PPMC3Fitting

The script `PPMC3Fitting` (which stands for **P**ump-**P**robe **M**ultiple **C**hannels **M**arkov **C**hain **M**onte-**C**arlo **F**itting) is designed to fit complicated observed pump-probe curves for different observables, that share the dynamical parameters.

It can do the:

- local fitting,
- global fitting,
- Monte-Carlo sampling of the errors with Metropolis algorithm

for one or multiple pump-probe dependencies.

The software was written in Python, and can be obtained from Stash: https://stash.desy.de/projects/CFA/repos/mcmcmcfitting/browse

Backgrounds on the fitting procedures

How each data set is fitted

Experimental data

Each data is given as a set of the experimental pump-probe points $\{t_i, y^{(\mathrm{exp})}(t_i) = y_i^{(\mathrm{exp})}, \sigma_i\}_{i=1}^{N}$, where *t* is the pump-probe delay, and *y* is the value of the observable, is the error of *y* (required here!), and *N* is the total number of the experimental points.

Theoretical approximation

We assume that each observable is being formed as a superposition of independent channels producing the same observable. This can be represented as

$$y(t) = \sum_{k=1}^{M} A_k \cdot f_k(t)$$

where *A* is the amplitude of the different channels, *M* is the total number of channels, and *f*(*t*) is the possible pump-probe-dependent yield of the observables (channel). There can be a few standard basic channels (see also Channels description files (what -c option requires)). We will describe them with the quasi-chemical reaction schemes. The notations are:

- M_0 is the initial molecule,
- $_{\mathrm{pump}}/_{\mathrm{probe}}$ are the pump/probe lasers photons,
- $n_{\mathrm{pump}}/n_{\mathrm{probe}}$ are the numbers of $_{\mathrm{pump}}/_{\mathrm{probe}}$ the in the process,
- M (=1,2,...) are the intermediates,
- M_{obs} is the outcome molecular specie that is being observed.

The main parameters of the interaction of the molecules with the pump-probe pulses are the t_0 – the temporal overlap of the pump and probe pulses (when two pulses hit the molecule simultaneously), and the cross-correlation time $_{\mathrm{cc}}$ of the pulses. If we assume the pump and the probe laser pulses to be Gaussian-shaped, the $_{\mathrm{cc}}$ is given as

$$\tau_{\mathrm{cc}}^2 = \frac{\tau_{\mathrm{pump}}^2}{n_{\mathrm{pump}}} + \frac{\tau_{\mathrm{probe}}^2}{n_{\mathrm{probe}}} + \tau_{\mathrm{jitter}}^2$$

where $_{\mathrm{pump}}$ and $_{\mathrm{probe}}$ are the durations of the pump-probe pulses in the pulse shape given as

$$\text{Pulse shape}(t) = \exp\left(-\frac{t^2}{\tau^2}\right)$$

and $_{\mathrm{jitter}}$ is the random fluctuations of the pump-probe delay (important for the experiments with free-electron lasers with SASE). The resulting functional dependencies can be derived by integrating a reaction scheme with distinct instant pulses ($f_0(t)$), and convoluting the result with the cross-correlation pulse shape:

$$f(t) = \exp(-t^2/\tau_{\mathrm{cc}}^2) \otimes f_0(t)$$

Note: we always use a convention that the (t-t_0)<0 corresponds to probe acting on the molecules before pump, and (t-t_0)>0 to pump acting on the molecules before probe. Therefore if we switch pump and probe, we would have to invert the (t-t_0) axis!

Constant (const) channel

This is the background of the pump-probe experiment. For instance, An example of this channel can be interaction with the single laser, i.e. $M_0 + n\gamma \rightarrow M_{\mathrm{obs}} + \ldots$. The functional dependence thus is

$$f(t) = \text{const}(t) \propto 1$$

Gaussian-shaped channel (gauss)

This happens if the observable forms only if both pump and probe lasers hit the system simultaneously ($M_0 + n_{\mathrm{pump}}\gamma_{\mathrm{pump}} + n_{\mathrm{probe}}\gamma_{\mathrm{probe}} \rightarrow M_{\mathrm{obs}} + \ldots$). In practice, this means two very fast sequential processes. One example is the formation of the sidebands of the photoelectrons, when freshly ionized electrons by a pump pulse absorb/emit one or several photons of the probe radiation. The functional dependence is thus

$$f(t) = \delta(t) \otimes \exp(-t^2/\tau_{\mathrm{cc}}) \propto \exp\left(-\frac{(t-t_0)^2}{\tau_{\mathrm{cc}}^2}\right)$$

Switch of the behavior (switch)

This happens if the absorption of the pump photon instantly produces something stable, that can further interact with the probe photons, i.e. reaction scheme is

$$\begin{cases} M_0 + n_{\mathrm{pump}}\gamma_{\mathrm{pump}} \rightarrow M_1 + \ldots \\ M_1 + n_{\mathrm{probe}}\gamma_{\mathrm{probe}} \rightarrow M_{\mathrm{obs}} + \ldots \end{cases}$$

This leads to the functional dependence given as

$$f(t) = \theta(t) \otimes \exp(-t^2/\tau_{\mathrm{cc}}) \propto \left[1 + \mathrm{erf}\left(\frac{(t-t_0)}{\tau_{\mathrm{cc}}}\right)\right]$$

with being Heaviside step function.

