the y-axis for the 2d-plots this is the colour scale. This parameter recognizes three settings. If set to "None" (Default) this uses the minimum and maximum of the data. A single value like in the example below and the intended use is the symmetric scale while a list with two entries an assymmetric scale e.g. intensity_range=3e-3 is converted into intensity_range=[-3e-3,3e-3]
- **self.ds_ori.columns.name** (*str, optional*) – (Default) 'Wavelength in nm'

  This is the general energy axis. here we define it with the unit. Change this to energy for use in e.g x-ray science
- **self.ds_ori.index.name** (*str, optional*) – Standard 'Time in %s' % self.baseunit
- **self.data_type** (*str (optional)*) – self.data_type='diff. Absorption in $mathregular{Delta OD}$'
- **self.fitcoeff** (*list (5 floats)*) – chirp correction polynom
- **self.chirp_file** (*str*) – if there is a file withthe right name write it here, otherwise None
- **self.figure_path** (*str*) – Path for saving figures, if set
- **self.save_figures_to_folder** (*bool*) – if True all figures are automatically saved when any plotfunction is called

Examples

```
>>> ta.bordercut=[350,1200]  #remove all data outside this limit
>>> ta.scattercut=[522,605]  #set data inside this limit to zero
>>> ta.timelimits=[0.2,5000]  #remove all data outside this limit
>>> ta.wave_nm_bin=5  #rebin the data to this width
>>> ta.intensity_range=3e-3  #equivalent to [-3e-3,3e-3]
>>> ta.intensity_range=[-1e-3,3e-3]  #intensity that is plotted in 2d plot and␣
↪y-axis in 1d plots
>>> ta.cmap=matplotlib.cm.prism  #choose different colour map
>>> ta.ignore_time_region=[-0.1,0.1] #ignore -0.1ps to 0.1ps
```

__read_ascii_data(*sep='\t'*, *decimal='.'*, *index_is_energy=False*, *transpose=False*, *sort_indexes=False*, *divide_times_by=None*, *shift_times_by=None*, *external_time=None*, *external_wave=None*, *use_same_name=True*, *correct_ascii_errors=True*, *data_type=None*, *units=None*, *baseunit=None*)

Fancy function that handles the import of pure ascii files.

Parameters

- **sep** (*str (optional)*) – is the separator between different numbers, typical are tap (Backslash t) (Default) ,one or multiple white spaces 'backslash s+' or comma ','.
- **decimal** (*str (optional)*) – sets the ascii symbol that is used for the decimal sign. In most countries this is '.'(Default) but it can be ',' in countries like Sweden or Germany
- **index_is_energy** (*bool (optional)*) – switches if the wavelength is given in nm (Default) or in eV (if True), currently everything is handled as wavelength in nm internally
- **data_type** (*str (optional)*) – data_type is the string that represents the intensity measurements. Usually this contains if absolute of differential data. This is used for the color intensity in the 2d plots and the y-axis for the 1d plots
- **units** (*str (optional)*) – this is used to identify the units on the energy axis and to label the slices, recognized is 'nm', 'eV' and 'keV' but if another unit like 'cm^-1' is used it will state energy in 'cm^-1'. Pleas observe that if you use the index_is_energy switch the program tries to convert this energy into wavelength.
- **baseunit** (*str (optional)*) – this is used to identify the units on the developing/time axis. This is name that is attached to the index of the dataframe. setting this during import is equivalent to ta.baseunit
- **transpose** (*bool (optional)*) – if this switch is False (Default) the wavelength are the columns and the rows the times.
- **sort_indexes** (*bool (optional)*) – For False (Default) I assume that the times and energies are already in a rising order. with this switch, both are sorted again.
- **divide_times_by** (*None or float (optional)*) – here a number can be given that scales the time by an arbitary factor. This is actually dividing the times by this value. Alternatively there is the variable self.baseunit. The latter only affects what is written on the axis, while this value is actually used to scale the times. None (Default) ignores this
- **shift_times_by** (*None, float (optional)*) – This a value by which the time axis is shifted during import. This is a useful option of e.g. the recording software does not compensate for t0 and the data is always shifted. None (Default) ignores this setting
- **external_time** (*None or str (optional)*) – Here a filename extension (string) can be given that contains the time vector. The file is assumed to be at the same path as the data and to contain a single type of separated data without header. If use_same_name = True (default) It assumes that this is the ending for the file. The filename itself is taken from the filename. e.g. if samp1.txt is the filename and external_time='.tid' the program searches samp1.tid for the times. The transpose setting is applied and sets where the times are to be inserted (row or column indexes) If use_same_name = False this should be the file containing the vector for the time (in the same format as the main file)
- **external_wave** (*None or str (optional)*) – Here a filename extension (string) can be given that contains the wavelength vector. If use_same_name = True (default) The file is assumed to be at the same path as the data and to contain a single type of separated data without header. This is the ending for the file. The filename itself is taken from the filename. e.g. if samp1.txt is the filename and external_wave='.wav' then the program searches samp1.wav for the wavelength. The transpose setting is applied and sets where

the wavelength are to be inserted (columns or row indexes) If use_same_name = False this should be a full filename that contains the vector

- **use_same_name** (*bool, optional*) – this switches if the external filename included the loaded filename or is a separate file
- **correct_ascii_errors** (*bool (optional)*) – If True (Default) then the code tries to catch some stuff like double minus signs and double dots

plot_func.**Frame_golay**(*df*, *window=5*, *order=2*, *transpose=False*)

Convenience method that returns the Golay smoothed data for each column (DataFrame) or the series

Parameters

- **df** (*pandas.DataFrame,pandas.Series*) – the DataFrame that has to be interpolated
- **window_size** (*int,optional*) – 5(Default) an integer that indicates how many units are to be interpolated
- **order** (*int, optional*) – 2 (Default) an integer that indicates what orderpolynoninal is to be used to interpolate the points. order=1 effectively turns this into a floating average
- **transpose** (*bool,optional*) – in which orientation is the interpolation to be done. Default is in within the column (usually timepoints)

Returns DataFrame or Series with the interpolation applied

Return type pandas.DataFrame or pandas.Series

plot_func.**GUI_open**(*project_list=None*, *path=None*, *filename_part=None*, *fileending='hdf5'*, *sep='\t'*, *decimal='.'*, *index_is_energy=False*, *transpose=False*, *sort_indexes=False*, *divide_times_by=None*, *shift_times_by=None*, *external_time=None*, *external_wave=None*, *use_same_name=True*, *data_type=None*, *units=None*, *baseunit=None*, *conversion_function=None*)

This Function 1. opens a gui and allows the selection of multiple saved projects, which are returned as a list 2. if given a list of project names opens them 3. if given the word 'all', opens all files in a given folder The general behavior is selected by the first parameter (project_list)

it is designed to open combined files that contain both the wavelength and the time. (e.g. SIA files as recorded by Pascher instruments software) or hdf5 projects saved by this software There are however a lot of additional options to open other ascii type files and adapt their format internally Important, as default the parameter "fileending" selects hdf5 files only, which are used as project files (see *plot_func.TA.Save_project()*) for opening of other files the fileending parameter needs to be changed.

Parameters

- **project_list** (*list (of str) or 'all', optional*) – Give a list of filenames that will be opened and returned as a list of objects if the project list is 'all' then all files in the folder specified in path. The parameter "filename_part" and "fileending" can be used to specify this selection
- **path** (*str or path object (optional)*) – if path is a string without the operation system dependent separator, it is treated as a relative path, e.g. data will look from the working directory in the sub director data. Otherwise this has to be a full path in either strong or path object form.
- **filename_part** (*str, optional*) – This parameter is only used for the option 'all', the (Default) None means do nothing. if a string is given then only files that start with this string will be read.

- **fileending** (*str, optional*) – this string is used to select the filetype that is suppose to open. For the GUI, only these files will be shown, with the option 'all' this selects the files that will be read in the folder, 'hdf5' (Default)
- **sep** (*str (optional)*) – is the separator between different numbers, typical are tap ' ' (Default) ,one or multiple white spaces 's+' or comma ',.
- **decimal** (*str (optional)*) – sets the ascii symbol that is used for the decimal sign. In most countries this is '.'(Default) but it can be ',' in countries like Sweden or Germany
- **index_is_energy** (*bool (optional)*) – switches if the wavelength is given in nm (Default) or in eV (if True), currently everything is handled as wavelength in nm internally
- **transpose** (*bool (optional)*) – if this switch is False (Default) the wavelength are the columns and the rows the times.
- **data_type** (*str (optional)*) – data_type is the string that represents the intensity measurements. Usually this contains if absolute of differential data. This is used for the color intensity in the 2d plots and the y-axis for the 1d plots
- **units** (*str (optional)*) – this is used to identify the units on the energy axis and to label the slices, recognized is 'nm', 'eV' and 'keV' but if another unit like 'cm^-1' is used it will state energy in 'cm^-1'. Pleas observe that if you use the index_is_energy switch the program tries to convert this energy into wavelength.
- **baseunit** (*str (optional)*) – this is used to identify the units on the developing/time axis. This is name that is attached to the index of the dataframe. setting this during import is equivalent to ta.baseunit
- **sort_indexes** (*bool (optional)*) – For False (Default) I assume that the times and energies are already in a rising order. with this switch, both are sorted again.
- **divide_times_by** (*None or float (optional)*) – here a number can be given that scales the time by an arbitary factor. This is actually dividing the times by this value. Alternatively there is the variable self.baseunit. The latter only affects what is written on the axis, while this value is actually used to scale the times. None (Default) ignores this
- **shift_times_by** (*None, float (optional)*) – This a value by which the time axis is shifted during import. This is a useful option of e.g. the recording software does not compensate for t0 and the data is always shifted. None (Default) ignores this setting
- **external_time** (*None or str (optional)*) – Here a filename extension (string) can be given that contains the time vector. The file is assumed to be at the same path as the data and to contain a single type of separated data without header. If use_same_name = True (default) It assumes that this is the ending for the file. The filename itself is taken from the filename. e.g. if samp1.txt is the filename and external_time='.tid' the program searches samp1.tid for the times. The transpose setting is applied and sets where the times are to be inserted (row or column indexes) If use_same_name = False this should be the file containing the vector for the time (in the same format as the main file)
- **external_wave** (*None or str (optional)*) – Here a filename extension (string) can be given that contains the wavelength vector. If use_same_name = True (default) The file is assumed to be at the same path as the data and to contain a single type of separated data without header. This is the ending for the file. The filename itself is taken from the filename. e.g. if samp1.txt is the filename and external_wave='.wav' then the program searches samp1.wav for the wavelength. The transpose setting is applied and sets where the wavelength are to be inserted (columns or row indexes) If use_same_name = False this should be a full filename that contains the vector

