lmfit-varpro Documentation

Release 0.0.4

Joris Snellenburg, Joern Weissenborn

Aug 09, 2018

USER DOCUMENTATION:

1 Warning 3

2 Additional warning for scientists 5

3 Installation 7

4 Usage 9

5 API Documentation 11

6 Abstract 35

7 Scientific Sources 37

8 Contributing 39

9 “how to” in depth 43

10 Authors 49

11 History 51

12 Credits 53

13 Indices and tables 55

Python Module Index 57

CHAPTER 1

Warning

This project is still in its pre-alpha phase and undergoing rapid development, including changes to the core API, thus it is *not* production ready.

CHAPTER 2

Additional warning for scientists

The algorithms provided by this package still need to be validated and reviewed, pending the official release it should not be used in scientific publications.

2.1 lmfit-varpro

Python-lmfit based implementation of variable projection

2.1.1 Credits

The credits can be found in the documentations credits section

CHAPTER 3

Installation

3.1 Stable release

To install lmfit-varpro, run this command in your terminal:

```
$ pip install lmfit-varpro
```

This is the preferred method to install lmfit-varpro, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

3.2 From sources

The sources for lmfit-varpro can be downloaded from the Github repo.

You can simply use pip to install it directly from the Github repo.

```
$ pip install git+https://github.com/glotaran/lmfit-varpro.git
```

Or you can either clone the public repository:

```
$ git clone git://github.com/glotaran/lmfit-varpro
```

Or download the tarball:

```
$ curl  -OL https://github.com/glotaran/lmfit-varpro/tarball/master
```

And once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 4

Usage

To use lmfit-varpro in a project:

```
import lmfit_varpro
```

CHAPTER 5

API Documentation

The API Documentation for lmfit_varpro is automatically created from its docstrings.

constraints	
qr_decomposition	
result	
separable_model	
util	

5.1 constraints

5.1.1 Classes

Summary

CompartmentEqualityConstraint	An CompartmentEqualityConstraint adds a penalty to the residual if 2 compartments of the e matrix differ more than by just a scaling parameter in the sum over a given range.

CompartmentEqualityConstraint

class CompartmentEqualityConstraint(*weight*, *i*, *j*, *parameter*, *erange*, *crange*)

Bases: `object`

An CompartmentEqualityConstraint adds a penalty to the residual if 2 compartments of the e matrix differ more than by just a scaling parameter in the sum over a given range. It calculates as

penalty = weight * (parameter * sum(c[range, i]) - c[range, j])

Methods Summary

calculate

calculate

CompartmentEqualityConstraint.**calculate**(*e_matrix*, *parameter*)

Methods Documentation

calculate(*e_matrix*, *parameter*)

crange = None
The range on the c matrix axis the constraint is applied on

erange = None
The range on the e matrix axis the constraint is applied on

i = None
Index of the first compartment

j = None
Index of the second compartment

parameter = None
Index of the parameter

weight = None
Weight factor of the penalty

5.2 qr_decomposition

5.2.1 Functions

Summary

qr_coefficents
qr_residual

qr_coefficents

qr_coefficents(*A*, *B*)

qr_residual

qr_residual(*A*, *B*)

5.3 result

5.3.1 Functions

Summary

iter

iter

iter(*data*, *c_matrix*)

5.3.2 Classes

Summary

SeparableModelResult

SeparableModelResult

class SeparableModelResult(*model*, *initial_parameter*, *nnls*, *equality_constraints*, *nan_policy='raise'*, **args*, ***kwargs*)

Bases: `lmfit.minimizer.Minimizer`

Attributes Summary

fitresult	The lmfit.MinimizerResult returned by the minimization.
values	Return Parameter values in a simple dictionary.

Methods Summary

ampgo	Finds the global minimum of a multivariate function using the AMPGO (Adaptive Memory Programming for Global Optimization) algorithm.
basinhopping	Use the *basinhopping* algorithm to find the global minimum of a function.
brute	Use the *brute* method to find the global minimum of a function.
c_matrix	
e_matrix	
emcee	Bayesian sampling of the posterior distribution using *emcee*.
eval	

Continued on next page

Table 8 – continued from previous page

final_residual	
final_residual_svd	
fit	
get_model	
least_squares	Least-squares minimization using scipy.optimize.least_squares.
leastsq	Use Levenberg-Marquardt minimization to perform a fit.
minimize	Perform the minimization.
penalty	Penalty function for scalar minimizers.
prepare_fit	Prepare parameters for fitting.
scalar_minimize	Scalar minimization using scipy.optimize.minimize.
unprepare_fit	Clean fit state, so that subsequent fits need to call prepare_fit().

ampgo

SeparableModelResult.**ampgo**(*params=None*, ***kws*)

Finds the global minimum of a multivariate function using the AMPGO (Adaptive Memory Programming for Global Optimization) algorithm.

Parameters

- **params** (Parameters, optional) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.
- ****kws** (*dict, optional*) – Minimizer options to pass to the ampgo algorithm, the options are listed below:

```
local: str (default is 'L-BFGS-B')
    Name of the local minimization method. Valid options␣
↪are:
    - 'L-BFGS-B'
    - 'Nelder-Mead'
    - 'Powell'
    - 'TNC'
    - 'SLSQP'
local_opts: dict (default is None)
    Options to pass to the local minimizer.
maxfunevals: int (default is None)
    Maximum number of function evaluations. If None, the␣
↪optimization will stop
    after `totaliter` number of iterations.
totaliter: int (default is 20)
    Maximum number of global iterations.
maxiter: int (default is 5)
    Maximum number of `Tabu Tunneling` iterations during␣
↪each global iteration.
glbtol: float (default is 1e-5)
    Tolerance whether or not to accept a solution after a␣
↪tunneling phase.
eps1: float (default is 0.02)
    Constant used to define an aspiration value for the␣
↪objective function during
    the Tunneling phase.
eps2: float (default is 0.1)
```

(continues on next page)

(continued from previous page)

```
    Perturbation factor used to move away from the latest␣
↪local minimum at the
    start of a Tunneling phase.
tabulistsize: int (default is 5)
    Size of the (circular) tabu search list.
tabustrategy: str (default is 'farthest')
    Strategy to use when the size of the tabu list exceeds␣
↪`tabulistsize`. It
    can be 'oldest' to drop the oldest point from the tabu␣
↪list or 'farthest'
    to drop the element farthest from the last local␣
↪minimum found.
disp: bool (default is False)
    Set to True to print convergence messages.
```

Returns Object containing the parameters from the ampgo method, with fit parameters, statistics and such. The return values (*x0*, *fval*, *eval*, *msg*, *tunnel*) are stored as *ampgo_<parname>* attributes.

Return type `MinimizerResult`

New in version 0.9.10.

Notes

The Python implementation was written by Andrea Gavana in 2014 (http://infinity77.net/global_optimization/index.html).

The details of the AMPGO algorithm are described in the paper "Adaptive Memory Programming for Constrained Global Optimization" located here:

http://leeds-faculty.colorado.edu/glover/fred%20pubs/416%20-%20AMP%20(TS)%20for%20Constrained%20Global%20Opt%20w%20Lasdon%20et%20al%20.pdf

basinhopping

`SeparableModelResult.`**`basinhopping`**(*params=None*, ***kws*)

Use the *basinhopping* algorithm to find the global minimum of a function.