Short lived intermediates = transients (trans)

If we have something short-lived formed by the pump pulse, we can probe it with the probe pulse by turning it into something observable, i.e. the reaction scheme:

$$\begin{cases} M_0 + n_{\text{pump}}\gamma_{\text{pump}} \rightarrow M_1 \text{ (short lived intermediate)} + \dots \\ M_1 \rightarrow M_2 \text{ (relaxation described by the reaction constant } \tau_{\text{r}}) \\ M_1 + n_{\text{probe}}\gamma_{\text{probe}} \rightarrow M_{\text{obs}} + \dots \end{cases}$$

This leads to the functional dependence of

$$f(t) = \left(\theta(t)\cdot\exp\left(-\frac{t}{\tau_{\text{r}}}\right)\right) \otimes \exp(-t^2/\tau_{\text{cc}}) \propto \exp\left(-\frac{(t-t_0)}{\tau_{\text{r}}}\right)\cdot\left[1+\text{erf}\left(\frac{(t-t_0)}{\tau_{\text{cc}}}-\frac{\tau_{\text{cc}}}{2\tau_{\text{r}}}\right)\right]\cdot\exp\left(\frac{\tau_{\text{cc}}^2}{4\tau_{\text{r}}^2}\right)$$

Fitting procedure

We have the two sets of parameters:

- linear parameters (amplitudes, or cross-sections of the different channels, *A*'s), this we will denote as a vector **A**,
- nonlinear parameters (t_0's, $_{cc}$'s, and $_r$'s), this we will denote as a vector **T**.

The problem can be that there can be multiple t_0's (if there are pre-/post-pulses, if the temporal shape of the laser pulse contains "shoulders"), multiple $_{cc}$'s (if different processes require different number of pump and/or probe photons), and $_r$'s (if there are multiple possible intermediates). We can try to separate fitting of these two sets of parameters. If we fix the T at some values, we will represent the theoretical function as a scalar product $y(t,\mathbf{A},\mathbf{T}) = \mathbf{A}\cdot\mathbf{f}(t,\mathbf{T})$ with vectors $\mathbf{A} = (A_1, A_2, \dots, A_N)^\dagger$ and $\mathbf{f} = (f_1(t,\mathbf{T}), f_2(t,\mathbf{T}), \dots, f_N(t,\mathbf{T}))^\dagger$. This leads to simple linear least-squares (LSQ) fitting. We want to minimize functional

$$\Phi(\mathbf{A}|\mathbf{T}) = \sum_i w_i (y_i^{(\text{exp})} - y_i^{(\text{theor})})^2 \rightarrow \min$$

with weights $w_i = \sigma_i^{-2}$. Extremum condition $\{\frac{\partial\Phi}{\partial A_k} = 0\}_{k=1}^M$ leads to standard linear LSQ equations $Q\mathbf{A} = \mathbf{B}$ with matrix Q containing elements $Q_{kl} = \sum_{i=1}^N w_i f_k(t_i) f_l(t_i)$, and vector **B** with elements $B_k = \sum_{i=1}^N w_i y_i^{(\text{exp})} f_k(t_i)$. The solution will give an optimal set of cross-sections **A** at the fixed nonlinear variables vector **T** (i.e. $\mathbf{A}_{\text{min}}(\mathbf{T})$), and the smallest possible LSQ functional with this **T**:

$$\Phi(\mathbf{T}) = \sum_i w_i (y_i^{(\text{exp})} - \mathbf{A}_{\text{min}}(\mathbf{T})\cdot\mathbf{f}_i)^2$$

Therefore, finding the optimal set of nonlinear parameters **T** ($\Phi(\mathbf{T}) \rightarrow \min$) will lead to the best possible solution. Therefore we consider linear parameters (cross-sections) fitting as an intermediate step procedure.

Regularization in the fitting

We can augment the fitting by replacing the $\Phi(\mathbf{T}) \rightarrow \min$ by an effective functional minimization $\Phi_{\text{effective}}(\mathbf{T}) = (\Phi_{\text{initial}}(\mathbf{T}) + \Phi_{\text{reg}}(\mathbf{T})) \rightarrow \min$, where $\Phi_{\text{reg}}(\mathbf{T})$ is the penalty (regularization) functional.

There are two cases when we might want to have the regularization.

1. One of the parameters has an independent experimental/theoretical estimation, namely the value ($_0$) and its error estimation (), and this can be used to constraint the fitting procedure. In this case, we need to add to the $\Phi_{\text{reg}}(\mathbf{T})$ a term $\frac{(\xi-\xi_0)^2}{2\sigma^2}$ (basically a Gaussian distribution, see next section).
2. Two channels can give variables close to each other, which will cause the linear dependence in the fits, and the close-by parameters (say, and) need to be "pushed apart". This can be done by adding to the $\Phi_{\text{reg}}(\mathbf{T})$ a term $\frac{\alpha}{|\xi-\eta|}$ (0 is the regularization parameter with the same dimensionality as and , and the physical meaning of the is the separation range we believe is for this pair of variables).

More details can be found in Optional: regularization files (-r option).