- **use_same_name** (*bool, optional*) – this switches if the external filename included the loaded filename or is a separate file True(default)

conversion_function: function(optional) function that receives should have the shape: return pandas Dataframe with time/frames in rows and wavelength/energy in columns, The function is tested to accept (in that order) a my_function(filename, external_time,external_wave), my_function(filename, external_time), my_function(filename,external_wave), my_function(filename) and return: the dataframe ds with the time_axis as rows and spectral axis as columns if the ds.index.name ia not empty the "time axis" is in to that name the spectral axis is in ds.columns.name the return is investigated if it is one, two, or three things. if two are returned then the second must be the name of what the intensity axis is. This value will then be set to data_type if three are returned the third is the baseunit (for the time axis) this allows to use the automatic naming in ps or nanosecond If the values units, data_type or baseunit are (manually) set in the import function the corresponding entries in datafram will be overwritten shift_times_by and divide_times_by will be applied if not None (useful to adjust for offset before chirp correction)

Returns

Return type List of opened TA objects

Examples

```
>>> import plot_func as pf
>>> project_list=pf.GUI_open() #start the GUI to open project Files
>>> project_list=pf.GUI_open(fileending='SIA') #start the GUI to open SIA Files
```

Opening a list of files using the file names

```
>>> project_list=pf.GUI_open(project_list = ['file1.SIA', 'file2.SIA'])
```

Opening all files in the folder "all_data" (relative to where the notebook is with the ending "hdf5"

```
>>> project_list=pf.GUI_open('all',path="all_data")
```

Opening a list of files with external time vector (of the same name) so it looks for a data file "file1.txt" and a file with the time information "file1.tid"

```
>>> project_list=pf.GUI_open(project_list = ['file1.txt', 'file2.txt'], external_
↪time = 'tid')
```

plot_func.**SVD**(*ds*, *times=None*, *scattercut=None*, *bordercut=None*, *timelimits=[0.5, 150]*, *wave_nm_bin=10*, *time_bin=None*, *wavelength_bin=None*, *plotting=True*, *baseunit='ps'*, *title=None*, *ignore_time_region=None*, *cmap=None*, *equal_energy_bin=None*, *data_type='differential Absorption in $\\mathregular{\\Delta OD}$'*)

This function calculates the SVD and plots an overview.

Parameters

- **ds** (*DataFrame*) – This dataframe contains the data to be plotted. It is copied and sliced into the regions defined. The dataframe expects the time to be in Index and the wavelength/energy to be in the columns. The spectra is plotted with a second (energy) axis
- **times** (*None or int*) – are the number of components to be used in the SVD (Default) is None (which is seen as 6)
- **plotting** (*bool*) – if True (Default) the functions plots the SVD, if False it returns the vectors

- **title** (*None or str*) – title to be used on top of each plot The (Default) None triggers self.filename to be used. Setting a specific title as string will be used in all plots. To remove the title all together set an empty string with this command title=""
- **baseunit** (*str*) – baseunit is a neat way to change the unit on the time axis of the plots. (Default) 'ps', but they can be frames or something similarly. This is changing only the label of the axis. During the import there is the option to divide the numbers by a factor. I have also used frames or fs as units. Important is that all time units will be labeled with this unit.
- **timelimits** (*None or list (of 2 floats), optional*) – cut times at the low and high time limit. (Default) [5e-1 , 150] uses the limits of measurement Important: If either the background or the chirp is to be fit this must include the time before zero! Useful: It is useful to work on different regions, starting with the longest (then use the ta.Backgound function prior to fit) and expand from there
- **scattercut** (*None or iterable (of floats or other iterable, always pairs!), optional*) – intented to "cut" one or multiple scatter regions. (if (Default) None nothing happens) If it is set the spectral region between the limits is set to zero. Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **bordercut** (*None or iterable (with two floats), optional*) – cut spectra at the low and high wavelength limit. (Default) None uses the limits of measurement
- **wave_nm_bin** (*None or float, optional*) – rebins the original data into even intervals. If set to None the original data will be used. If set to a width (e.g. 2nm), the wavelength axis will be divided into steps of this size and the mean of all measurements in the interval is taken. The re-binning stops as soon as the measured stepsize is wider than given here, then the original bins are used. This function is particularly useful for spectrometer with non-linear dispersion, like a prism in the infrared. (Default = 10)
- **wavelength_bin** (*float, optional*) – the width used in kinetics, see below (Default) 10nm
- **ignore_time_region** (*None or list (of two floats or of lists), optional*) – cut set a time range with a low and high limit from the fits. (Default) None nothing happens The region will be removed during the fitting process (and will be missing in the fit-result plots) Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **time_bin** (*None or int, optional*) – is dividing the points on the time-axis in even bins and averages the found values in between. This is a hard approach that also affects the fits. I do recommend to use this carefully, it is most useful for modulated data. A better choice for transient absorption that only affects the kinetics is 'time_width_percent'
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that con-

tain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.

plot_func.**Species_Spectra**(*ta=None*, *conc=None*, *das=None*)

useful help function that returns a dictionary that has DataFrame as entries and the names of the components as keys

Parameters

- **ta** (*plot_func.TA object, optional*) – This object should contain a successful fit. The function will cycle through the fitted species and return the matrix that is formed from the dynamics and the species associated spectrum If this given, then "conc" and "das" are ignored. We cycle through the columns of the concentration and take the same column from the das Frame.
- **conc** (*DataFrame, optional*) – Is read only if ta_object is None. This should contain the concentration matrix with the species as as columns
- **das** (*DataFrame, optional*) – This should contain the spectra of the species with one column per spectrum. The position of the columns must match the columns in the conc (at least this is what is assumed)

Examples

dicten=Species_Spectra(ta)

plot_func.**Summarize_scans**(*list_of_scans=None*, *path_to_scans='Scans'*, *list_to_dump='range'*, *window1=None*, *window2=None*, *save_name='combined.SIA'*, *fileending='SIA'*, *filename_part='Scan'*, *return_removed_list=False*, *sep='\t'*, *decimal='.'*, *index_is_energy=False*, *transpose=False*, *sort_indexes=False*, *divide_times_by=None*, *shift_times_by=None*, *external_time=None*, *external_wave=None*, *use_same_name=True*, *return_ds_only=False*, *data_type=None*, *units=None*, *baseunit=None*, *conversion_function=None*, *fitcoeff=None*, *base_TA_object=None*, *value_filter=None*, *zscore_filter_level=None*, *zscore_in_window=True*, *dump_times=True*, *replace_values=None*, *drop_scans=False*)

Average single scans. Uses single scans of the data set and plots them as average after different conditions. Usually one defines one or two windows in which the intensity is integrated. This integrated number is then displayed for each scan in the list. There are different tools to select certain scans that are excluded from the summary. These are defined in the list_to_dump. This list can take either be a list with the number, or a string with the words 'single' or 'range' (see below)

Parameters

- **list_of_scans** (*None, 'gui' or list*) – 'gui' (choose scans via gui)

  None (Default) load scan files from the specified folder (path_to_scans) with the specified file-ending (file_ending), if filename_part is a string than only files with this string in the name are taken

  list of names (strings) loads this list of files list of integers (that will be directly attached to the filename_part) to form the file name

- **path_to_scans** (*None, str or path object, optional*) – specify relative or absolute path to the scan-files (Default:'Scans')

- **file_ending** (*str, optional*) – specify the file extension of the single scan files. The Gui will only show this fileending (Default: '.SIA')
- **filename_part** (*str*) – specify a part of the string included in all scan-files (Default: 'Scan')
- **window1** (*None or list of 4 floats, optional*) – window in time and wavelength over which each scan is averaged.

  window must have the shape [start time, end time, start wavelength, end wavelength] (Default: None)
- **window2** (*list of 4 floats, optional*) – window in time and wavelength over which each scan is averaged.

  window must have the shape [start time, end time, start wavelength, end wavelength] (Default: None) IF not given then only one window will be displayed
- **list_to_dump** (*list, 'single' or 'range', or None, optional*) – takes a list of scans to be excluded from the average, this list can be indexes (order) in which the scans come, or a list of names. if this is given as a list the option "range" is offered, which allows to add additional selection to the cut.

  'single' allows you (in a GUI) to click on single points in plotted window1 or two that is to be removed, useful for spike removal and makes only sense in conjunction with at least a defined window1, if none is defined window1 = [0.5,10,300,1200] will be set automatically. A right click removes the last selection a middle click applies it. An empty middle click (without selecting anything) finishes the gui

  'range' allows you (in a GUI) to click and define regions.

  first left click is the left side of the window, second left click the ride side of the window. Third left click the left side of the second window,... A right click removes the last set point. a middle click finishes and applies the selection

  An **empty middle click** (without selecting anything) finishes the gui

  useful for spike removal and definition of exclusion region (e.g. where the sample died) This makes only sense in conjunction with at least a defined window1 , if none is defined window1 = [0.5,10,300,1200] will be set automatically if None then it is not filtered, but simply returned
- **data_type** (*str (optional)*) – data_type is the string that represents the intensity measurements. Usually this contains if absolute of differential data. This is used for the color intensity in the 2d plots and the y-axis for the 1d plots
- **units** (*str (optional)*) – this is used to identify the units on the energy axis and to label the slices, recognized is 'nm', 'eV' and 'keV' but if another unit like 'cm^-1' is used it will state energy in 'cm^-1'. Pleas observe that if you use the index_is_energy switch the program tries to convert this energy into wavelength.
- **baseunit** (*str (optional)*) – this is used to identify the units on the developing/time axis. This is name that is attached to the index of the dataframe. setting this during import is equivalent to ta.baseunit
- **save_name** (*str, optional*) – specify name for saving the combined scans (Default) 'combined.SIA')
- **return_removed_list** (*bool, optional*) – (Default) False, returns the list of removed scans instead of the averaged data set. (this list could then be given as "list_to_dump" to get the averaged datafile too. If a file name is given for saved file (which is Default) then the file is saved anyways.