This method calls scipy.optimize.basinhopping using the default arguments. The default minimizer is *BFGS*, but since lmfit supports parameter bounds for all minimizers, the user can choose any of the solvers present in scipy.optimize.minimize.

Parameters **`params`** (`Parameters` object, optional) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.

Returns Object containing the optimization results from the basinhopping algorithm.

Return type `MinimizerResult`

New in version 0.9.10.

brute

`SeparableModelResult.`**`brute`**(*params=None*, *Ns=20*, *keep=50*)

Use the *brute* method to find the global minimum of a function.

The following parameters are passed to scipy.optimize.brute and cannot be changed:

brute() arg	Value	Description
full_output	1	Return the evaluation grid and the objective function's values on it.
finish	None	No "polishing" function is to be used after the grid search.
disp	False	Do not print convergence messages (when finish is not None).

It assumes that the input Parameters have been initialized, and a function to minimize has been properly set up.

Parameters

- **params** (Parameters, optional) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.
- **Ns** (*int, optional*) – Number of grid points along the axes, if not otherwise specified (see Notes).
- **keep** (*int, optional*) – Number of best candidates from the brute force method that are stored in the candidates attribute. If 'all', then all grid points from scipy.optimize.brute are stored as candidates.

Returns Object containing the parameters from the brute force method. The return values (*x0*, *fval*, *grid*, *Jout*) from scipy.optimize.brute are stored as *brute_<parname>* attributes. The *MinimizerResult* also contains the *candidates* attribute and *show_candidates()* method. The *candidates* attribute contains the parameters and chisqr from the brute force method as a namedtuple, ('Candidate', ['params', 'score']), sorted on the (lowest) chisqr value. To access the values for a particular candidate one can use *result.candidate[#].params* or *result.candidate[#].score*, where a lower # represents a better candidate. The *show_candidates(#)* uses the pretty_print() method to show a specific candidate-# or all candidates when no number is specified.

Return type MinimizerResult

New in version 0.9.6.

Notes

The *brute()* method evalutes the function at each point of a multidimensional grid of points. The grid points are generated from the parameter ranges using *Ns* and (optional) *brute_step*. The implementation in scipy.optimize.brute requires finite bounds and the *range* is specified as a two-tuple *(min, max)* or slice-object *(min, max, brute_step)*. A slice-object is used directly, whereas a two-tuple is converted to a slice object that interpolates *Ns* points from *min* to *max*, inclusive.

In addition, the *brute()* method in lmfit, handles three other scenarios given below with their respective slice-object:

- **lower bound (min) and brute_step are specified:** range = (*min*, *min* + *Ns* * *brute_step*, *brute_step*).
- **upper bound (max) and brute_step are specified:** range = (*max* - *Ns* * *brute_step*, *max*, *brute_step*).
- **numerical value (value) and brute_step are specified:** range = (*value* - (*Ns*//2) * *brute_step*, *value* + (*Ns*//2) * *brute_step*, *brute_step*).

c_matrix

SeparableModelResult.**c_matrix**(**args*, ***kwargs*)

e_matrix

SeparableModelResult.**e_matrix**(**args*, ***kwargs*)

emcee

SeparableModelResult.**emcee**(*params=None*, *steps=1000*, *nwalkers=100*, *burn=0*, *thin=1*, *ntemps=1*, *pos=None*, *reuse_sampler=False*, *workers=1*, *float_behavior='posterior'*, *is_weighted=True*, *seed=None*)

Bayesian sampling of the posterior distribution using *emcee*.

Bayesian sampling of the posterior distribution for the parameters using the *emcee* Markov Chain Monte Carlo package. The method assumes that the prior is Uniform. You need to have *emcee* installed to use this method.

Parameters

- **params** (Parameters, optional) – Parameters to use as starting point. If this is not specified then the Parameters used to initialize the Minimizer object are used.
- **steps** (*int, optional*) – How many samples you would like to draw from the posterior distribution for each of the walkers?
- **nwalkers** (*int, optional*) – Should be set so $nwalkers >> nvarys$, where *nvarys* are the number of parameters being varied during the fit. "Walkers are the members of the ensemble. They are almost like separate Metropolis-Hastings chains but, of course, the proposal distribution for a given walker depends on the positions of all the other walkers in the ensemble." - from the *emcee* webpage.
- **burn** (*int, optional*) – Discard this many samples from the start of the sampling regime.
- **thin** (*int, optional*) – Only accept 1 in every *thin* samples.
- **ntemps** (*int, optional*) – If *ntemps > 1* perform a Parallel Tempering.
- **pos** (*numpy.ndarray, optional*) – Specify the initial positions for the sampler. If *ntemps == 1* then *pos.shape* should be *(nwalkers, nvarys)*. Otherwise, *(ntemps, nwalkers, nvarys)*. You can also initialise using a previous chain that had the same *ntemps*, *nwalkers* and *nvarys*. Note that *nvarys* may be one larger than you expect it to be if your *userfcn* returns an array and *is_weighted is False*.
- **reuse_sampler** (*bool, optional*) – If you have already run *emcee* on a given *Minimizer* object then it possesses an internal `sampler` attribute. You can continue to draw from the same sampler (retaining the chain history) if you set this option to True. Otherwise a new sampler is created. The *nwalkers*, *ntemps*, *pos*, and *params* keywords are ignored with this option. **Important**: the Parameters used to create the sampler must not change in-between calls to *emcee*. Alteration of Parameters would include changed `min`, `max`, `vary` and `expr` attributes. This may happen, for example, if you use an altered Parameters object and call the *minimize* method in-between calls to *emcee*.
- **workers** (*Pool-like or int, optional*) – For parallelization of sampling. It can be any Pool-like object with a map method that follows the same calling sequence as the built-in *map* function. If int is given as the argument, then a multiprocessing-based pool is spawned internally with the corresponding number of parallel processes. 'mpi4py'-based parallelization and 'joblib'-based parallelization pools can also be used here. **Note**: because of multiprocessing overhead it may only be worth parallelising if the objective function is expensive to calculate, or if there are a large number of objective evaluations per step (*ntemps * nwalkers * nvarys*).

- **float_behavior** (*str, optional*) – Specifies meaning of the objective function output if it returns a float. One of:
  - 'posterior' - objective function returns a log-posterior probability
  - 'chi2' - objective function returns χ^2

  See Notes for further details.
- **is_weighted** (*bool, optional*) – Has your objective function been weighted by measurement uncertainties? If *is_weighted is True* then your objective function is assumed to return residuals that have been divided by the true measurement uncertainty *(data - model) / sigma*. If *is_weighted is False* then the objective function is assumed to return unweighted residuals, *data - model*. In this case *emcee* will employ a positive measurement uncertainty during the sampling. This measurement uncertainty will be present in the output params and output chain with the name *__lnsigma*. A side effect of this is that you cannot use this parameter name yourself. **Important** this parameter only has any effect if your objective function returns an array. If your objective function returns a float, then this parameter is ignored. See Notes for more details.
- **seed** (int or *numpy.random.RandomState*, optional) – If *seed* is an int, a new *numpy.random.RandomState* instance is used, seeded with *seed*. If *seed* is already a *numpy.random.RandomState* instance, then that *numpy.random.RandomState* instance is used. Specify *seed* for repeatable minimizations.