How Monte-Carlo sampling works

In reality, we can have multiple possible local minima solutions of the nonlinear parameters **T** with similar values of ($\mathbf{T}$) (does not matter, with regularization or not), or the local shape of the functional in the vicinity of the optimal **T** solution can be far away from parabolic, that would lead to bad estimation of the error of the nonlinear parameters with the standard formulas. To get a better estimation of the errors we will apply the Monte-Carlo sampling procedure. We assume that the deviation of each *i*-th measurement and theory is distributed with a normal distribution (invert the idea of the LSQ fitting 🙂). The probability of each solution with nonlinear parameters **T** is thus given by equation

$$P(\mathbf{T}) = P_0\cdot\exp(-\Phi(\mathbf{T}))$$

with an unknown normalization constant P_0. This can be solved by the Metropolis algorithm.

- Let's assume we have the current values of the **T** given by $\mathbf{T}_{\text{this}}$ with functional value $_{\text{this}}$=($\mathbf{T}_{\text{this}}$).
- We generate a new trial vector of the value $\mathbf{T}_{\text{trial}}$ with functional value $_{\text{trial}}$=($\mathbf{T}_{\text{trial}}$).
- We calculate the acceptance probability of this trial point by a formula $P_{\text{acc}} = S\cdot\min\{1, \exp(\Phi_{\text{this}} - \Phi_{\text{trial}})\}$ (S>0 is an arbitrary scaling parameter, that defines acceptance ratio, changed by parameter `"--MetroProbScale"`, S=1 by default).

- We generate the trial probability P_{trial} as a uniformly-distributed random value between 0 and 1.
- We compare P_{trial} and P_{acc}.
    - If $P_{\text{acc}} \geq P_{\text{trial}}$ we accept the new point, i.e. on the next iteration $T_{\text{this}} = T_{\text{trial}}$.
    - If $P_{\text{acc}} < P_{\text{trial}}$ we reject the new point, i.e. on the next iteration $T_{\text{this}} = T_{\text{this}}$.

With this algorithm we generate a long trajectory of values **T**: $\{\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_N\}$ (here we redefined the *N* to be the length of the trajectory, given by parameter "`--NumOfMCMCSimPoints`"). The first part of the trajectory is usually ignored as an equilibration stage (controlled by the parameter "`--WhichPartOfMCMCTrjToIgnore`"). From the trajectory we can compute the probability distribution of the **T** parameter, or any functional dependence dependent from **T** (*F*(**T**)): $\langle F \rangle \approx \sum_{i=N_{\min}}^{N} F(\mathbf{T}_i)$.

How to use?

One has to apply the top-level script ppmc3fitting.py.

The simplest application can be given by the command

Simplest comand to run the script

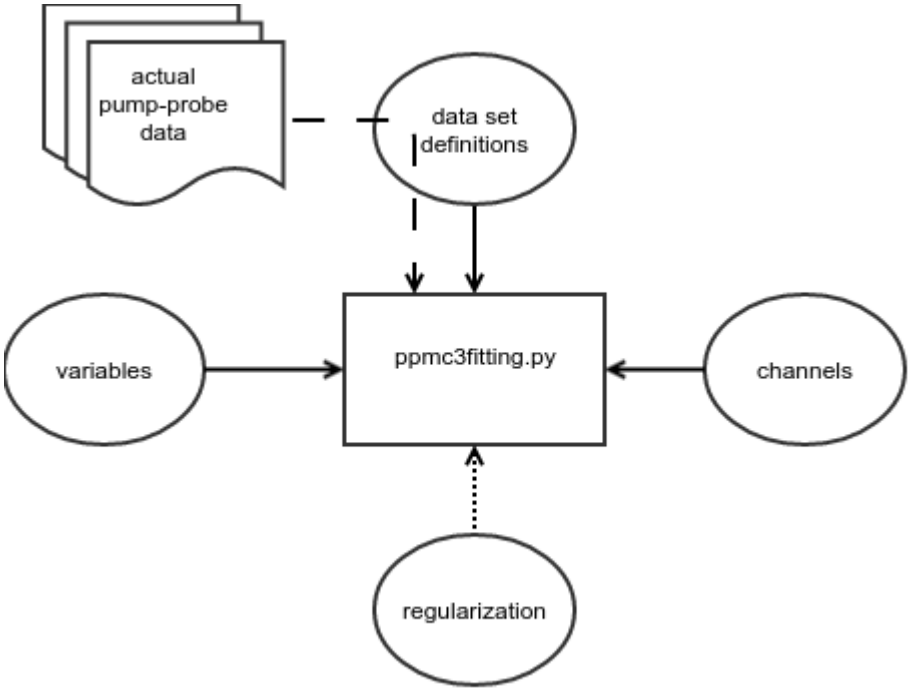
```
python3 ppmc3fitting.py -d dataset.def -c channels.def -v variables.def
```

In this case, one should have the actual pump-probe data stored in different files, and three files defining the job to be done:

- file defining the data sets (here – dataset.def, see DataSet files (what goes to -d)),
- file defining the channels (here – channels.def, see Channels description files (what -c option requires)),
- file defining the variables (here – variables.def, see Variables descripton files (what -v option requires)).

Additionally, the actual datasets should be provided (see How should pump-probe data look like?).

The additional options can be viewed using `-h` or `--help` flag.



Installation and usage

1. Get a copy of the code from https://stash.desy.de/projects/CFA/repos/mcmcmcfitting/browse. Either clone the git using git clone, or save it some other way.
2. Software depends on a few Python modules. All of them can be installed using pip (as e.g. `pip install <module name> --user`), these modules are:
    a. NumPy
    b. SciPy
    c. configparser
    d. re
    e. argparse
3. Optionally, one can make the script executable e.g. via `chmod +x ppmc3fitting.py`.
4. To run the script either add the path to the script to the PATH variable in Shell/Bash/Z Shell/..., or give the full path to the file.