- **sep** (*str (optional)*) – is the separator between different numbers, typical are tap (Backslash t) (Default) ,one or multiple white spaces 'backslash s+' or comma ',',
- **decimal** (*str (optional)*) – sets the ascii symbol that is used for the decimal sign. In most countries this is '.'(Default) but it can be ',' in countries like Sweden or Germany
- **index_is_energy** (*bool (optional)*) – switches if the wavelength is given in nm (Default) or in eV (if True), currently everything is handled as wavelength in nm internally
- **transpose** (*bool (optional)*) – if this switch is False (Default) the wavelength are the columns and the rows the times.
- **sort_indexes** (*bool (optional)*) – For False (Default) I assume that the times and energies are already in a rising order. with this switch, both are sorted again.
- **divide_times_by** (*None or float (optional)*) – here a number can be given that scales the time by an arbitary factor. This is actually dividing the times by this value. Alternatively there is the variable self.baseunit. The latter only affects what is written on the axis, while this value is actually used to scale the times. None (Default) ignores this
- **shift_times_by** (*None, float (optional)*) – This a value by which the time axis is shifted during import. This is a useful option of e.g. the recording software does not compensate for t0 and the data is always shifted. None (Default) ignores this setting
- **external_time** (*None or str (optional)*) – Here a filename extension (string) can be given that contains the time vector. The file is assumed to be at the same path as the data and to contain a single type of separated data without header. If use_same_name = True (default) It assumes that this is the ending for the file. The filename itself is taken from the filename. e.g. if samp1.txt is the filename and external_time='.tid' the program searches samp1.tid for the times. The transpose setting is applied and sets where the times are to be inserted (row or column indexes) If use_same_name = False this should be the file containing the vector for the time (in the same format as the main file)
- **external_wave** (*None or str (optional)*) – Here a filename extension (string) can be given that contains the wavelength vector. If use_same_name = True (default) The file is assumed to be at the same path as the data and to contain a single type of separated data without header. This is the ending for the file. The filename itself is taken from the filename. e.g. if samp1.txt is the filename and external_wave='.wav' then the program searches samp1.wav for the wavelength. The transpose setting is applied and sets where the wavelength are to be inserted (columns or row indexes) If use_same_name = False this should be a full filename that contains the vector
- **use_same_name** (*bool, optional*) – this switches if the external filename included the loaded filename or is a separate file True(default)
- **conversion_function** (*function(optional)*) – function that receives should have the shape: return pandas Dataframe with time/frames in rows and wavelength/energy in columns, The function is tested to accept (in that order) a my_function(filename, external_time,external_wave), my_function(filename, external_time), my_function(filename,external_wave), my_function(filename) and return: the dataframe ds with the time_axis as rows and spectral axis as columns if the ds.index.name ia not empty the "time axis" is in to that name the spectral axis is in ds.columns.name the return is investigated if it is one, two, or three things. if two are returned then the second must be the name of what the intensity axis is. This value will then be set to data_type if three are returned the third is the baseunit (for the time axis) this allows to use the automatic naming in ps or nanosecond If the values units, data_type or baseunit are (manually) set in the import function the corresponding entries in datafram will be overwritten shift_times_by and divide_times_by will be applied if not None (useful to adjust for offset before chirp correction)

- **return_ds_only** (*boolean,(optional)*) – if False (Dafault) returns a TA object, otherwise just a DataFrame
- **fitcoeff** (*list, optional*) – these should be the shirp parameteres that are to be applied to all sub scans in the list.
- **base_TA_object** (*TA object, optional*) – instead of the fit_coefficients a Ta object can be provided that is then used as a template, meaning that the scattercuts and bordercuts will be applied before the filtering.
- **value_filter** (*None, float or iterable with two entries, optional*) – if float, everything above that value or below -abs(value_filter) will be filtered replaced with replace_values if iterable, then first is lower treshold, second is upper treshold
- **zscore_filter_level** (*float, optional*) – if this value is set then the manual selection will be replaced with an automatic filter, the following options, dump_times = True, replace_values = None, drop_scans = False decide what is done to the values that are filtered typical value would be e.g. 3
- **zscore_in_window** (*bool,*) – decides if the filter is applied in the windows or over the whole matrix (using statistics on the values)
- **dump_times** (*bool,optional*) – Standard True means that if the zscore filter filters a file the bad time is droped for the average
- **replace_values** (*None, float, optional*) – if dump times is False the values will be replaced with this value. = None, drop_scans = False
- **drop_scans** (*bool,optional*) – Default: = False. This is the harshest type to filter and means that the whole scan is dropped

Returns

- *TA object if return_ds_only is False(Default) averaged dataset (ds) of the selected scans or*
- *(if return_removed_list = True) the list of removed scans.*

Examples

Use use a range to select the rejected scans, look on the scans by integrating the window 0.5ps to 1ps and 450nm to 470nm

```
>>> import plot_func as pf #import the module
>>> window1=[0.5,1,450,470] #define the window
>>> #use a 'GUI' to select the files
>>> pf.Summarize_scans(list_of_scans='gui',window1=window1)
>>> #use all scans in the subfolder scans that have the word 'Scan' in them and use␣
↪the ending 'SIA'
>>> pf.Summarize_scans(path_to_scans = 'Scans', filepart_name = 'Scan',␣
↪window1=window1)
>>> #This does the same as these are standard
>>> pf.Summarize_scans(window1=window1)
```

plot_func.**build_c**(*times*, *mod='paral'*, *pardf=None*, *sub_steps=None*)

Build concentration matrix after model the parameters are: resolution is the width of the rise time (at sigma 50% intensity) This function can also be used to create illustration dynamics. The parallel decays are created explicit, while the consecutive decays are created by sampling the populations at the times given in the first vector and evaluate the progression at a number of substeps defined bu sub_samples (10 by default)

Parameters

- **times** (*np.array*) – array with the times at which the dataframe should be generated. In general the experimental times
- **mod** (*str, optional*) – this selects the model that is used to generate the concentrations.

  1. 'paral' (Default) or 'exponential' both are equivalent
  2. 'consecutive' or 'full_consecutive'

  In 2 the 'consecutive' and 'full_consecutive' are different in that for consecutive the optimization is done using 'exponential' (as it shoudl give the same times) and then only in the last (final) iteration the 'full consecutive' differential equation is used. This has significant speed advantages, but can lead to errors particularly for the very fast times.
- **sub_step** (*int, optional*) – defines how many times the iterative loop (used in consecutive only) is sampling the concentrations between the times given in "times"
- **pardf** (*pd.DataFrame*) – This dataframe must contain the parameter that are used for creating the dynamics the parameter must be named with the index. For the internal functions this must contain these keys:
  - 't0' = zero time, mandatory
  - 'resolution' = instrument response function, mandatory
  - 'background',optional = if this keyword is present a flat constant background is created (=1 over the whole time)
  - 'infinite',optional = if this keyword is present a new non decaying component is formed with the last decay time.
  - 'explicit_GS',optional = if this keyword is present the pulse function (= ground state) is added explicitly to the data
  - 'k0,k1,…' = with increasing integers are taken as decay times. te number of these components is used to determine how many shall be generated.
- **sub_sample** (*bool or integer*) – Default(None) does nothing This switch turns on a additional sampling of the kinetics, meaning that we add the number of steps between each measured steps for the model formation usage: sub_sample=10

Examples

plot_func.**changefonts**(*weight='bold'*, *font='standard'*, *SMALL_SIZE=11*, *MEDIUM_SIZE=13*, *LARGE_SIZE=18*)

Small function that sets the matplotlib font sizes and fonts, written as conveniens to not need to remember all the codes and what is names what. Calling the function will change the matplotlib *rc* settings

Parameters

- **weight** (*str, optional*) – 'bold' or 'normal'
- **font** (*str, optional*) – this is a meta switch that changes the family. known are: 'standard'='DejaVu Sans'

  'arial'='Arial'

  'helvetica'= 'Helvetica'

  'garamond'='Garamond'

  'verdana'='Verdana'

'bookman'='Bookman'

'times'='Times New Roman'

- **SMALL_SIZE** (*int, optional*) – (DEFAULT = 11)

  all written text, legend title and face size

- **MEDIUM_SIZE** (*int, optional*) – (DEFAULT = 13)

  tick size and tick numbers

- **LARGE_SIZE** (*int, optional*) – (DEFAULT = 18)

  axis titles, figure titles, axis labels

plot_func.**err_func**(*paras*, *ds*, *mod='paral'*, *final=False*, *log_fit=False*, *dump_paras=False*, *write_paras=True*, *filename=None*, *ext_spectra=None*, *dump_shapes=False*, *sub_sample=None*, *pulse_sample=None*)

function that calculates and returns the error for the global fit. This function is intended for fitting a single dataset.

Parameters

- **ds** (*DataFrame*) – This dataframe contains the data to be fitted. This has to be shaped as it is intended to (so all shping parameters already applied. The dataframe expects the time to be in Index and the wavelength/energy to be in the columns. The spectra is plotted with a second (energy) axis
- **paras** (*lmfit parameter oject*) – The parameter object that defines what is calculated
- **mod** (*str or function, optional*) – The model selection is depending if it is an internal or external model. The internal functions are triggered by calling their name Two main are currently implemented

  1. 'paral' (Default) or 'exponential'
  2. 'consecutive' or 'full_consecutive'

  In 2 the 'consecutive' and 'full_consecutive' are different in that for consecutive the optimization is done using 'exponential' (as it shoudl give the same times) and then only in the last (final) iteration the 'full consecutive' differential equation is used. This has significant speed advantages, but can lead to errors particularly for the very fast times.