Returns MinimizerResult object containing updated params, statistics, etc. The updated params represent the median (50th percentile) of all the samples, whilst the parameter uncertainties are half of the difference between the 15.87 and 84.13 percentiles. The *MinimizerResult* also contains the `chain`, `flatchain` and `lnprob` attributes. The `chain` and `flatchain` attributes contain the samples and have the shape *(nwalkers, (steps - burn) // thin, nvarys)* or *(ntemps, nwalkers, (steps - burn) // thin, nvarys)*, depending on whether Parallel tempering was used or not. *nvarys* is the number of parameters that are allowed to vary. The `flatchain` attribute is a *pandas.DataFrame* of the flattened chain, *chain.reshape(-1, nvarys)*. To access flattened chain values for a particular parameter use *result.flatchain[parname]*. The `lnprob` attribute contains the log probability for each sample in `chain`. The sample with the highest probability corresponds to the maximum likelihood estimate.

Return type `MinimizerResult`

Notes

This method samples the posterior distribution of the parameters using Markov Chain Monte Carlo. To do so it needs to calculate the log-posterior probability of the model parameters, *F*, given the data, *D*, $\ln p(F_{true}|D)$. This 'posterior probability' is calculated as:

$$\ln p(F_{true}|D) \propto \ln p(D|F_{true}) + \ln p(F_{true})$$

where $\ln p(D|F_{true})$ is the 'log-likelihood' and $\ln p(F_{true})$ is the 'log-prior'. The default log-prior encodes prior information already known about the model. This method assumes that the log-prior probability is *-numpy.inf* (impossible) if the one of the parameters is outside its limits. The log-prior probability term is zero if all the parameters are inside their bounds (known as a uniform prior). The log-likelihood function is given by[1]:

$$\ln p(D|F_{true}) = -\frac{1}{2}\sum_n \left[\frac{(g_n(F_{true}) - D_n)^2}{s_n^2} + \ln(2\pi s_n^2)\right]$$

The first summand in the square brackets represents the residual for a given datapoint (g being the generative model, D_n the data and s_n the standard deviation, or measurement uncertainty,

[1] http://dan.iel.fm/emcee/current/user/line/

of the datapoint). This term represents χ^2 when summed over all data points. Ideally the objective function used to create *lmfit.Minimizer* should return the log-posterior probability, $\ln p(F_{true}|D)$. However, since the in-built log-prior term is zero, the objective function can also just return the log-likelihood, unless you wish to create a non-uniform prior.

If a float value is returned by the objective function then this value is assumed by default to be the log-posterior probability, i.e. *float_behavior is 'posterior'*. If your objective function returns χ^2, then you should use a value of *'chi2'* for *float_behavior*. *emcee* will then multiply your χ^2 value by -0.5 to obtain the posterior probability.

However, the default behaviour of many objective functions is to return a vector of (possibly weighted) residuals. Therefore, if your objective function returns a vector, *res*, then the vector is assumed to contain the residuals. If *is_weighted is True* then your residuals are assumed to be correctly weighted by the standard deviation (measurement uncertainty) of the data points (*res = (data - model) / sigma*) and the log-likelihood (and log-posterior probability) is calculated as: *-0.5 * numpy.sum(res**2)*. This ignores the second summand in the square brackets. Consequently, in order to calculate a fully correct log-posterior probability value your objective function should return a single value. If *is_weighted is False* then the data uncertainty, *s_n*, will be treated as a nuisance parameter and will be marginalized out. This is achieved by employing a strictly positive uncertainty (homoscedasticity) for each data point, $s_n = \exp(\_\_lnsigma)$. *__lnsigma* will be present in *MinimizerResult.params*, as well as *Minimizer.chain*, *nvarys* will also be increased by one.

References

eval

`SeparableModelResult.`**`eval`**`(`**args*, ***kwargs*`)`

final_residual

`SeparableModelResult.`**`final_residual`**`(`**args*, ***kwargs*`)`

final_residual_svd

`SeparableModelResult.`**`final_residual_svd`**`(`**args*, ***kwargs*`)`

fit

`SeparableModelResult.`**`fit`**`(`**args*, ***kwargs*`)`

get_model

`SeparableModelResult.`**`get_model`**`()`

least_squares

`SeparableModelResult.`**`least_squares`**`(`*params=None*, ***kws*`)`

Least-squares minimization using scipy.optimize.least_squares.

This method wraps scipy.optimize.least_squares, which has inbuilt support for bounds and robust loss functions. By default it uses the Trust Region Reflective algorithm with a linear loss function (i.e., the standard least-squares problem).

Parameters

- **params** (Parameters, optional) – Parameters to use as starting point.
- ****kws** (*dict, optional*) – Minimizer options to pass to scipy.optimize.least_squares.

Returns Object containing the optimized parameter and several goodness-of-fit statistics.

Return type MinimizerResult

Changed in version 0.9.0: Return value changed to MinimizerResult.

leastsq

SeparableModelResult.**leastsq**(*params=None*, ***kws*)

Use Levenberg-Marquardt minimization to perform a fit.

It assumes that the input Parameters have been initialized, and a function to minimize has been properly set up. When possible, this calculates the estimated uncertainties and variable correlations from the covariance matrix.

This method calls scipy.optimize.leastsq. By default, numerical derivatives are used, and the following arguments are set:

leastsq() arg	Default Value	Description
xtol	1.e-7	Relative error in the approximate solution
ftol	1.e-7	Relative error in the desired sum of squares
maxfev	2000*(nvar+1)	Maximum number of function calls (nvar= # of variables)
Dfun	None	Function to call for Jacobian calculation

Parameters

- **params** (Parameters, optional) – Parameters to use as starting point.
- ****kws** (*dict, optional*) – Minimizer options to pass to scipy.optimize.leastsq.

Returns Object containing the optimized parameter and several goodness-of-fit statistics.

Return type MinimizerResult

Changed in version 0.9.0: Return value changed to MinimizerResult.

minimize

SeparableModelResult.**minimize**(*method='leastsq'*, *params=None*, ***kws*)

Perform the minimization.

Parameters

- **method** (*str, optional*) – Name of the fitting method to use. Valid values are:
  - *'leastsq'*: Levenberg-Marquardt (default)
  - *'least_squares'*: Least-Squares minimization, using Trust Region Reflective method by default
  - *'differential_evolution'*: differential evolution
  - *'brute'*: brute force method
  - *'basinhopping'*: basinhopping
  - *'ampgo'*: Adaptive Memory Programming for Global Optimization
  - '*nelder*': Nelder-Mead
  - *'lbfgsb'*: L-BFGS-B
  - *'powell'*: Powell
  - *'cg'*: Conjugate-Gradient
  - *'newton'*: Newton-CG
  - *'cobyla'*: Cobyla
  - *'bfgs'*: BFGS
  - *'tnc'*: Truncated Newton
  - *'trust-ncg'*: Newton-CG trust-region
  - *'trust-exact'*: nearly exact trust-region (SciPy >= 1.0)
  - *'trust-krylov'*: Newton GLTR trust-region (SciPy >= 1.0)
  - *'trust-constr'*: trust-region for constrained optimization (SciPy >= 1.1)
  - *'dogleg'*: Dog-leg trust-region
  - *'slsqp'*: Sequential Linear Squares Programming

  In most cases, these methods wrap and use the method with the same name from *scipy.optimize*, or use *scipy.optimize.minimize* with the same *method* argument. Thus '*leastsq*' will use *scipy.optimize.leastsq*, while '*powell*' will use *scipy.optimize.minimizer(..., method='powell')*

  For more details on the fitting methods please refer to the SciPy docs.
- **params** (Parameters, optional) – Parameters of the model to use as starting values.
- ****kws** (*optional*) – Additional arguments are passed to the underlying minimization method.