Examples

Examples of the application are distributed with the scripts. They can be found in here: https://stash.desy.de/projects/CFA/repos/mcmcmcfitting/browse/tests/fitting_test

Channels description files (what -c option requires)

An example file can look like this:

Example of channels definition file

```
[Baseline]
type = const

[PumpProbeSwitch]
type = switch
t0  = t0
tcc = tcc


[ForwardTransient]
type = trans
t0  = t0
tcc = tcc
tr  = tr
ProbeIsPump : false
```

Each block defines a channel (a building block of the fitting functions).

In the square bracket is the name of the channel, this name it should be referred by in the dataset definition file. Note, that the capital letters will be converted to lower case letters, therefore don't use case-sensitive names.

Within the definition of each channel, there are a few possible parameters:

- `type` = const/gauss/switch/trans. This is the type of the fitting function (see below).
- `t0` – is the name of the variable from the variable definition file, that represents the t_0 parameter (temporal overlap of the pump and probe pulses). This parameter is not required for the 'const' type.
- tcc – is the name of the variable from the variable definition file, cross-correlation time ($_{cc}$), which consists of the duration of the pump and probe pulses, and of the random fluctuations of the pump-probe delay. This parameter is not required for the 'const' type.
- tr – is the name of the variable from the variable definition file, relaxation rate of the short-lived intermediate. This parameter is required only for the 'trans' type.
- `PumpIsProbe` – boolean flag (can be True, False, yes, no,...) to control the direction of the pump-probe axis for this particular channel. By default is False, and therefore the channel will see only the +x. If set to true, the channel will use the -x. Crucial only for 'trans' type and for the 'switch' type, when no 'const' channel is given.

Types of the channels (fitting functions)

type = const

This is a constant value with respect to pump-probe delay:

$$f(t) = \mathrm{const}(t) = 1$$

Physical meaning: background signal.

type = gauss

This is a Gaussian function in the form:

$$f(t) = \exp\left(-\frac{(t-t_0)^2}{\tau_{cc}^2}\right)$$

Physical meaning: some cross-correlation process, that happens instantly only if both pump and probe pulses hit the system simultaneously.

type = switch

This is the switch for some process being enabled only if the pump is hitting the target before the probe. The functional form is:

$$f(t) = \left[1 + \mathrm{erf}\left(\frac{(t-t_0)}{\tau_{cc}}\right)\right]$$

type = trans

This is a signal of a shortly lived transient with the lifetime of $_r$. Functional form is

$$f(t) = \exp\left(-\frac{(t-t_0)}{\tau_r}\right) \cdot \left[1 + \mathrm{erf}\left(\frac{(t-t_0)}{\tau_{cc}} - \frac{\tau_{cc}}{2\tau_r}\right)\right] \cdot \exp\left(\frac{\tau_{cc}^2}{4\tau_r^2}\right)$$

$\left(\exp\left(\frac{\tau_{cc}^2}{4\tau_r^2}\right)\right.$ is there to prevent the function from becoming too large/too small).

What if `PumpIsProbe = True`?

If `PumpIsProbe = False` (default), then we use *f*(*t*). If `PumpIsProbe = True`, then we will use *f*(-*t*) instead!

DataSet files (what goes to -d)

This file defines which pump-probe datasets (see How should pump-probe data look like?) are to be taken, and how to fit them. An example file looks like this:

Example of dataset definition file

```
[LKEe]
file : flu_lkee.dat
channels :  Baseline   PumpProbeSwitch    ForwardTransient

[dication]
file : flu-dication_yield.dat
channels : Baseline   PumpProbeSwitch BackwardTransient  ForwardTransient
```

Each block indicates the dataset. The squared parentheses contain the name of the dataset, that will be used in the code. Each dataset definition must contain two parameters:

- `file` – this is the name of the file to take the pump-probe data (in the format described in How should pump-probe data look like?) from.
- `channels` – this is the list of channels from the channels description (Channels description files (what -c option requires)) file. Note: all the channels' names will be converted to the lower case string!

How should pump-probe data look like?

The pump-probe data should be either three-column files with columns contents being

```
<(x) 1st column contains the pump-probe delays>  <(y) 2nd column contains the fitted observable value>  <(yerrors) 3rd column is the errors of the y's>
```

or four-column file with (this is what is being produced by the CAMPFancyAnalysis)

```
<(x) 1st column contains the pump-probe delays>  <(y) 2nd column contains the fitted observable value> <3rd column can contain anything, but presumably there are the errors of the x's> <(yerrors) 4th column is the errors of the y's>
```

In both cases only the `x,y,yerrors` data are taken.

Optional: regularization files (-r option)

There is a possibility to also use the regularization in the scripts. Two main reasons for that are:

- sometimes one of the parameters has an independent experimental/theoretical estimation, and this can be used to constraint the fitting procedure;
- sometimes two channels can give variables close to each other, which will cause the linear dependence in the fits, and the close-by parameters need to be "pushed apart".