  As external model a function is handed to this parameter, this function must accept the times and an paramater Dataframe and return a DataFrame with the concentrations (similar to build_c)

  for the internal functions: This datafram must contain the parameter that are used for creadting the dynamics the parameter must be named with the index. 't0' = zero time, mandatory 'resolution' = instrument response function, mandatory 'background',optional = if this keyword is present a flat constant background is created (=1 over the whole time) 'infinite',optional = if this keyword is present a new non decaying component is formed with the last decay time. 'explicit_GS' = if this keyword is present thenthe ground state (including the bleach) will be added as a explicit component 'k0,k1,...' = with increasing integers are taken as decay times. te number of these components is used to determine how many shall be generated.

- **final** (*bool, optional*) – this switch decides if just the squared error is returned (for False) (Default) or if the full matrixes are returned, including the r2 are returned.
- **log_fit** (*bool, optional*) – if False (Default) then the parameter are handed to the fitting function as they are, if true then all times are first converted to log space.

- **dump_paras** (*bool, optional*) – (Default) is False, If True creates two files in the working folder, one with the currently used parameter created at the end of each optimisation step, and one with the set of parameter that up to now gave the lowest error. Intented to store the optimisation results if the fit needs to be interrupted (if e.g. Ampgo simply needs to long to optimize.) useful option if things are slow
- **filename** (*None or str, optional*) – Only used in conjunction with 'dump_paras'. The program uses this filename to dump the parameter to disk
- **ext_spectra** (*DataFrame, optional*) – (Default) is None, if given substract this spectra from the DataMatrix using the intensity given in "C(t)" this function will only work for external models. The name of the spectral column must be same as the name of the column used. If not the spectrum will be ignored. The spectrum will be interpolated to the spectral points of the model ds before the substraction. a number of parameters can be defined to aid this process. These parameter are defined as normal parameters. "ext_spectra_scale" multiplies all spectra by this value (e.g. -1 to put the steady state absorption spectra in) "ext_spectra_shift" shifts all spectra by this value to compensate for calibration differences "ext_spectra_guide" (from version 7.1.0) This is a switch, if this keyword is present, then the spectra are used as guides and not exclusively. This means the code will assume that these spectra are correct and substract them, then calulate the difference and return as DAS the provided spectra plus the difference spectra
- **write_paras** (*bool, optional*) – if True(Default) writes the currently varried values to screen

plot_func.**err_func_multi**(*paras*, *mod='paral'*, *final=False*, *log_fit=False*, *multi_project=None*, *unique_parameter=None*, *weights=None*, *dump_paras=False*, *filename=None*, *ext_spectra=None*, *dump_shapes=False*, *same_DAS=False*, *write_paras=True*, *sub_sample=None*, *pulse_sample=None*)

function that calculates and returns the error for the global fit. This function is intended for fitting of multiple datasets

Parameters

- **paras** (*lmfit parameter oject*) – The parameter object that defines what is calculated
- **mod** (*str or function, optional*) – The model selection is depending if it is an internal or external model. The internal functions are triggered by calling their name Two main are currently implemented

  1. 'paral' (Default) or 'exponential'
  2. 'consecutive' or 'full_consecutive'

  In 2 the 'consecutive' and 'full_consecutive' are different in that for consecutive the optimization is done using 'exponential' (as it shoudl give the same times) and then only in the last (final) iteration the 'full consecutive' differential equation is used. This has significant speed advantages, but can lead to errors particularly for the very fast times.

  for the internal functions: This datafram must contain the parameter that are used for creadting the dynamics the parameter must be named with the index. 't0' = zero time, mandatory 'resolution' = instrument response function, mandatory 'background',optional = if this keyword is present a flat constant background is created (=1 over the whole time) 'infinite',optional = if this keyword is present a new non decaying component is formed with the last decay time. 'explicit_GS' = if this keyword is present thenthe ground state (including the bleach) will be added as a explicit component

  'k0,k1,...' = with increasing integers are taken as decay times. te number of these components is used to determine how many shall be generated.

As external model a function is handed to this parameter, this function must accept the times and an paramater Dataframe and return a DataFrame with the concentrations (similar to build_c)

- **final** (*bool, optional*) – this switch decides if just the squared error is returned (for False) (Default) or if the full matrixes are returned, including the r2 are returned.
- **log_fit** (*bool, optional*) – if False (Default) then the parameter are handed to the fitting function as they are, if true then all times are first converted to log space.
- **dump_paras** (*bool, optional*) – (Default) is False, If True creates two files in the working folder, one with the currently used parameter created at the end of each optimisation step, and one with the set of parameter that up to now gave the lowest error. Intented to store the optimisation results if the fit needs to be interrupted (if e.g. Ampgo simply needs to long to optimize.) useful option if things are slow
- **filename** (*None or str, optional*) – Only used in conjunction with 'dump_paras'. The program uses this filename to dump the parameter to disk
- **multi_project** (*None or list (of TA projects), optional*) – This switch is triggering the simultaneous optimisation of multiple datasets. multi_project is as (Default) None. it expects an iterable (typically list) with other TA projects (like ta) that are then optimised with the same parameter. This means that all projects get the same parameter object for each iteration of the fit and return their individual error, which is summed linearly. The "weights" option allows to give each multi_project a specific weight (number) that is multiplied to the error. If the weight object has the same number of items as the multi_project it is assumed that the triggering object (the embedded project) has the weight of 1, otherwise the first weight is for the embedded project. The option 'unique_parameter' takes (a list) of parameter that are not to be shared between the projects (and that are not optimized either) The intended use of this is to give e.g. the pump power for multiple experiments to study non linear behaviour. Returned will be only the parameter set for the optimium combination of all parameter. Internally, we iterate through the projects and calculate for each project the error for each iteration. Important to note is that currently this means that each DAS/SAS is calculated independently! For performing the same calculation with a single DAS, the Matrixes need to be concatenated before the run and an external function used to create a combined model. As this is very difficult to implement reliably For general use (think e.g. different pump wavelength) this has to be done manually.
- **unique_parameter** (*None or str or list (of strings), optional*) – only used in conjunction with 'multi_project', it takes (a list) of parameter that are not to be shared between the projects (and that are not optimized either) The intended use of this is to give e.g. the pump power for multiple experiments to study non linear behaviour. (Default) None
- **same_DAS** (*bool,optional*) – changes the fit behavior and uses the same DAS for the optimization. This means that the ds are stacked before the fill int rounds
- **weights** (*list of floats, optional*) – only used in conjunction with 'multi_project'. The "weights" option allows to give each multi_project a specific weight (number) that is multiplied to the error. If the weight object has the same number of items as the 'multi_project' it is assumed that ta (the embedded project) has the weight of 1, otherwise the first weight is for the embedded object
- **ext_spectra** (*DataFrame, optional*) – (Default) is None, if given substract this spectra from the DataMatrix using the intensity given in "C(t)" this function will only work for external models. The name of the spectral column must be same as the name of the column used. If not the spectrum will be ignored. The spectrum will be interpolated to the spectral points of the model ds before the substraction. a number of parameters can be defined to aid this process. These parameter are defined as normal parameters. "ext_spectra_scale"

multiplies all spectra by this value (e.g. -1 to put the steady state absorption spectra in) "ext_spectra_shift" shifts all spectra by this value to compensate for calibration differences "ext_spectra_guide" (from version 7.1.0) This is a switch, if this keyword is present, then the spectra are used as guides and not exclusively. This means the code will assume that these spectra are correct and substract them, then calulate the difference and return as DAS the provided spectra plus the difference spectra

- **write_paras** (*bool, optional*) – if True(Default) writes the currently varried values to screen

plot_func.**fill_int**(*ds*, *c*, *final=True*, *baseunit='ps'*, *return_shapes=False*)

solving the intensity an equation_way, takes the target dataframe and the concentration frame prepares the matrixes(c) the tries to solve this equation system using eps=np.linalg.lstsq(AA,Af,rcond=-1)[0] if failes it returns a dictionary with 1000 as error (only entry) if successful it returns a dictionary that contains the fit_error = (AE**2).sum() with AE beeing the difference of measured and calcuated matrix

Parameters

- **ds** (*DataFrame*) – DataFrame to be fitted
- **c** (*DataFrame*) – DataFrame oontaining the concentration matrix (the concentrations as with the times as index. Each different species has a column with the species name as column name
- **final** (*bool,optional*) – if True (Default) the complete solutions will be attached otherwise only the error is attached
- **baseunit** (*str,optional*) – this string is used as unit for the time axis
- **return_shapes** (*bool,optional*) – Default = False, if True, then the concentrations and spectra are added to the re (even if not final)

Returns

re – the dictionary "re" attached to the object containing all the matrixes and parameter.

if "final" is True:

- "A" Shaped measured Matrix
- "AC" Shaped calculated Matrix
- "AE" Difference between A and AC = linear error
- "DAC" DAS or SAS, labeled after the names given in the function (the columns of c) Care must be taken that this mesured intensity is C * DAS, the product. For exponential model the concentrations are normalized
- "c" The Concentrations (meaning the evolution of the concentrations over time. Care must be taken that this mesured intensity is C * DAS, the product. For exponential model the concentrations are normalized
- "error" is the S2, meaning AE**2.sum().sum()

else:

- "error" is the S2, meaning AE**2.sum()

Return type dict

plot_func.**norm**(*df*)

Min max norming of a dataframe

plot_func.**plot1d**(*ds=None*, *wavelength=None*, *width=None*, *ax=None*, *subplot=False*, *title=None*, *intensity_range=None*, *baseunit='ps'*, *timelimits=None*, *scattercut=None*, *bordercut=None*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *plot_type='symlog'*, *lintresh=0.1*, *text_in_legend=None*, *lines_are='smoothed'*, *color_offset=0*, *ignore_time_region=None*, *linewidth=1*, *data_type='differential Absorption in $\\mathregular{\\Delta OD}$'*, *units='nm'*, *from_fit=False*)

Plots the single line kinetic for specific wavelength given with the parameter wavelength.