Returns Object containing the optimized parameter and several goodness-of-fit statistics.

Return type MinimizerResult

Changed in version 0.9.0: Return value changed to MinimizerResult.

penalty

SeparableModelResult.**penalty**(*fvars*)

Penalty function for scalar minimizers.

Parameters **fvars** (*numpy.ndarray*) – Array of values for the variable parameters.

Returns

r – The evaluated user-supplied objective function.

If the objective function is an array of size greater than 1, use the scalar returned by *self.reduce_fcn*. This defaults to sum-of-squares, but can be replaced by other options.

Return type float

prepare_fit

SeparableModelResult.**prepare_fit**(*params=None*)

Prepare parameters for fitting.

Prepares and initializes model and Parameters for subsequent fitting. This routine prepares the conversion of `Parameters` into fit variables, organizes parameter bounds, and parses, "compiles" and checks constrain expressions. The method also creates and returns a new instance of a `MinimizerResult` object that contains the copy of the Parameters that will actually be varied in the fit.

Parameters **params** (`Parameters`, optional) – Contains the Parameters for the model; if None, then the Parameters used to initialize the Minimizer object are used.

Returns

Return type `MinimizerResult`

Notes

This method is called directly by the fitting methods, and it is generally not necessary to call this function explicitly.

Changed in version 0.9.0: Return value changed to `MinimizerResult`.

scalar_minimize

SeparableModelResult.**scalar_minimize**(*method='Nelder-Mead'*, *params=None*, ***kws*)

Scalar minimization using scipy.optimize.minimize.

Perform fit with any of the scalar minimization algorithms supported by scipy.optimize.minimize. Default argument values are:

scalar_minimize() arg	Default Value	Description
method	Nelder-Mead	fitting method
tol	1.e-7	fitting and parameter tolerance
hess	None	Hessian of objective function

Parameters

- **method** (*str, optional*) – Name of the fitting method to use. One of:
  - 'Nelder-Mead' (default)
  - 'L-BFGS-B'
  - 'Powell'
  - 'CG'
  - 'Newton-CG'
  - 'COBYLA'
  - 'BFGS'
  - 'TNC'
  - 'trust-ncg'
  - 'trust-exact' (SciPy >= 1.0)
  - 'trust-krylov' (SciPy >= 1.0)
  - 'trust-constr' (SciPy >= 1.1)
  - 'dogleg'
  - 'SLSQP'
  - 'differential_evolution'
- **params** (Parameters, optional) – Parameters to use as starting point.
- ****kws** (*dict, optional*) – Minimizer options pass to scipy.optimize.minimize.

Returns Object containing the optimized parameter and several goodness-of-fit statistics.

Return type MinimizerResult

Changed in version 0.9.0: Return value changed to MinimizerResult.

Notes

If the objective function returns a NumPy array instead of the expected scalar, the sum of squares of the array will be used.

Note that bounds and constraints can be set on Parameters for any of these methods, so are not supported separately for those designed to use bounds. However, if you use the differential_evolution method you must specify finite (min, max) for each varying Parameter.

unprepare_fit

SeparableModelResult.**unprepare_fit**()
Clean fit state, so that subsequent fits need to call prepare_fit().

removes AST compilations of constraint expressions.

Methods Documentation

ampgo(*params=None*, ***kws*)
Finds the global minimum of a multivariate function using the AMPGO (Adaptive Memory Programming for Global Optimization) algorithm.

Parameters

- **params** (Parameters, optional) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.
- ****kws** (*dict, optional*) – Minimizer options to pass to the ampgo algorithm, the options are listed below:

```
local: str (default is 'L-BFGS-B')
    Name of the local minimization method. Valid options␣
↪are:
    - 'L-BFGS-B'
    - 'Nelder-Mead'
    - 'Powell'
    - 'TNC'
    - 'SLSQP'
local_opts: dict (default is None)
    Options to pass to the local minimizer.
maxfunevals: int (default is None)
    Maximum number of function evaluations. If None, the␣
↪optimization will stop
    after `totaliter` number of iterations.
totaliter: int (default is 20)
    Maximum number of global iterations.
maxiter: int (default is 5)
    Maximum number of `Tabu Tunneling` iterations during␣
↪each global iteration.
glbtol: float (default is 1e-5)
    Tolerance whether or not to accept a solution after␣
↪a tunneling phase.
eps1: float (default is 0.02)
    Constant used to define an aspiration value for the␣
↪objective function during
    the Tunneling phase.
eps2: float (default is 0.1)
    Perturbation factor used to move away from the␣
↪latest local minimum at the
    start of a Tunneling phase.
tabulistsize: int (default is 5)
    Size of the (circular) tabu search list.
tabustrategy: str (default is 'farthest')
    Strategy to use when the size of the tabu list␣
↪exceeds `tabulistsize`. It
    can be 'oldest' to drop the oldest point from the␣
↪tabu list or 'farthest'
```

(continues on next page)

(continued from previous page)

```
    to drop the element farthest from the last local␣
↪minimum found.
disp: bool (default is False)
    Set to True to print convergence messages.
```

Returns Object containing the parameters from the ampgo method, with fit parameters, statistics and such. The return values (*x0*, *fval*, *eval*, *msg*, *tunnel*) are stored as *ampgo_<parname>* attributes.

Return type `MinimizerResult`

New in version 0.9.10.

Notes

The Python implementation was written by Andrea Gavana in 2014 (http://infinity77.net/global_optimization/index.html).

The details of the AMPGO algorithm are described in the paper "Adaptive Memory Programming for Constrained Global Optimization" located here:

http://leeds-faculty.colorado.edu/glover/fred%20pubs/416%20-%20AMP%20(TS)%20for%20Constrained%20Global%20Opt%20w%20Lasdon%20et%20al%20.pdf

`basinhopping` (*params=None*, ***kws*)

Use the *basinhopping* algorithm to find the global minimum of a function.

This method calls scipy.optimize.basinhopping using the default arguments. The default minimizer is *BFGS*, but since lmfit supports parameter bounds for all minimizers, the user can choose any of the solvers present in scipy.optimize.minimize.

Parameters **`params`** (`Parameters` object, optional) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.

Returns Object containing the optimization results from the basinhopping algorithm.

Return type `MinimizerResult`

New in version 0.9.10.

`brute` (*params=None*, *Ns=20*, *keep=50*)

Use the *brute* method to find the global minimum of a function.