This can be done with a file that looks, for instance, like this:

Regularization file example

```
   constraint   t0_a    0.0  0.5
 constraint     tcc_a   0.1  0.5
  repellent     t0_a  t0_b   0.01
```

Constraint type parameters

These parameters are to impose an *a priori* known value of some variable. Note: there can be as many constraints for a single variable, as we want. A corresponding definition line in the file has the following structure:

`constraint <name of the variable from -v file> <known value of the variable> <error of the variable, standard deviation>`

This block will add a penalty term $\frac{(\xi-\xi_0)^2}{2\sigma^2}$ with $_0$ being the `<known value of the variable>`, being `<error of the variable, standard deviation>`, and is the value of the variable `<name of the variable from -v file>`.

Repellent type parameters

These parameters are to push apart two variables, that are close to each other, and that cause bad linear dependencies. A corresponding definition line in the file has the following structure:\

`repellent <name of the fist variable from -v file> <name of the second variable from -v file> <regularization parameter >`

This block will add a penalty term $\frac{\alpha}{|\xi-\eta|}$ $(\alpha \geq 0)$ to the regularization functional.

Variables descripton files (what -v option requires)

This file should contain a description of the variables. For example:

Example of variables definition file

```
[t0]
minvalue = 12.3
maxvalue = 13.1
inivalue = 12.649
maxdisp  = 0.005

[tcc]
minvalue = 0.07
maxvalue = 0.20
inivalue = 0.15
maxdisp  = 0.005

[tr]
minvalue = 0.005
maxvalue = 0.200
inivalue = 0.050
maxdisp  = 0.005
```

Each block is a definition for a single variable with a name (identifier) given in the squared parentheses. **Note**, that the upper case letters will turn to lower case letters internally, therefore choose wisely the names of the variables. Each variable is defined by four possible keys:

- `minvalue/maxvalue` (required) – these are the lower/upper bounaries for the possible values of the variable.
- `inivalue` (optional) – this is the initial value of the variable, the fitting/Monte-Carlo (MC) sampling will start from. By default is None, and if not given, the initial value will be drawn from the uniform distribution in the ranges [`minvalue`; `maxvalue`).
- `maxdisp` (optional) – this is the maximal displacement of the variable in the MC sampling step. By default is None, and if not given, each MC iteration will draw this variable from the uniform distribution in the ranges [`minvalue`; `maxvalue`). If given, each next trial value of this variable will be $x_{\mathrm{new}} = x_{\mathrm{previous}} + \mathrm{maxdisp} \cdot \xi$ where *x*'s are the variable values (new and previous), and is the random variable drawn from the uniform distribution in the range [-1;1).

Software structure

The script consists of the library (`libMCMCMCFitting.py`) with different routines and the user interface script (`ppmc3fitting.py`).

`libMCMCMCFitting`

The library can be used as an API for something more complicated, than what `ppmc3fitting.py` can do. The general idea for the fitting routines will be discussed somewhere else. The contents of the library are the following.

Fitting functions

These are

- `ConstFunc` (type of the function = 'const')
- `GaussFunc` (type of the function = 'gauss')
- `SwitchDirectFunc` (type of the function = 'switch')
- `SwitchBackwardFunc` (type of the function = 'switch')
- `TransDirectFunc` (type of the function = 'trans')
- `TransBackwardFunc` (type of the function = 'trans')

Each of these functions can be returned based on the type of function requested using `GetSomeFunc` routine.

Reading of the input files

The direct reading of certain types of files is done using three routines. Each of these routines uses the configparser module of Python, they take only a single argument: the name of the file to be read. As a result of their execution, all of them return a dictionary with certain values.

- `readVariablesDefinitions` (just reads the variables definition file). It returns a dictionary with the keys being the names of the variables, and each of the dictionary values contains another dictionary with the keys "value" (the current value of the variable), "minvalue"/" maxvalue" (the minimal/maximal values of the allowed range for the variable), and the "maxdisp" (the maximal displacement in the Monte-Carlo sampling, can be None).
- `readChannelsDefinitions` (just reads the channels definition file). The keys of this dictionary are the channel names, the value accessed by the channel name are the definitions of the channels with the keys being "type" (the type of the representing fitting function), "t0" (the variable that is the t0 for this channel, required for all four types of the channels), "tcc" (cross-correlation time, is not required and thus is ignored for the 'const' type), and "tr" (relaxation time, only needed in the "trans" type).
- `readDataSetsDefinitions` ((just reads the channels definition file). The returned dictionary has the names of the datasets as the keys, and each dataset has a few keys: "x", "y", "yerr" (Numpy 1D arrays containing the x,y, values of the pump-probe curves, and the errors for y-values), "channels" (list of the names of the channels to fit the discussed dataset).

Note! All the keys given in all the files are always turned into the case insensitive lower case strings!

Then there also is a generalized routine, `ReadTheData`, that is being used to obtain a prepared data.

Definition of ReadTheData function

```
def ReadTheData(DataDescription,      # data description file name
                ChannelsDescription,  # channels description file name
                VariablesDescription, # variables description file name
                RegularizationDescription = None, # regularization description file name
                xmin=None, xmax=None, Npts=100, xMinMaxPart=0.1   # these are parameters of the
DataSetHandler class example initiation
               ):
```

This function returns:

1. ListOfData, a list of DataSetHandler instances, storing the datas and the definition of the functions for the fitting
2. VarHand, a VariablesHandler class example, which controls the life of the fitted nonlinear variables
3. RegInstance (None or AdditionalConstraints class example) -- is the thing controlling the regularization

Handlers of the data

There are three handlers of the data.