Parameters

- **ds** (*DataFrame*) – This dataframe contains the data to be plotted. It is copied and sliced into the regions defined. The dataframe expects the time to be in Index and the wavelength/energy to be in the columns. The spectra is plotted with a second (energy) axis
- **wavelength** (*float or list (of floats)*) – wavelength is in the object called "rel_wave" and works with "width" (in the object called "wavelength_bin") together for the creation of kinetic plots. When plotting kinetic spectra one line will be plotted for each entrance in the list/vector rel_wave. During object generation the vector np.arange(300,1000,100) is set as standard. Another typical using style would be to define a list of interesting wavelength at which a kinetic development is to be plotted. At each selected wavelength the data between wavelength+ta.wavelength_bin and wavelength-ta.wavelength_bin is averaged for each timepoint returned
- **data_type** (*str*) – this is the datatype and effectively the unit put on the intensity axis (Default)'differential Absorption in $mathregular{Delta OD}$
- **width** (*float, optional*) – the width used in kinetics, see below (Default) 10nm
- **ax** (*None, matplotlib axis object optional*) – If None (Default) a new plot is is created and a new axis, otherwise ax needs to be Matplotlib Axis
- **subplot** (*bool, optional*) – If False (Default) axis labels and such are set. If True, we plot into the same axis and do not set labels
- **text_in_legend** (*None, str, optional*) – extra text to be put into the legend (above the lines)
- **lines_are** (*str, optional*) – Depending on this parameter the plots contain: 'smoothed' = data lines of golay smoothed data (Default) 'data' = dots are data, 'fitted' = not data, just lines shown
- **title** (*None or str*) – title to be used on top of each plot The (Default) None triggers self.filename to be used. Setting a specific title as string will be used in all plots. To remove the title all together set an empty string
- **linewidth** (*float, optional*) – linewidht to be used for plotting
- **intensity_range** (*None, float or list [of two floats]*) – intensity_range is a general switch that governs what intensity range the plots show. For the 1d plots this is the y-axis for the 2d-plots this is the colour scale. This parameter recognizes three settings. If set to "None" (Default) this uses the minimum and maximum of the data. A single value like in the example below and the intended use is the symmetric scale while a list with two entries an assymmetric scale e.g. intensity_range=3e-3 is converted into intensity_range=[-3e-3,3e-3]
- **baseunit** (*str*) – baseunit is a neat way to change the unit on the time axis of the plots. (Default) 'ps', but they can be frames or something similarly. This is changing only the label of the axis. During the import there is the option to divide the numbers by a factor. I have also used frames or fs as units. Important is that all time units will be labeled with this unit.
- **timelimits** (*None or list (of 2 floats), optional*) – cut times at the low and high time limit. (Default) None uses the limits of measurement Important: If either the

background or the chirp is to be fit this must include the time before zero! Useful: It is useful to work on different regions, starting with the longest (then use the ta.Backgound function prior to fit) and expand from there

- **scattercut** (*None or iterable (of floats or other iterable, always pairs!), optional*) – intented to "cut" one or multiple scatter regions. (if (Default) None nothing happens) If it is set the spectral region between the limits is set to zero. Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **bordercut** (*None or iterable (with two floats), optional*) – cut spectra at the low and high wavelength limit. (Default) None uses the limits of measurement
- **ignore_time_region** (*None or list (of two floats or of lists), optional*) – cut set a time range with a low and high limit from the fits. (Default) None nothing happens The region will be removed during the fitting process (and will be missing in the fit-result plots) Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.
- **color_offset** (*int, optional*) – At the (Default) 0 the colours are chose from the beginning, for a larger value Color_offset colors are skipped. Usually only used if multiple plots are created, and the data/or fit is only shown for some of them.
- **plot_type** (*None or str*) – is a general setting that can influences what time axis will be used for the plots. "symlog" (linear around zero and logarithmic otherwise) "lin" and "log" are valid options.
- **lintresh** (*float*) – The pseudo logratihmic range "symlog" is used for most time axis. Symlog plots a range around time zero linear and beyond this linear treshold 'lintresh' on a logarithmic scale. (Default) 0.3
- **from_fit** (*bool optional*) – i needed this swtich to avoid re-slicing of data in spectal axis for equal energy bins

plot_func.**plot2d**(*ds*, *ax=None*, *title=None*, *intensity_range=None*, *baseunit='ps'*, *timelimits=None*, *scattercut=None*, *bordercut=None*, *wave_nm_bin=None*, *ignore_time_region=None*, *time_bin=None*, *log_scale=False*, *plot_type='symlog'*, *lintresh=1*, *wavelength_bin=None*, *levels=256*, *use_colorbar=True*, *cmap=None*, *data_type='differential Absorption in $\\mathregular{\\Delta OD}$'*, *equal_energy_bin=None*, *from_fit=False*)

function for plotting matrix of TA data.

Parameters

- **ds** (*DataFrame*) – This dataframe contains the data to be plotted. It is copied and sliced into the regions defined. The dataframe expects the time to be in Index and the wavelength/energy to be in the columns. The spectra is plotted with a second (energy) axis
- **ax** (*None, matplotlib axis object optional*) – If None (Default) a new plot is is created and a new axis, otherwise ax needs to be Matplotlib Axis
- **data_type** (*str*) – this is the datatype and effectively the unit put on the intensity axis (Default)'differential Absorption in $mathregular{Delta OD}$
- **title** (*None or str*) – title to be used on top of each plot The (Default) None triggers self.filename to be used. Setting a specific title as string will be used in all plots. To remove the title all together set an empty string with this command title=""
- **intensity_range** (*None, float or list [of two floats]*) – intensity_range is a general switch that governs what intensity range the plots show. For the 1d plots this is the y-axis for the 2d-plots this is the colour scale. This parameter recognizes three settings. If set to "None" (Default) this uses the minimum and maximum of the data. A single value like in the example below and the intended use is the symmetric scale while a list with two entries an assymmetric scale e.g. intensity_range=3e-3 is converted into intensity_range=[-3e-3,3e-3]
- **baseunit** (*str*) – baseunit is a neat way to change the unit on the time axis of the plots. (Default) 'ps', but they can be frames or something similarly. This is changing only the label of the axis. During the import there is the option to divide the numbers by a factor. I have also used frames or fs as units. Important is that all time units will be labeled with this unit.
- **timelimits** (*None or list (of 2 floats), optional*) – cut times at the low and high time limit. (Default) None uses the limits of measurement Important: If either the background or the chirp is to be fit this must include the time before zero! Useful: It is useful to work on different regions, starting with the longest (then use the ta.Backgound function prior to fit) and expand from there
- **scattercut** (*None or iterable (of floats or other iterable, always pairs!), optional*) – intented to "cut" one or multiple scatter regions. (if (Default) None nothing happens) If it is set the spectral region between the limits is set to zero. Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **bordercut** (*None or iterable (with two floats), optional*) – cut spectra at the low and high wavelength limit. (Default) None uses the limits of measurement
- **wave_nm_bin** (*None or float, optional*) – rebins the original data into even intervals. If set to None the original data will be used. If set to a width (e.g. 2nm), the wavelength axis will be divided into steps of this size and the mean of all measurements in the interval is taken. The re-binning stops as soon as the measured stepsize is wider than given here, then the original bins are used. This function is particularly useful for spectrometer with non-linear dispersion, like a prism in the infrared.
- **equal_energy_bin** (*None or float(optional)*) – if this is set the wave_nm_bin is ignored and the data is rebinned into equal energy bins (based upon that the data is in nm. If dual axis is on then the lower axis is energy and the upper is wavelength

- **ignore_time_region** (*None or list (of two floats or of lists), optional*) – cut set a time range with a low and high limit from the fits. (Default) None nothing happens The region will be removed during the fitting process (and will be missing in the fit-result plots) Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **time_bin** (*None or int, optional*) – is dividing the points on the time-axis in even bins and averages the found values in between. This is a hard approach that also affects the fits. I do recommend to use this carefully, it is most useful for modulated data. A better choice for transient absorption that only affects the kinetics is 'time_width_percent'
- **log_scale** (*bool, optional*) – If True (Default), The 2D plots (Matrix) is plotted with a pseudo logarithmic intensity scale. This usually does not give good results unless the intensity scale is symmetric
- **plot_type** (*None or str*) – is a general setting that can influences what time axis will be used for the plots. "symlog" (linear around zero and logarithmic otherwise) "lin" and "log" are valid options.
- **lintresh** (*float*) – The pseudo logratihmic range "symlog" is used for most time axis. Symlog plots a range around time zero linear and beyond this linear treshold 'lintresh' on a logarithmic scale. (Default) 0.3
- **wavelength_bin** (*float, optional*) – the width used in kinetics, see below (Default) 10nm
- **levels** (*int, optional*) – how many different colours to use in the description. less makes for more contrast but less intensity details (Default) 256
- **use_colorbar** (*bool, optional*) – if True (Default) a colour bar is added to the 2d plot for intensity explanation, switch mostely used for creating multiple plots
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.
- **from_fit** (*bool optional*) – it needed this swtich to avoid re-slicing of data in spectal axis for equal energy bins

plot_func.**plot2d_fit**(*re*, *error_matrix_amplification=5*, *use_images=True*, *patches=False*, *title=None*, *intensity_range=None*, *baseunit='ps'*, *timelimits=None*, *scattercut=None*, *bordercut=None*, *wave_nm_bin=None*, *ignore_time_region=None*, *time_bin=None*, *log_scale=False*, *scale_type='symlog'*, *lintresh=1*, *wavelength_bin=None*, *levels=256*, *plot_with_colorbar=True*, *cmap=None*, *data_type='differential Absorption in $\\mathregular{\\Delta OD}$'*, *equal_energy_bin=None*)