The following parameters are passed to scipy.optimize.brute and cannot be changed:

brute() arg	Value	Description
full_output	1	Return the evaluation grid and the objective function's values on it.
finish	None	No "polishing" function is to be used after the grid search.
disp	False	Do not print convergence messages (when finish is not None).

It assumes that the input Parameters have been initialized, and a function to minimize has been properly set up.

Parameters

- **params** (Parameters, optional) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.
- **Ns** (*int, optional*) – Number of grid points along the axes, if not otherwise specified (see Notes).
- **keep** (*int, optional*) – Number of best candidates from the brute force method that are stored in the candidates attribute. If 'all', then all grid points from scipy.optimize.brute are stored as candidates.

Returns Object containing the parameters from the brute force method. The return values (*x0*, *fval*, *grid*, *Jout*) from scipy.optimize.brute are stored as *brute_<parname>* attributes. The *MinimizerResult* also contains the *candidates* attribute and *show_candidates()* method. The *candidates* attribute contains the parameters and chisqr from the brute force method as a namedtuple, ('Candidate', ['params', 'score']), sorted on the (lowest) chisqr value. To access the values for a particular candidate one can use *result.candidate[#].params* or *result.candidate[#].score*, where a lower # represents a better candidate. The *show_candidates(#)* uses the pretty_print() method to show a specific candidate-# or all candidates when no number is specified.

Return type MinimizerResult

New in version 0.9.6.

Notes

The *brute()* method evalutes the function at each point of a multidimensional grid of points. The grid points are generated from the parameter ranges using *Ns* and (optional) *brute_step*. The implementation in scipy.optimize.brute requires finite bounds and the *range* is specified as a two-tuple *(min, max)* or slice-object *(min, max, brute_step)*. A slice-object is used directly, whereas a two-tuple is converted to a slice object that interpolates *Ns* points from *min* to *max*, inclusive.

In addition, the *brute()* method in lmfit, handles three other scenarios given below with their respective slice-object:

- **lower bound (min) and brute_step are specified:** range = (*min*, *min* + *Ns* * *brute_step*, *brute_step*).
- **upper bound (max) and brute_step are specified:** range = (*max* - *Ns* * *brute_step*, *max*, *brute_step*).
- **numerical value (value) and brute_step are specified:** range = (*value* - (*Ns*//2) * *brute_step*, *value* + (*Ns*//2) * *brute_step*, *brute_step*).

c_matrix(**args*, ***kwargs*)

e_matrix(**args*, ***kwargs*)

emcee(*params=None*, *steps=1000*, *nwalkers=100*, *burn=0*, *thin=1*, *ntemps=1*, *pos=None*, *reuse_sampler=False*, *workers=1*, *float_behavior='posterior'*, *is_weighted=True*, *seed=None*)

Bayesian sampling of the posterior distribution using *emcee*.

Bayesian sampling of the posterior distribution for the parameters using the *emcee* Markov Chain Monte Carlo package. The method assumes that the prior is Uniform. You need to have *emcee* installed to use this method.

Parameters

- **params** (Parameters, optional) – Parameters to use as starting point. If this is not specified then the Parameters used to initialize the Minimizer object are used.
- **steps** (*int, optional*) – How many samples you would like to draw from the posterior distribution for each of the walkers?
- **nwalkers** (*int, optional*) – Should be set so $nwalkers >> nvarys$, where *nvarys* are the number of parameters being varied during the fit. "Walkers are the members of the ensemble. They are almost like separate Metropolis-Hastings chains but, of course, the proposal distribution for a given walker depends on the positions of all the other walkers in the ensemble." - from the *emcee* webpage.
- **burn** (*int, optional*) – Discard this many samples from the start of the sampling regime.
- **thin** (*int, optional*) – Only accept 1 in every *thin* samples.
- **ntemps** (*int, optional*) – If *ntemps > 1* perform a Parallel Tempering.
- **pos** (*numpy.ndarray, optional*) – Specify the initial positions for the sampler. If *ntemps == 1* then *pos.shape* should be *(nwalkers, nvarys)*. Otherwise, *(ntemps, nwalkers, nvarys)*. You can also initialise using a previous chain that had the same *ntemps*, *nwalkers* and *nvarys*. Note that *nvarys* may be one larger than you expect it to be if your *userfcn* returns an array and *is_weighted is False*.
- **reuse_sampler** (*bool, optional*) – If you have already run *emcee* on a given *Minimizer* object then it possesses an internal `sampler` attribute. You can continue to draw from the same sampler (retaining the chain history) if you set this option to True. Otherwise a new sampler is created. The *nwalkers*, *ntemps*, *pos*, and *params* keywords are ignored with this option. **Important**: the Parameters used to create the sampler must not change in-between calls to *emcee*. Alteration of Parameters would include changed `min`, `max`, `vary` and `expr` attributes. This may happen, for example, if you use an altered Parameters object and call the *minimize* method in-between calls to *emcee*.
- **workers** (*Pool-like or int, optional*) – For parallelization of sampling. It can be any Pool-like object with a map method that follows the same calling sequence as the built-in *map* function. If int is given as the argument, then a multiprocessing-based pool is spawned internally with the corresponding number of parallel processes. 'mpi4py'-based parallelization and 'joblib'-based parallelization pools can also be used here. **Note**: because of multiprocessing overhead it may only be worth parallelising if the objective function is expensive to calculate, or if there are a large number of objective evaluations per step (*ntemps * nwalkers * nvarys*).
- **float_behavior** (*str, optional*) – Specifies meaning of the objective function output if it returns a float. One of:
  - 'posterior' - objective function returns a log-posterior probability
  - 'chi2' - objective function returns χ^2

  See Notes for further details.
- **is_weighted** (*bool, optional*) – Has your objective function been weighted by measurement uncertainties? If *is_weighted is True* then your objective function is assumed to return residuals that have been divided by the

true measurement uncertainty *(data - model) / sigma*. If *is_weighted is False* then the objective function is assumed to return unweighted residuals, *data - model*. In this case *emcee* will employ a positive measurement uncertainty during the sampling. This measurement uncertainty will be present in the output params and output chain with the name *__lnsigma*. A side effect of this is that you cannot use this parameter name yourself. **Important** this parameter only has any effect if your objective function returns an array. If your objective function returns a float, then this parameter is ignored. See Notes for more details.

- **seed** (int or *numpy.random.RandomState*, optional) – If *seed* is an int, a new *numpy.random.RandomState* instance is used, seeded with *seed*. If *seed* is already a *numpy.random.RandomState* instance, then that *numpy.random.RandomState* instance is used. Specify *seed* for repeatable minimizations.

Returns MinimizerResult object containing updated params, statistics, etc. The updated params represent the median (50th percentile) of all the samples, whilst the parameter uncertainties are half of the difference between the 15.87 and 84.13 percentiles. The *MinimizerResult* also contains the `chain`, `flatchain` and `lnprob` attributes. The `chain` and `flatchain` attributes contain the samples and have the shape *(nwalkers, (steps - burn) // thin, nvarys)* or *(ntemps, nwalkers, (steps - burn) // thin, nvarys)*, depending on whether Parallel tempering was used or not. *nvarys* is the number of parameters that are allowed to vary. The `flatchain` attribute is a *pandas.DataFrame* of the flattened chain, *chain.reshape(-1, nvarys)*. To access flattened chain values for a particular parameter use *result.flatchain[parname]*. The `lnprob` attribute contains the log probability for each sample in `chain`. The sample with the highest probability corresponds to the maximum likelihood estimate.