`VariablesHandler`

This class is used to store the variables and generate their new values (in Monte-Carlo sampling).

Initiation of the example requires all three dictionaries produced by `readDataSetsDefinitions, readChannelsDefinitions`, and `readVariablesDefinitions` functions (in this order). The important methods of this class are:

- `VariablesHandler.getMap()` – get a map: a list of the variables' names. It is being used to map a sampled variables vector to the actual variables in each channel. The maps are being produced by the consequential application of the `GetOnlyTheChannels`, `GetOnlyTheVariables`, and `GetVariableMap` routines.

- `VariablesHandler.getVarVec()` – get the current variables vector, a Numpy array with the values.
- `VariablesHandler.getBounds()` – get a tuple of (min,max) tuples for each variable, can be directly used by the Scipy.optimize routines as the variables boundaries.
- `VariablesHandler.updateVarVec()` – change the currently stored variables.
- `VariablesHandler.genTrialVarVec()` – returns a trial vector according to the rules for sampling this variable.

`DataSetHandler`

This class stores the actual data for fitting, fitting functions, makes linear least-squares (LSQ) fitting and gives the data for plotting. The initiation is like this:

`DataSetHandler(DataSet, DataSetName, Channels, VariableMap, xmin=None, xmax=None, Npts=100, xMinMaxPart=0.1)`, where

- `DataSet` is the dictionary being a value of the dictionary produced by the `readDataSetsDefinitions` routine accessed using the data set name,
- `DataSetName` is the name of the dataset (e.g. the key in the dataset dictionary),
- `Channels` is the dictionary produced by the `readChannelsDefinitions` function.
- `VariableMap` is the list of variables names, produced either by the `VariablesHandler.getMap()` method, or using the `GetVariableMap`.
- `xmin`/`xmax` are the boundaries of the theoretical function produced for plotting, by default they are determined using the `xMinMaxPart` parameter.
- `Npts` is the number of points in the theoretical function produced for plotting,
- `xMinMaxPart` is the parameter to automatically determines the `xmin`/`xmax` by taking them to be the last/first experimental point + `xMinMaxPart * (last point value - first point value)`.

Important methods.

- `DataSetHandler.update(VariablesVector)`, updates the theoretical fits according to the new values of the fitted parameters.
- `DataSetHandler.getChNames()`, shows the names of the channels, in the form of the array.
- `DataSetHandler.getAmp()`, shows the amplitudes of the different channels.
- `DataSetHandler.getCurrentYtheror(GiveExpPoints = False)`, returns the theoretical points of the fit, `GiveExpPoints` controls which grid to use (if `True`, the experimental grid will be used).
- `DataSetHandler.getCurrentXtheror(GiveExpPoints = False)`, returns the x-points or the theoretical points of the fit, `GiveExpPoints` controls which grid to use (if `True`, the experimental grid will be used).
- `DataSetHandler.getLSQ()`, this returns the LSQ deviation between experiment and theory.

`AdditionalConstraints`

This class is for doing the regularization.

`AdditionalConstraints(FileName, VariableMap)`, where

- `FileName` is the name of the file containing the regularization parameters,
- `VariableMap` is the list of variables names, produced either by the `VariablesHandler.getMap()` method, or using the `GetVariableMap`.

Important methods.

- `AdditionalConstraints.getRegValue(VarVec)` – returns the value of the regularization functional with the given vector of variables.

Functions to do the fitting

There are three functions for doing that:

- `DoTheGlobalFit(ListOfData, VarHand, AddConstr=None, Method = 'DiffEvo', AdditionalArguments=dict(), ReturnDictionary=False, AddInfoToFiles = None)`, does global fit of the dependencies.
- `DoTheMCMCSampling(ListOfData, VarHand, AddConstr=None, NSteps=10000, NStepsToIgnore=None, PartToIgnore=0.1, AcceptanceProbabilityScale = 1.0, AddInfoToFiles = None, NBinsInHist = 50, PrintStatus = True)`, performs Monte-Carlo sampling of the nonlinear fitted variables, to obtain a reasonable error estimates.
- `DoTheLocalFit(ListOfData, VarHand, AddConstr=None, Method = 'Powell', AdditionalArguments=dict(), ReturnDictionary=False, AddInfoToFiles = None)`, does the local optimization.

The main parameters are shared:

- `ListOfData` is a list of `DataSetHandler` instances,
- `VarHand` is a `VariablesHandler` instance,
- `AddConstr` is either `None` (then no regularization), or the `AdditionalConstraints` instance.
- `AddInfoToFiles` is an additional description, that will be added to the names of the output files produced by the functions.