Plots the fit output as a single plot with meas,fitted and difference. The differnece used err_matrix_amplification as a factor. patches moves the labels from the title into white patches in the top of the figure

Parameters

- **re** (*dict*) – Dictionary that contains the fit results and specific the dataframes A, AC and AE
- **error_matrix_amplification** (*int, optional*) – the error matrix AE is multiplied by this factor for the plot.
- **use_images** (*bool:*) – (Default)True converts the matrix into images, to reduce the filesize.
- **data_type** (*str*) – this is the datatype and effectively the unit put on the intensity axis (Default)'differential Absorption in $mathregular{Delta OD}$
- **patches** (*bool, optional*) – If False (Default) the names "measured" "fitted" "difference" will be placed above the images. If True, then they will be included into the image (denser)
- **title** (*None or str*) – title to be used on top of each plot The (Default) None triggers self.filename to be used. Setting a specific title as string will be used in all plots. To remove the title all together set an empty string with this command title=""
- **intensity_range** (*None, float or list [of two floats]*) – intensity_range is a general switch that governs what intensity range the plots show. For the 1d plots this is the y-axis for the 2d-plots this is the colour scale. This parameter recognizes three settings. If set to "None" (Default) this uses the minimum and maximum of the data. A single value like in the example below and the intended use is the symmetric scale while a list with two entries an assymmetric scale e.g. intensity_range=3e-3 is converted into intensity_range=[-3e-3,3e-3]
- **baseunit** (*str*) – baseunit is a neat way to change the unit on the time axis of the plots. (Default) 'ps', but they can be frames or something similarly. This is changing only the label of the axis. During the import there is the option to divide the numbers by a factor. I have also used frames or fs as units. Important is that all time units will be labeled with this unit.
- **timelimits** (*None or list (of 2 floats), optional*) – cut times at the low and high time limit. (Default) None uses the limits of measurement Important: If either the background or the chirp is to be fit this must include the time before zero! Useful: It is useful to work on different regions, starting with the longest (then use the ta.Backgound function prior to fit) and expand from there
- **scattercut** (*None or iterable (of floats or other iterable, always pairs!), optional*) – intented to "cut" one or multiple scatter regions. (if (Default) None nothing happens) If it is set the spectral region between the limits is set to zero. Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **bordercut** (*None or iterable (with two floats), optional*) – cut spectra at the low and high wavelength limit. (Default) None uses the limits of measurement
- **wave_nm_bin** (*None or float, optional*) – rebins the original data into even intervals. If set to None the original data will be used. If set to a width (e.g. 2nm), the wavelength axis will be divided into steps of this size and the mean of all measurements in the interval

is taken. The re-binning stops as soon as the measured stepsize is wider than given here, then the original bins are used. This function is particularly useful for spectrometer with non-linear dispersion, like a prism in the infrared.

- **equal_energy_bin** (*None or float(optional)*) – if this is set the wave_nm_bin is ignored and the data is rebinned into equal energy bins (based upon that the data is in nm. If dual axis is on then the lower axis is energy and the upper is wavelength
- **ignore_time_region** (*None or list (of two floats or of lists), optional*) – cut set a time range with a low and high limit from the fits. (Default) None nothing happens The region will be removed during the fitting process (and will be missing in the fit-result plots) Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **time_bin** (*None or int, optional*) – is dividing the points on the time-axis in even bins and averages the found values in between. This is a hard approach that also affects the fits. I do recommend to use this carefully, it is most useful for modulated data. A better choice for transient absorption that only affects the kinetics is 'time_width_percent'
- **log_scale** (*bool, optional*) – If True (Default), The 2D plots (Matrix) is plotted with a pseudo logarithmic intensity scale. This usually does not give good results unless the intensity scale is symmetric
- **Scale_type** (*None or str*) – is a general setting that can influences what time axis will be used for the plots. "symlog" (linear around zero and logarithmic otherwise) "lin" and "log" are valid options.
- **lintresh** (*float*) – The pseudo logratihmic range "symlog" is used for most time axis. Symlog plots a range around time zero linear and beyond this linear treshold 'lintresh' on a logarithmic scale. (Default) 0.3
- **wavelength_bin** (*float, optional*) – the width used in kinetics, see below (Default) 10nm
- **levels** (*int, optional*) – how many different colours to use in the description. less makes for more contrast but less intensity details (Default) 256
- **plot_with_colorbar** (*bool, optional*) – if True (Default) a colour bar is added to the 2d plot for intensity explanation, switch mostely used for creating multiple plots
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.

plot_func.**plot_fit_output**(*re, ds, cmap=<matplotlib.colors.LinearSegmentedColormap object>, plotting=range(0, 6), title=None, path=None, filename=None, f='standard', intensity_range=0.01, baseunit='ps', timelimits=None, scattercut=None, bordercut=None, error_matrix_amplification=20, wave_nm_bin=5, rel_wave=None, width=10, rel_time=[1, 5, 10], time_width_percent=10, ignore_time_region=None, save_figures_to_folder=True, log_fit=False, mod=None, subplot=False, color_offset=0, log_scale=True, savetype='png', evaluation_style=False, lintresh=1, scale_type='symlog', patches=False, print_click_position=False, data_type='differential Absorption in $\\mathregular{\\Delta OD}$', plot_second_as_energy=True, units='nm', equal_energy_bin=None*)

Purly manual function that plots all the fit output figures. Quite cumbersome, but offers a lot of manual options. The figures can be called separately or with a list of plots. e.g. range(6) call plots 0-5 Manual plotting of certain type: This is a wrapper function that triggers the plotting of all the fitted plots. The parameter in this plot call are to control the general look and features of the plot. Which plots are printed is defined by the command (plotting) The plots are generated from the fitted Matrixes and as such only will work after a fit was actually completed (and the "re" dictionary attached to the object.) In all plots the RAW data is plotted as dots and the fit with lines

Contents of the plots

0. DAC contains the assigned spectra for each component of the fit. For a modelling with independent exponential decays this corresponds to the "Decay Associated Spectra" (DAS). For all other models this contains the "Species Associated Spectra" (SAS). According to the model the separate spectra are labeled by time (process) or name, if a name is associated in the fitting model. The spectra are shown in the extracted strength in the right pane and normalized in the left. Extracted strength means that the measured spectral strength is the intensity (concentration matrix) times this spectral strength. As the concentration maxima for all DAS are 1 this corresponds to the spectral strength for the DAS. (please see the documentation for the fitting algorithm for further details)
1. summed intensity. All wavelength of the spectral axis are summed for data and fit. The data is plotted in a number of ways vs linear and logarithmic axis. This plot is not ment for publication but very useful to evaluate the quality of a fit.
2. plot kinetics for selected wavelength (see corresponding RAW plot)
3. plot spectra at selected times (see corresponding RAW plot)
4. plots matrix (measured, modelled and error Matrix). The parameter are the same as used for the corresponding RAW plot with the addition of "error_matrix_amplification" which is a scaling factor multiplied onto the error matrix. I recommend to play with different "cmap", "log_scale" and "intensity_scale" to create a pleasing plot
5. concentrations. In the progress of the modelling/fitting a matrix is generated that contains the relative concentrations of the species modelled. This plot is showing the temporal development of these species. Further details on how this matrix is generated can be found in the documentation of the fitting function. The modeled spectra are the convolution of these vectors (giving the time-development) and the DAS/SAS (giving the spectral development).

Parameters

- **ds** (*DataFrame*) – This dataframe contains the data to be plotted. It is copied and sliced into the regions defined. The dataframe expects the time to be in Index and the wavelength/energy to be in the columns. The spectra is plotted with a second (energy) axis
- **re** (*dict*) – Dictionary that contains the fit results and specific the dataframes A, AC and AE
- **data_type** (*str*) – this is the datatype and effectively the unit put on the intensity axis (Default)'differential Absorption in $mathregular{Delta OD}$

- **error_matrix_amplification** (*int, optional*) – the error matrix AE is multiplied by this factor for the plot.
- **plotting** (*int or iterable (of integers), optional*) – This parameter determines which figures are plotted the figures can be called separately with plotting = 1 or with a list of plots (Default) e.g.~plotting=range(6) calls plots 0,1,2,3,4,5 The plots have the following numbers: 0 - DAS or SAS 1 - summed intensity 2 - Kinetics 3 - Spectra 4 - Matrixes 5 - Concentrations (the c-object) The plotting takes all parameter from the "ta" object unless otherwise specified
- **path** (*None, str or path object, optional*) – This defines where the files are saved if the safe_figures_to_folder parameter is True, quite useful if a lot of data sets are to be printed fast. If a path is given, this is used. If a string like the (Default) "result_figures" is given, then a subfolder of this name will be used (an generated if necessary) relative to self.path. Use and empty string to use the self.path If set to None, the location of the plot_func will be used and a subfolder with title "result_figures" be generated here
- **savetype** (*str or iterable (of str), optional*) – matplotlib allows the saving of figures in various formats. (Default) "png", typical and recommendable options are "svg" and "pdf".
- **evaluation_style** (*bool, optional*) – True (Default = False) adds a lot of extra information in the plot
- **title** (*None or str, optional*) – "title=None" is in general the filename that was loaded. Setting a specific title will be used in all plots. To remove the title all together set an empty string with title=""
- **scale_type** (*str, optional*) – refers to the time-axis and takes, 'symlog' (Default)(linear around zero and logarithmic otherwise) and 'lin' for linear and 'log' for logarithmic, switching all the time axis to this type
- **patches** (*bool, optional*) – If False (Default) the names "measured" "fitted" "difference" will be placed above the images. If True, then they will be included into the image (denser)
- **filename** (*str, optional*) – offers to replace the base-name used for all plots (to e.g.~specify what sample was used). if (Default) None is used, the self.filename is used as a base name. The filename plays only a role during saving, as does the path and savetype

save_figures_to_folder [bool, optional] (Default) is True, if True the Figures are automatically saved

log_scale [bool, optional] If True (Default), The 2D plots (Matrix) is plotted with a pseudo logarithmic intensity scale. This usually does not give good results unless the intensity scale is symmetric

subplot [bool, optional] If False (Default) axis labels and such are set. If True, we plot into the same axis and do not set labels

color_offset [int, optional] At the (Default) 0 the colours are chose from the beginning, for a larger value Color_offset colors are skipped. Usually only used if multiple plots are created, and the data/or fit is only shown for some of them.