Return type `MinimizerResult`

Notes

This method samples the posterior distribution of the parameters using Markov Chain Monte Carlo. To do so it needs to calculate the log-posterior probability of the model parameters, *F*, given the data, *D*, $\ln p(F_{true}|D)$. This 'posterior probability' is calculated as:

$$\ln p(F_{true}|D) \propto \ln p(D|F_{true}) + \ln p(F_{true})$$

where $\ln p(D|F_{true})$ is the 'log-likelihood' and $\ln p(F_{true})$ is the 'log-prior'. The default log-prior encodes prior information already known about the model. This method assumes that the log-prior probability is *-numpy.inf* (impossible) if the one of the parameters is outside its limits. The log-prior probability term is zero if all the parameters are inside their bounds (known as a uniform prior). The log-likelihood function is given by[1]:

$$\ln p(D|F_{true}) = -\frac{1}{2}\sum_n \left[\frac{(g_n(F_{true}) - D_n)^2}{s_n^2} + \ln(2\pi s_n^2)\right]$$

The first summand in the square brackets represents the residual for a given datapoint (g being the generative model, D_n the data and s_n the standard deviation, or measurement uncertainty, of the datapoint). This term represents χ^2 when summed over all data points. Ideally the objective function used to create *lmfit.Minimizer* should return the log-posterior probability,

[1] http://dan.iel.fm/emcee/current/user/line/

$\ln p(F_{true}|D)$. However, since the in-built log-prior term is zero, the objective function can also just return the log-likelihood, unless you wish to create a non-uniform prior.

If a float value is returned by the objective function then this value is assumed by default to be the log-posterior probability, i.e. *float_behavior is 'posterior'*. If your objective function returns χ^2, then you should use a value of *'chi2'* for *float_behavior*. *emcee* will then multiply your χ^2 value by -0.5 to obtain the posterior probability.

However, the default behaviour of many objective functions is to return a vector of (possibly weighted) residuals. Therefore, if your objective function returns a vector, *res*, then the vector is assumed to contain the residuals. If *is_weighted is True* then your residuals are assumed to be correctly weighted by the standard deviation (measurement uncertainty) of the data points (*res = (data - model) / sigma*) and the log-likelihood (and log-posterior probability) is calculated as: *-0.5 * numpy.sum(res**2)*. This ignores the second summand in the square brackets. Consequently, in order to calculate a fully correct log-posterior probability value your objective function should return a single value. If *is_weighted is False* then the data uncertainty, *s_n*, will be treated as a nuisance parameter and will be marginalized out. This is achieved by employing a strictly positive uncertainty (homoscedasticity) for each data point, $s_n = \exp(\_\_lnsigma)$. *__lnsigma* will be present in *MinimizerResult.params*, as well as *Minimizer.chain*, *nvarys* will also be increased by one.

References

eval(**args*, ***kwargs*)

final_residual(**args*, ***kwargs*)

final_residual_svd(**args*, ***kwargs*)

fit(**args*, ***kwargs*)

fitresult
The lmfit.MinimizerResult returned by the minimization.

get_model()

least_squares(*params=None*, ***kws*)
Least-squares minimization using scipy.optimize.least_squares.

This method wraps scipy.optimize.least_squares, which has inbuilt support for bounds and robust loss functions. By default it uses the Trust Region Reflective algorithm with a linear loss function (i.e., the standard least-squares problem).

Parameters

- **params** (Parameters, optional) – Parameters to use as starting point.
- ****kws** (*dict, optional*) – Minimizer options to pass to scipy.optimize.least_squares.

Returns Object containing the optimized parameter and several goodness-of-fit statistics.

Return type MinimizerResult

Changed in version 0.9.0: Return value changed to MinimizerResult.

leastsq(*params=None*, ***kws*)
Use Levenberg-Marquardt minimization to perform a fit.

It assumes that the input Parameters have been initialized, and a function to minimize has been properly set up. When possible, this calculates the estimated uncertainties and variable correlations from the covariance matrix.

This method calls scipy.optimize.leastsq. By default, numerical derivatives are used, and the following arguments are set:

leastsq() arg	Default Value	Description
xtol	1.e-7	Relative error in the approximate solution
ftol	1.e-7	Relative error in the desired sum of squares
maxfev	2000*(nvar+1)	Maximum number of function calls (nvar= # of variables)
Dfun	None	Function to call for Jacobian calculation

Parameters

- **params** (Parameters, optional) – Parameters to use as starting point.
- ****kws** (*dict, optional*) – Minimizer options to pass to scipy.optimize.leastsq.

Returns Object containing the optimized parameter and several goodness-of-fit statistics.

Return type MinimizerResult

Changed in version 0.9.0: Return value changed to MinimizerResult.

minimize(*method='leastsq'*, *params=None*, ***kws*)
Perform the minimization.

Parameters

- **method** (*str, optional*) – Name of the fitting method to use. Valid values are:
  - *'leastsq'*: Levenberg-Marquardt (default)
  - *'least_squares'*: Least-Squares minimization, using Trust Region Reflective method by default
  - *'differential_evolution'*: differential evolution
  - *'brute'*: brute force method
  - *'basinhopping'*: basinhopping
  - *'ampgo'*: Adaptive Memory Programming for Global Optimization
  - '*nelder*': Nelder-Mead
  - *'lbfgsb'*: L-BFGS-B
  - *'powell'*: Powell
  - *'cg'*: Conjugate-Gradient
  - *'newton'*: Newton-CG
  - *'cobyla'*: Cobyla
  - *'bfgs'*: BFGS

- *'tnc'*: Truncated Newton
- *'trust-ncg'*: Newton-CG trust-region
- *'trust-exact'*: nearly exact trust-region (SciPy >= 1.0)
- *'trust-krylov'*: Newton GLTR trust-region (SciPy >= 1.0)
- *'trust-constr'*: trust-region for constrained optimization (SciPy >= 1.1)
- *'dogleg'*: Dog-leg trust-region
- *'slsqp'*: Sequential Linear Squares Programming

In most cases, these methods wrap and use the method with the same name from *scipy.optimize*, or use *scipy.optimize.minimize* with the same *method* argument. Thus '*leastsq*' will use *scipy.optimize.leastsq*, while '*powell*' will use *scipy.optimize.minimizer(..., method='powell')*

For more details on the fitting methods please refer to the SciPy docs.

- **params** (Parameters, optional) – Parameters of the model to use as starting values.
- ****kws** (*optional*) – Additional arguments are passed to the underlying minimization method.

Returns Object containing the optimized parameter and several goodness-of-fit statistics.

Return type MinimizerResult

Changed in version 0.9.0: Return value changed to MinimizerResult.

penalty(*fvars*)

Penalty function for scalar minimizers.

Parameters **fvars** (*numpy.ndarray*) – Array of values for the variable parameters.

Returns

r – The evaluated user-supplied objective function.

If the objective function is an array of size greater than 1, use the scalar returned by *self.reduce_fcn*. This defaults to sum-of-squares, but can be replaced by other options.

Return type float

prepare_fit(*params=None*)

Prepare parameters for fitting.