The specialized parameters are given below

`DoTheGlobalFit/DoTheLocalFit`

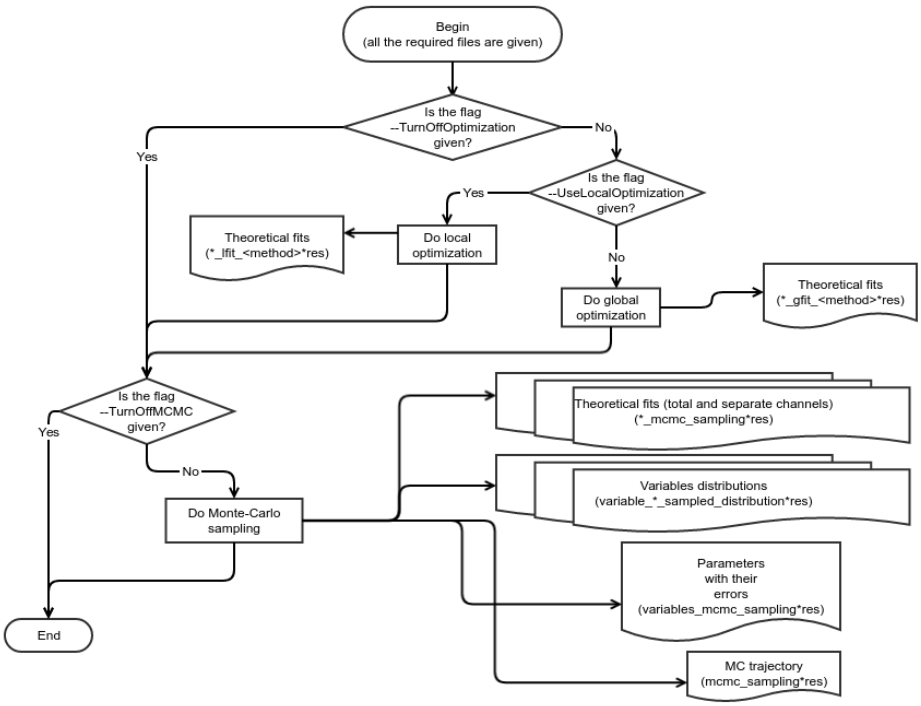
- `Method` is the method of the `Scipy.optimize` to be used. Only two are supported for global fit: 'DiffEvo' (differential evolution) and 'DualAnn' (dual annealing). Local fit supports all the types of the `Scipy.optimize.minimize`.
- `AdditionalArguments` are the additional arguments of the optimization method function, given in the form of a dictionary.
- `ReturnDictionary` -- determines whether it will return a vector of variables, or a dictionary (good for printing).

`DoTheMCMCSampling`

- `NSteps` -- number of steps in MCMC simulation,
- `NStepsToIgnore` – how many first steps to ignore (overrides PartToIgnore if provided).
- `PartToIgnore` – which part of the MCMC simulation to ignore (is overridden by NStepsToIgnore if not None),
- `AcceptanceProbabilityScale` — scaling coefficient for the Metropolis acceptance probability.
- `NBinsInHist` – number of bins in the histograms.
- `PrintStatus` – bool flag to print the status of the MCMC sampling.

`ppmc3fitting`

The way this software works (flagwise) is the following:



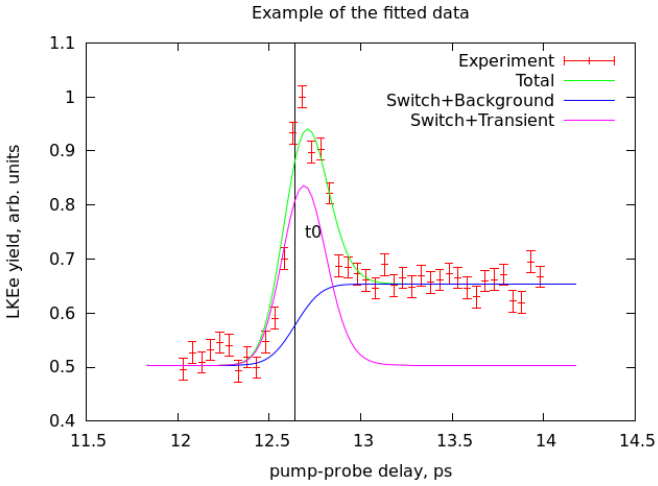
Understanding the results

The script ppmc3fitting.py gives multiple output files. Here is a small guide to understanding them.

Files produced by the local/global fitting

The fitting produces two files:

- `variables_[g,l]fit_<name of the optimization method><optional additional information, if given>.res` This file contains the values of the fitted variables.
- `<dataset name>_[g,l]fit_<name of the optimization method><optional additional information, if given>.res` This file contains a theoretical fitted function (including separate channels) fitted to each experimental dataset (distinct by the name), that can be used for plotting. The length of the function outside the region of the experimental value is defined by the option "`--MinMaxPart`", whether the number of theoretical points for the plotting is given by the parameter "`--NumOfTheorPoints`". The plots produced by that are e.g. this:



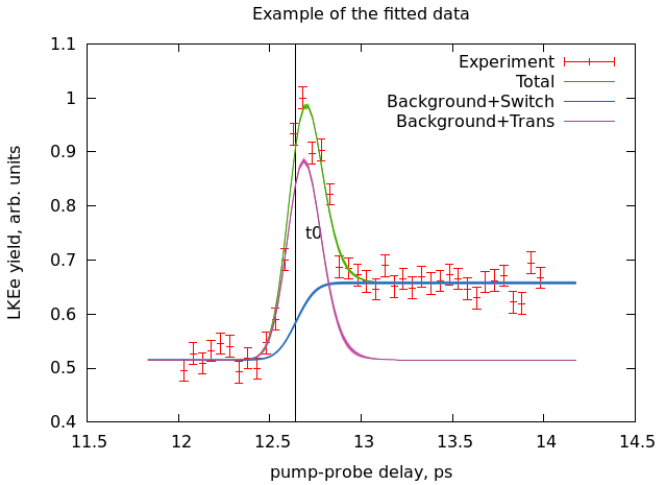
- `<dataset name>_residuals_[g,l]fit_<name of the optimization method><optional additional information, if given>.res` This file contains residuals of the fit, defined as the difference of theoretical fitted function (including separate channels) from each experimental point in the dataset (distinct by the name), that can be used for plotting.