lintresh [float] The pseudo logratihmic range "symlog" is used for most time axis. Symlog plots a range around time zero linear and beyond this linear treshold 'lintresh' on a logarithmic scale. (Default) 1

rel_time [float or list/vector (of floats), optional] For each entry in rel_time a spectrum is plotted. If time_width_percent=0 (Default) the nearest measured timepoint is chosen. For other values see 'time_width_percent'

time_width_percent [float] "rel_time" and "time_width_percent" work together for creating spectral plots at specific timepoints. For each entry in rel_time a spectrum is plotted. If however e.g.

time_width_percent=10 the region between the timepoint closest to the 1.1 x timepoint and 0.9 x timepoint is averaged and shown (and the legend adjusted accordingly). This is particularly useful for the densly sampled region close to t=0. Typically for a logarithmic recorded kinetics, the timepoints at later times will be further appart than 10 percent of the value, but this allows to elegantly combine values around time=0 for better statistics. This averaging is only applied for the plotting function and not for the fits.

ignore_time_region [None or list (of two floats or of lists), optional] cut set a time range with a low and high limit from the fits. (Default) None nothing happens The region will be removed during the fitting process (and will be missing in the fit-result plots) Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]

width [float, optional] the width used in kinetics, see below (Default) 10nm

rel_wave [float or list (of floats), optional] 'rel_wave' and 'width' (in the object called 'wavelength_bin' work together for the creation of kinetic plots. When plotting kinetic spectra one line will be plotted for each entrance in the list/vector rel_wave. During object generation the vector np.arange(300,1000,100) is set as standard. Another typical using style would be to define a list of interesting wavelength at which a kinetic development is to be plotted. At each selected wavelength the data between wavelength+ta.wavelength_bin and wavelength-ta.wavelength_bin is averaged for each timepoint returned

timelimits [None or list (of 2 floats), optional] cut times at the low and high time limit. (Default) None uses the limits of measurement Important: If either the background or the chirp is to be fit this must include the time before zero! Useful: It is useful to work on different regions, starting with the longest (then use the ta.Backgound function prior to fit) and expand from there

scattercut [None or iterable (of floats or other iterable, always pairs!), optional] intented to "cut" one or multiple scatter regions. (if (Default) None nothing happens) If it is set the spectral region between the limits is set to zero. Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]

bordercut [None or iterable (with two floats), optional] cut spectra at the low and high wavelength limit. (Default) None uses the limits of measurement

wave_nm_bin [None or float, optional] rebins the original data into even intervals. If set to None the original data will be used. If set to a width (e.g. 2nm), the wavelength axis will be divided into steps of this size and the mean of all measurements in the interval is taken. The re-binning stops as soon as the measured stepsize is wider than given here, then the original bins are used. This function is particularly useful for spectrometer with non-linear dispersion, like a prism in the infrared.

equal_energy_bin [None or float(optional)] if this is set the wave_nm_bin is ignored and the data is rebinned into equal energy bins (based upon that the data is in nm. If dual axis is on then the lower axis is energy and the upper is wavelength

intensity_range [None, float or list [of two floats]] intensity_range is a general switch that governs what intensity range the plots show. For the 1d plots this is the y-axis for the 2d-plots this is the colour scale. This parameter recognizes three settings. If set to "None" (Default) this uses the minimum and maximum of the data. A single value like in the example below and the intended use is the symmetric scale while a list with two entries an assymmetric scale e.g. intensity_range=3e-3 is converted into intensity_range=[-3e-3,3e-3]

baseunit [str] baseunit is a neat way to change the unit on the time axis of the plots. (Default) 'ps', but they can be frames or something similarly. This is changing only the label of the axis. During the import there is the option to divide the numbers by a factor. I have also used frames or fs as units. Important is that all time units will be labeled with this unit.

f [str] f is a replacement title that is set instead of the title. mainly used to have some options (Default) is 'standard'

log_fit [bool, optional] (default)= False Used for legend generation, tells if the fit was in log or lin space

mod [str, optional] Used for legend generation, tells what model was used for fitting

cmap [None or matplotlib color map, optional] is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.

print_click_position [bool, optional] if True then the click position is printed for the spectral plots

Examples

```
>>> ta.plot_fit_output(ta.re,ta.ds)
```

plot_func.**plot_raw**(*ds=None*, *plotting=range(0, 4)*, *title=None*, *intensity_range=0.01*, *baseunit='ps'*, *timelimits=None*, *scattercut=None*, *bordercut=None*, *wave_nm_bin=None*, *width=10*, *rel_wave=array([400, 500, 600, 700, 800])*, *rel_time=[1, 5, 10]*, *time_width_percent=10*, *ignore_time_region=None*, *time_bin=None*, *cmap=None*, *color_offset=0*, *log_scale=True*, *plot_type='symlog'*, *lintresh=0.3*, *times=None*, *save_figures_to_folder=False*, *savetype='png'*, *path=None*, *filename=None*, *print_click_position=False*, *data_type='differential Absorption in $\\mathregular{\\Delta OD}$'*, *plot_second_as_energy=True*, *units='nm'*, *return_plots=False*, *equal_energy_bin=None*)

This is the extended plot function, for convenient object based plotting see TA.Plot_RAW This function plotts of various RAW (non fitted) plots. Based on the DataFrame ds a number of cuts are created using the shaping parameters explained below. In all plots the RAW data is plotted as dots and interpolated with lines (using Savitzky-Golay window=5, order=3 interpolation).

Parameters

- **ds** (*DataFrame*) – This dataframe contains the data to be plotted. It is copied and sliced into the regions defined. The dataframe expects the time to be in Index and the wavelength/energy to be in the columns. The spectra is plotted with a second (energy) axis
- **plotting** (*int or iterable (of integers), optional*) – This parameter determines which figures are plotted the figures can be called separately with plotting = 1 or with a list of plots (Default) e.g.plotting=range(4) calls plots 0,1,2,3 The plots have the following numbers:

  0. Matrix
  1. Kinetics
  2. Spectra
  3. SVD

- **title** (*None or str*) – title to be used on top of each plot The (Default) None triggers self.filename to be used. Setting a specific title as string will be used in all plots. To remove the title all together set an empty string with this command title=""

- **intensity_range** (*None, float or list [of two floats]*) – intensity_range is a general switch that governs what intensity range the plots show. For the 1d plots this is the y-axis for the 2d-plots this is the colour scale. This parameter recognizes three settings. If set to "None" (Default) this uses the minimum and maximum of the data. A single value like in the example below and the intended use is the symmetric scale while a list with two entries an assymmetric scale e.g. intensity_range=3e-3 is converted into intensity_range=[-3e-3,3e-3]
- **data_type** (*str*) – this is the datatype and effectively the unit put on the intensity axis (Default)'differential Absorption in $mathregular{Delta OD}$
- **baseunit** (*str*) – baseunit is a neat way to change the unit on the time axis of the plots. (Default) "ps", but they can be frames or something similarly. This is changing only the label of the axis. During the import there is the option to divide the numbers by a factor. I have also used frames or fs as units. Important is that all time units will be labeled with this unit.
- **timelimits** (*None or list (of 2 floats), optional*) – cut times at the low and high time limit. (Default) None uses the limits of measurement Important: If either the background or the chirp is to be fit this must include the time before zero! Useful: It is useful to work on different regions, starting with the longest (then use the ta.Backgound function prior to fit) and expand from there
- **scattercut** (*None or iterable (of floats or other iterable, always pairs!), optional*) – intented to "cut" one or multiple scatter regions. (if (Default) None nothing happens) If it is set the spectral region between the limits is set to zero. Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **bordercut** (*None or iterable (with two floats), optional*) – cut spectra at the low and high wavelength limit. (Default) None uses the limits of measurement
- **wave_nm_bin** (*None or float, optional*) – rebins the original data into even intervals. If set to None the original data will be used. If set to a width (e.g. 2nm), the wavelength axis will be divided into steps of this size and the mean of all measurements in the interval is taken. The re-binning stops as soon as the measured stepsize is wider than given here, then the original bins are used. This function is particularly useful for spectrometer with non-linear dispersion, like a prism in the infrared.
- **equal_energy_bin** (*None or float(optional)*) – if this is set the wave_nm_bin is ignored and the data is rebinned into equal energy bins (based upon that the data is in nm. If dual axis is on then the lower axis is energy and the upper is wavelength
- **width** (*float, optional*) – the width used in kinetics, see below (Default) 10nm
- **rel_wave** (*float or list (of floats), optional*) – "rel_wave" and "width" (in the object called "wavelength_bin" work together for the creation of kinetic plots. When plotting kinetic spectra one line will be plotted for each entrance in the list/vector rel_wave. During object generation the vector np.arange(300,1000,100) is set as standard. Another typical using style would be to define a list of interesting wavelength at which a kinetic development is to be plotted. At each selected wavelength the data between wavelength+ta.wavelength_bin and wavelength-ta.wavelength_bin is averaged for each timepoint returned
- **rel_time** (*float or list/vector (of floats), optional*) – For each entry in rel_time a spectrum is plotted. If time_width_percent=0 (Default) the nearest measured timepoint is chosen. For other values see "time_width_percent"
- **time_width_percent** (*float*) – "rel_time" and "time_width_percent" work together for creating spectral plots at specific timepoints. For each entry in rel_time a spectrum is plotted. If however e.g. time_width_percent=10 the region between the timepoint closest to the 1.1 x timepoint and 0.9 x timepoint is averaged and shown (and the legend adjusted accordingly).