Prepares and initializes model and Parameters for subsequent fitting. This routine prepares the conversion of Parameters into fit variables, organizes parameter bounds, and parses, "compiles" and checks constrain expressions. The method also creates and returns a new instance of a MinimizerResult object that contains the copy of the Parameters that will actually be varied in the fit.

Parameters **params** (Parameters, optional) – Contains the Parameters for the model; if None, then the Parameters used to initialize the Minimizer object are used.

Returns

Return type MinimizerResult

Notes

This method is called directly by the fitting methods, and it is generally not necessary to call this function explicitly.

Changed in version 0.9.0: Return value changed to `MinimizerResult`.

scalar_minimize(*method='Nelder-Mead'*, *params=None*, ***kws*)
Scalar minimization using scipy.optimize.minimize.

Perform fit with any of the scalar minimization algorithms supported by scipy.optimize.minimize. Default argument values are:

scalar_minimize() arg	Default Value	Description
method	`Nelder-Mead`	fitting method
tol	1.e-7	fitting and parameter tolerance
hess	None	Hessian of objective function

Parameters

- **method** (*str, optional*) – Name of the fitting method to use. One of:
  - 'Nelder-Mead' (default)
  - 'L-BFGS-B'
  - 'Powell'
  - 'CG'
  - 'Newton-CG'
  - 'COBYLA'
  - 'BFGS'
  - 'TNC'
  - 'trust-ncg'
  - 'trust-exact' (SciPy >= 1.0)
  - 'trust-krylov' (SciPy >= 1.0)
  - 'trust-constr' (SciPy >= 1.1)
  - 'dogleg'
  - 'SLSQP'
  - 'differential_evolution'
- **params** (`Parameters`, optional) – Parameters to use as starting point.
- ****kws** (*dict, optional*) – Minimizer options pass to scipy.optimize.minimize.

Returns Object containing the optimized parameter and several goodness-of-fit statistics.

Return type `MinimizerResult`

Changed in version 0.9.0: Return value changed to `MinimizerResult`.

Notes

If the objective function returns a NumPy array instead of the expected scalar, the sum of squares of the array will be used.

Note that bounds and constraints can be set on Parameters for any of these methods, so are not supported separately for those designed to use bounds. However, if you use the differential_evolution method you must specify finite (min, max) for each varying Parameter.

unprepare_fit()
Clean fit state, so that subsequent fits need to call prepare_fit().

removes AST compilations of constraint expressions.

values
Return Parameter values in a simple dictionary.

5.4 separable_model

5.4.1 Classes

Summary

SeparableModel

SeparableModel

class SeparableModel
Bases: object

Methods Summary

c_matrix
data
e_matrix
eval
fit
retrieve_e_matrix
retrieve_e_matrix_from_c

c_matrix

SeparableModel.**c_matrix**(*parameter*, **args*, ***kwargs*)

data

SeparableModel.**data**(***kwargs*)

e_matrix

SeparableModel.**e_matrix**(*parameter*, **args*, ***kwarg*)

eval

SeparableModel.**eval**(*parameter*, **args*, ***kwargs*)

fit

SeparableModel.**fit**(*initial_parameter*, *nnls*, *constraints*, *nan_policy='raise'*, **args*, ***kwargs*)

retrieve_e_matrix

SeparableModel.**retrieve_e_matrix**(*parameter*, **args*, ***kwargs*)

retrieve_e_matrix_from_c

SeparableModel.**retrieve_e_matrix_from_c**(*c_matrix*, ***kwargs*)

Methods Documentation

c_matrix(*parameter*, **args*, ***kwargs*)

data(***kwargs*)

e_matrix(*parameter*, **args*, ***kwarg*)

eval(*parameter*, **args*, ***kwargs*)

fit(*initial_parameter*, *nnls*, *constraints*, *nan_policy='raise'*, **args*, ***kwargs*)

retrieve_e_matrix(*parameter*, **args*, ***kwargs*)

retrieve_e_matrix_from_c(*c_matrix*, ***kwargs*)

5.5 util

5.5.1 Functions

Summary

dot

dot

dot(*e*, *c*)

CHAPTER 6

Abstract

CHAPTER 7

Scientific Sources

7.1 Published Papers

7.2 Additional Resources

7.3 Citation Key

CHAPTER 8

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

8.1 Types of Contributions

8.1.1 Report Bugs

Report bugs at https://github.com/glotaran/lmfit-varpro/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

8.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

8.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

8.1.4 Write Documentation

lmfit-varpro could always use more documentation, whether as part of the official lmfit-varpro docs, in docstrings, or even on the web in blog posts, articles, and such. If you are writing docstrings please use the NumPyDoc style to write them.

8.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/glotaran/lmfit-varpro/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

8.2 Get Started!

Ready to contribute? Here's how to set up *lmfit-varpro* for local development.

1. Fork the *lmfit-varpro* repo on GitHub.
2. Clone your fork locally:

```
$git clone git@github.com:your_name_here/lmfit-varpro.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$mkvirtualenv lmfit_varpro
$cd lmfit_varpro/
$python setup.py develop
```

4. Create a branch for local development:

```
$git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass all tests (unit tests, codestyle tests and doc creation test):

```
 $tox

To get all requirements run `pip install -r requirements_dev.txt` in your virtualenv.
```

6. Commit your changes and push your branch to GitHub:

```
$git add .
$git commit -m "Your detailed description of your changes."
$git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6 and 3.7. Check https://travis-ci.org/glotaran/lmfit_varpro/pull_requests and make sure that the tests pass for all supported Python versions.

8.4 Tips

To run a subset of tests:

```
$ py.test tests.test_lmfit_varpro
```

8.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 9

"how to" in depth

This section serves as a more complete guide for new developers, as well as place to put useful resources for fast lookup i.e. if you forgot an option for `.. toctree::`.

9.1 Virtual envs "how to" in depth

This Section explains how to get you virtual env up an running with different virtual env providers.

9.1.1 Using conda

The full conda documentation.

Note: This is the recommended way if you use Windows.

Installation

First you have to download anaconda (conda installation with "full" science stack) or miniconda (minimal conda installation) for your OS and follow its install instructions.

After that is done (maybe a restart of the terminal or PC is needed) have the `conda` command available in your terminal:

```
$conda update conda
```

Environment creation

If that is working, create an environment:

```
$conda create --name glotaran python=3.6 -y
```

Note: Python 3.7 could also be used, but packages can't be installed with `conda install packages` right now. If the packages are are on PIPY already they can still be installed with `pip install package`.

De-/Activating an Environment

To activate the environment run:

```
$source activate glotaran
```

Or to deactivate respectively:

```
$source deactivate
```

Note: On default Windows terminal (cmd/PS) you might need omit `source` and run `activate glotaran/deactivate` instead.

Note: To easily manage your conda environments you can use the tool enboard .

Warning: If you want to use `enboard` with git bash on Windows, this won't work out of the box. You will have to edit your `.bash_profile` as follows:

```
export CONDA_ROOT_DIR='/path/to/conda/windows/style' # i.e. mine is 'C:\Anaconda3'
alias python='winpty python'
alias enboard='winpty enboard'
```

9.1.2 Using mkvirtualenv

The full virtualenvwrapper documentation.