Global optimization results are denoted with "`gfit`" in the file names, and the local optimization – with "`lfit`".

Files produced by the Monte-Carlo (MC) sampling

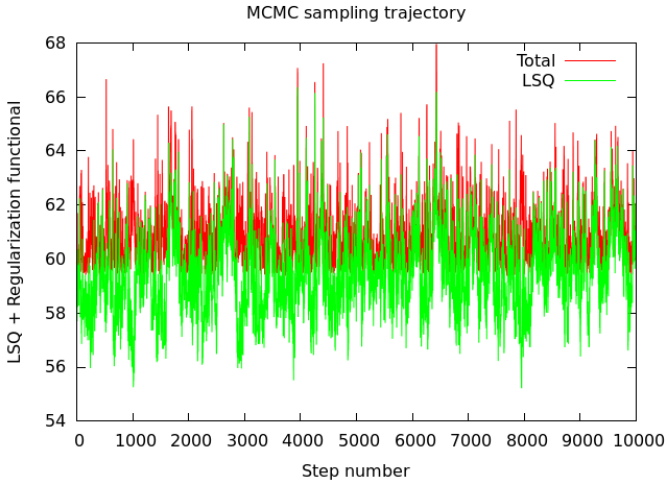
MC sampling produces lots of files.

- Similar to the fitting, it produces a file called `variables_mcmc_sampling<optional additional information, if given>.res`, which contains the average values of the variables and their standard deviations, that come from the sampled distribution.
- `<dataset name>_mcmc_sampling<optional additional information, if given>.res`, which contains a theoretical fitted function (including separate channels) fitted to each experimental dataset (distinct by the name), that can be used for plotting. The length of the function outside the region of the experimental value is defined by the option "`--MinMaxPart`", whether the number of theoretical points for the plotting is given by the parameter "`--NumOfTheorPoints`". The plots produced by that are e.g. this:

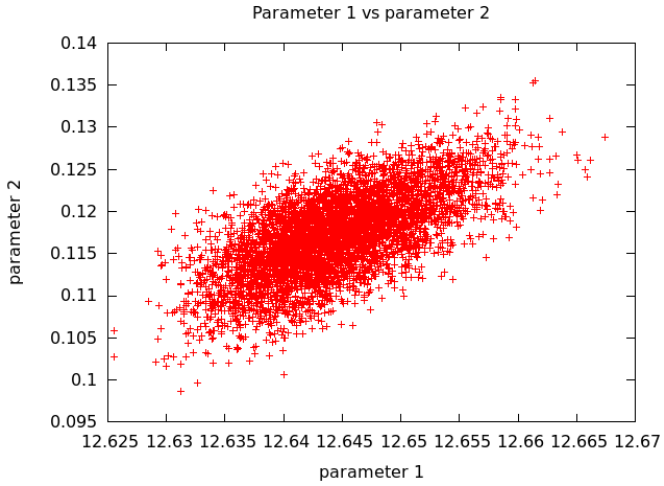


- `mcmc_sampling<optional additional information, if given>.trj` – this is the Metropolis MC trajectory (full), which contain the values of the variable on the each step of the simulation, the acceptance probability, trial random number, and the values of the least-

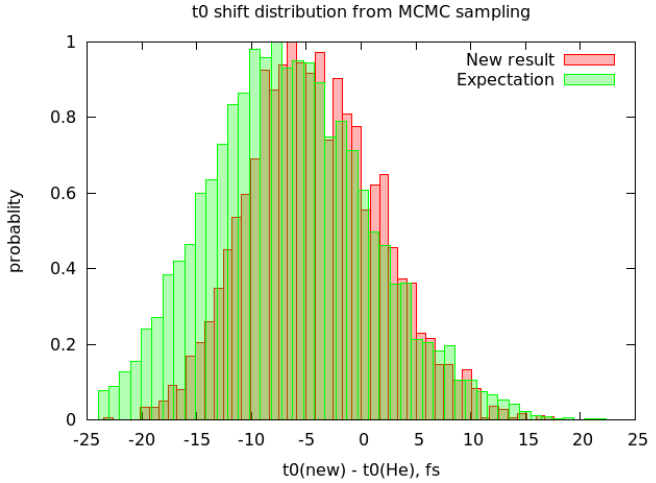
squares (LSQ) and regularization functional. This can give the graphs of the trajectory, or the relative distributions of the different



parameters, e.g.: or



- variable_<name of the variable>_sampled_distribution<optional additional information, if given>.res – this is the MC sampled probability distributions for each fitted variable. The number of bins of the distribution is controlled by the "--NumOfBinsInHist" parameter. The distribution show the actual shape (non-Gaussian for nonlinear parameters):



.

- <dataset name>_residuals_mcmc_sampling<optional additional information, if given>.res, which contains a difference between theoretical fitted function (including separate channels) and experiment (fit residuals) for each experimental dataset (distinct by the name), that can be used for plotting.