This is particularly useful for the densly sampled region close to t=0. Typically for a logarithmic recorded kinetics, the timepoints at later times will be further appart than 10 percent of the value, but this allows to elegantly combine values around time=0 for better statistics. This averaging is only applied for the plotting function and not for the fits.

- **ignore_time_region** (*None or list (of two floats or of lists), optional*) – cut set a time range with a low and high limit from the fits. (Default) None nothing happens The region will be removed during the fitting process (and will be missing in the fit-result plots) Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **time_bin** (*None or int, optional*) – is dividing the points on the time-axis in even bins and averages the found values in between. This is a hard approach that also affects the fits. I do recommend to use this carefully, it is most useful for modulated data. A better choice for transient absorption that only affects the kinetics is "time_width_percent"
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g. your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.
- **color_offset** (*int, optional*) – At the (Default) 0 the colours are chose from the beginning, for a larger value Color_offset colors are skipped. Usually only used if multiple plots are created, and the data/or fit is only shown for some of them.
- **log_scale** (*bool, optional*) – If True (Default), The 2D plots (Matrix) is plotted with a pseudo logarithmic intensity scale. This usually does not give good results unless the intensity scale is symmetric
- **Scale_type** (*None or str*) – is a general setting that can influences what time axis will be used for the plots. "symlog" (linear around zero and logarithmic otherwise) "lin" and "log" are valid options.
- **lintresh** (*float*) – The pseudo logratihmic range "symlog" is used for most time axis. Symlog plots a range around time zero linear and beyond this linear treshold 'lintresh' on a logarithmic scale. (Default) 0.3
- **times** (*None or int*) – are the number of components to be used in the SVD (Default) is None (which is seen as 6)
- **save_figures_to_folder** (*bool, optional*) – (Default) is False, if True the Figures are automatically saved
- **savetype** (*str or iterable (of str), optional*) – matplotlib allows the saving of figures in various formats. (Default) "png", typical and recommendable options are "svg"

and “pdf”.

- **path** (*None, str or path object, optional*) – This defines where the files are saved if the safe_figures_to_folder parameter is True, quite useful if a lot of data sets are to be printed fast. If a path is given, this is used. If a string like the (Default) “result_figures” is given, then a subfolder of this name will be used (an generated if necessary) relative to self.path. Use and empty string to use the self.path If set to None, the location of the plot_func will be used and a subfolder with title “result_figures” be generated here
- **filename** (*str, optional*) – offers to replace the base-name used for all plots (to e.g.~specify what sample was used). if (Default) None is used, the self.filename is used as a base name. The filename plays only a role during saving, as does the path and savetype
- **print_click_position** (*bool, optional*) – if True then the click position is printed for the spectral plots
- **return_plots** (*bool, optional*) – (Default) False, return is ignoriert. For True a dictionary with the handles to the figures is returned

plot_func.**plot_time**(*ds*, *ax=None*, *rel_time=None*, *time_width_percent=10*, *ignore_time_region=None*, *wave_nm_bin=None*, *title=None*, *text_in_legend=None*, *baseunit='ps'*, *lines_are='smoothed'*, *scattercut=None*, *bordercut=None*, *subplot=False*, *linewidth=1*, *color_offset=0*, *intensity_range=None*, *plot_second_as_energy=True*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *data_type='differential Absorption in $\\mathregular{\\Delta OD}$'*, *units='nm'*, *equal_energy_bin=None*, *from_fit=None*)

Function to create plots at a certain time. In general you give under rel_time a list of times at which yu do want to plot the time width percentage means that this function integrates ewverything plus minus 10% at this time. lines_are is a switch that regulates what is plotted. data plots the data only, smoothed plots the data and a smoothed version of the data, fitted plots only the fit. the subplot switch is for using this to plot e.g. multiple different datasets.

Parameters

- **ds** (*DataFrame*) – This dataframe contains the data to be plotted. It is copied and sliced into the regions defined. The dataframe expects the time to be in Index and the wavelength/energy to be in the columns. The spectra is plotted with a second (energy) axis
- **ax** (*None or matplotlib axis object, optional*) – if None (Default), a figure and axis will be generated for the plot, if axis is given the plot will placed in there.
- **data_type** (*str*) – this is the datatype and effectively the unit put on the intensity axis (Default)’differential Absorption in $mathregular{Delta OD}$
- **rel_time** (*float or list/vector (of floats), optional*) – For each entry in rel_time a spectrum is plotted. If time_width_percent=0 (Default) the nearest measured timepoint is chosen. For other values see ‘time_width_percent’
- **time_width_percent** (*float*) – “rel_time” and “time_width_percent” work together for creating spectral plots at specific timepoints. For each entry in rel_time a spectrum is plotted. If however e.g. time_width_percent=10 the region between the timepoint closest to the 1.1 x timepoint and 0.9 x timepoint is averaged and shown (and the legend adjusted accordingly). This is particularly useful for the densly sampled region close to t=0. Typically for a logarithmic recorded kinetics, the timepoints at later times will be further appart than 10 percent of the value, but this allows to elegantly combine values around time=0 for better statistics. This averaging is only applied for the plotting function and not for the fits.

- **ignore_time_region** (*None or list (of two floats or of lists), optional*) – cut set a time range with a low and high limit from the fits. (Default) None nothing happens The region will be removed during the fitting process (and will be missing in the fit-result plots) Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **wave_nm_bin** (*None or float, optional*) – rebins the original data into even intervals. If set to None the original data will be used. If set to a width (e.g. 2nm), the wavelength axis will be divided into steps of this size and the mean of all measurements in the interval is taken. The re-binning stops as soon as the measured stepsize is wider than given here, then the original bins are used. This function is particularly useful for spectrometer with non-linear dispersion, like a prism in the infrared.
- **title** (*None or str, optional*) – title to be used on top of each plot The (Default) None triggers self.filename to be used. Setting a specific title as string will be used in all plots. To remove the title all together set an empty string with this command title=""
- **linewidth** (*float, optional*) – linewidth to be used for plotting
- **text_in_legend** (*str, optional*) – text to be used in legend before the actually lines and colours (set as heasder)
- **baseunit** (*str*) – baseunit is a neat way to change the unit on the time axis of the plots. (Default) 'ps', but they can be frames or something similarly. This is changing only the label of the axis. During the import there is the option to divide the numbers by a factor. I have also used frames or fs as units. Important is that all time units will be labeled with this unit.
- **scattercut** (*None or iterable (of floats or other iterable, always pairs!), optional*) – intented to "cut" one or multiple scatter regions. (if (Default) None nothing happens) If it is set the spectral region between the limits is set to zero. Usage single region: [lower region limit,upper region limit], use for multiple regions:[[lower limit 1,upper limit 1],[lower limit 2,upper limit 2],...]
- **bordercut** (*None or iterable (with two floats), optional*) – cut spectra at the low and high wavelength limit. (Default) None uses the limits of measurement
- **bool** (*subplot ;*) – False (Default) means this is a main plot in this axis! if True then this is the second plot in the axis and things like axis ticks should not be reset this also avoids adding the object to the legend
- **optional** – False (Default) means this is a main plot in this axis! if True then this is the second plot in the axis and things like axis ticks should not be reset this also avoids adding the object to the legend
- **color_offset** (*int, optional*) – At the (Default) 0 the colours are chose from the beginning, for a larger value Color_offset colors are skipped. Usually only used if multiple plots are created, and the data/or fit is only shown for some of them.
- **intensity_range** (*None, float or list [of two floats]*) – intensity_range is a general switch that governs what intensity range the plots show. For the 1d plots this is the y-axis for the 2d-plots this is the colour scale. This parameter recognizes three settings. If set to "None" (Default) this uses the minimum and maximum of the data. A single value like in the example below and the intended use is the symmetric scale while a list with two entries an assymmetric scale e.g. intensity_range=3e-3 is converted into intensity_range=[-3e-3,3e-3]
- **plot_second_as_energy** (*bool, optional*) – For (Default) True a second x-axis is plotted with "eV" as unit
- **cmap** (*None or matplotlib color map, optional*) – is a powerfull variable that chooses the colour map applied for all plots. If set to None (Default) then the self.cmap is

used. As standard I use the color map "jet" from matplotlib. There are a variety of colormaps available that are very usefull. Beside "jet", "viridis" is a good choice as it is well visible under red-green blindness. Other useful maps are "prism" for high fluctuations or diverging color maps like "seismic". See https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html for a comprehensive selection. In the code the colormaps are imported so if plot_func is imported as pf then self.cmap=pf.cm.viridis sets viridis as the map to use. Internally the colors are chosen with the "colm" function. The 2d plots require a continuous color map so if something else is give 2d plots are shown automatically with "jet". For all of the 1d plots however I first select a number of colors before each plot. If cmap is a continous map then these are sampled evenly over the colourmap. Manual iterables of colours cmap=[(1,0,0),(0,1,0),(0,0,1),...] are also accepted, as are vectors or dataframes that contain as rows the colors. There must be of course sufficient colors present for the numbers of lines that will be plotted. So I recommend to provide at least 10 colours (e.g.~your university colors). colours are always given as a, list or tuple with RGA or RGBA (with the last A beeing the Alpha=transparency. All numbers are between 0 and 1. If a list/vector/DataFrame is given for the colours they will be used in the order provided.

plot_func.**s2_vs_smin2**(*Spectral_points=512*, *Time_points=130*, *number_of_species=3*, *fitted_kinetic_pars=7*, *target_quality=0.95*)

dfn is numerator and number of fitted parameters, dfd is denominator and number of degrees of freedom, F-test is deciding if a set of parameters gives a statistical significant difference. T-test is if a single parameter gives statistical difference. Null hypothesis, all parameter are zero, if significant, the coefficients improve the fit the f-statistics compares the number of "fitted parameter"=number of species*number of spectral points + number of kinetic parameter "free points"=number of species*number of spectral points*number of time points - fitted parameter within the target quality, meaning, what fraction do my variances need to have, so that I'm 100% * target_quality sure that they are different from zero

PYTHON MODULE INDEX

INDEX

Symbols

B

C

E

F

G

M

N

P

S

T