Installation

To install `virtualenvwrapper` run:

```
$pip install virtualenvwrapper
$source /usr/local/bin/virtualenvwrapper.sh
```

Note: Depending on your python installation you will have to search for the location of `virtualenvwrapper.sh` and change the path accordingly.

Warning: The line `source /usr/local/bin/virtualenvwrapper.sh` is for Posix Terminals and might not work on Windows terminals.

Environment creation

To create an environment with `virtualenvwrapper` run:

```
$mkvirtualenv glotaran
```

You should now already be in that environment:

```
(glotaran)$
```

De-/Activating an Environment

To change in an existing environment from a fresh terminal run:

```
$workon glotaran
```

Or to deactivate respectively:

```
$deactivate
```

9.1.3 Setting up glotaran

Once you got your environment running you can start contributing to glotaran. Just run the following commands and you are all set:

```
(glotaran)$git clone https://github.com/<your_name_here>/glotaran.git
(glotaran)$cd glotaran
(glotaran)$python -m pip install -r requirements_dev.txt
(glotaran)$pip install -e .
```

9.2 Documentation "how to" in depth

Our documentation is build using Sphinx, which uses `reStructuredText` (and with extensions `Markdown`) to compile documentation as `html`, `LaTeX`, `PDF` and more. It takes care of linking all pages together, building a search index and also extraction the documentation written in the docstrings of the code.

9.2.1 How to use Sphinx in general

First you have enter your virtual env (if you don't know how, have a look here: *Get Started!* or *Virtual envs "how to" in depth*)

When you are in your virtual env (here called `glotaran`) navigate to glotarans *docs* folder:

```
(glotaran)$cd docs
```

Note: Consider for the following steps that, if you are on a Posix system (Linux, MacOS, BSD or Git Bash/migwin on Windows) use `make`, on normal Windows cmd/PS use `make.bat` instead. If your Git Bash is missing the *make* functionality you can follow this guide.

Once you are in the *docs* folder, generating/compiling the documentation is as easy as running:

```
(glotaran)$make html
```

The documentation than can be found is the folder `docs/_build/html`, where you can open it by double clicking `index.html`

Warning: The `reStructuredText` Syntax isn't as forgiving as `html` (where browsers correct most of the falsey). It's more like `LaTeX`, which is why it is recommended to compile often, for errors not to stack up.

It might happen, that you change the documentation and can't see the changes after a refresh in the browser. Since Sphinx to reduce the compile time, it only recompiles the changed files, which can lead to problems if you add new files, because the indexing wasn't updated. If this happens, you can force Sphinx to rebuild the whole documentation by first running:

```
(glotaran)$make clean
```

Workflow

1. Change the docs
2. Build the docs:

   ```
   (glotaran)$make html
   ```

3. Look at the commandline interface and make sure no errors happened.
4. Refresh the you browser to see the changes.
5. If there are no changes, even so there was no error, force Sphinx to rebuild all:

   ```
   (glotaran)$make clean html
   ```

6. Start with step 1 again.

Useful resources:

- Sphinx reST Docs
- Sphinx/reST Memo
- reST Cheatsheet
- Restructured Text (reST) and Sphinx CheatSheet
- Read the Docs Sphinx Theme
- Sphinx Configuration

Often used commands (for Windows replace `make with `make.bat):

- `(glotaran)$make html`
- `(glotaran)$make clean`

- `(glotaran)$make clean html`
- `(glotaran)$make help`

9.2.2 Generate API Documentation

The API Documentation will be generated automatically form the docstrings. Those Docstrings should be formatted in the NumPyDoc style. Please make use of all available features as you see fit.

The features are:

- Parameters
- Returns
- Raises
- See Also
- Notes
- References
- Examples

If you add `packages`, `modules`, `classes`, `methods`, `attributes`, `functions` or `exceptions`, please read the introduction of *Api Documentation Creation Helper*.

Often used commands (for Windows replace `` `make `` with `` `make.bat ``):

- `(glotaran)$make html`
- `(glotaran)$make clean_all`
- `(glotaran)$make api_docs`
- `(glotaran)$make clean_all api_docs html`

Api Documentation Creation Helper

The helper Module to generate the API documentation is located at *docs/generate_api_documentation.py*.

The functionality is available by calling `make api_docs` on a Posix system or `make.bat api_docs` on Windows.

If you add `packages`, `modules`, `classes`, `methods`, `attributes`, `functions` or `exceptions`, you might need to run `make clean_all` on a Posix system or `make.bat clean_all` on Windows to see changes in the documentation.

The generation of the API is done by traversing the main package *traverse_module* and listing all child modules for autosummary to process (see *write_api_documentation*, *api_documentation.rst* and *_templates/api_documentation_template.rst*).

If the child module is also a package all its contained modules will be listed (see *write_known_packages*, *known_packages.rst*, *_templates/known_packages_template.rst* and *_templates/autosummary/module.rst*).

To understand how it works in detail the following links might be of help:

- Sphinx Templating Docs
- Jinja Templating

- Sphinx autosummary Docs
- Sphinx autodoc Docs

CHAPTER 10

Authors

10.1 Original Founder

Ivo van Stokkum <i.h.m.van.stokkum@vu.nl>

10.2 Development Lead

- Joris Snellenburg <j.snellenburg@gmail.com>
- Joern Weissenborn <joern.weissenborn@gmail.com>

10.3 Contributors

Sebastian Weigand <s.weigand.phy@gmail.com>

10.4 Special Thanks

- Stefan Schuetz <YamiNoKeshin@gmail.com>

CHAPTER 11

History

11.1 0.0.3 (2018-08-09)

- exposed *nan_policy* option

11.2 0.0.1 (2018-07-22)

- First release on PyPI.

CHAPTER 12

Credits

Thanks goes to these projects/peoples work that helped us developing:

- cookiecutter (documentation skeleton)
- audreyr/cookiecutter-pypackage (documentation skeleton)

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

lmfit_varpro.constraints, 11
lmfit_varpro.qr_decomposition, 12
lmfit_varpro.result, 13
lmfit_varpro.separable_model, 33
lmfit_varpro.util, 34

Index

A

ampgo() (SeparableModelResult method), 24

B

basinhopping() (SeparableModelResult method), 25
brute() (SeparableModelResult method), 25

C

c_matrix() (SeparableModel method), 34
c_matrix() (SeparableModelResult method), 26
calculate() (CompartmentEqualityConstraint method), 12
CompartmentEqualityConstraint (class in lmfit_varpro.constraints), 11
crange (CompartmentEqualityConstraint attribute), 12

D

data() (SeparableModel method), 34
dot() (in module lmfit_varpro.util), 34

E

e_matrix() (SeparableModel method), 34
e_matrix() (SeparableModelResult method), 26
emcee() (SeparableModelResult method), 26
erange (CompartmentEqualityConstraint attribute), 12
eval() (SeparableModel method), 34
eval() (SeparableModelResult method), 29

F

final_residual() (SeparableModelResult method), 29
final_residual_svd() (SeparableModelResult method), 29
fit() (SeparableModel method), 34
fit() (SeparableModelResult method), 29
fitresult (SeparableModelResult attribute), 29

G

get_model() (SeparableModelResult method), 29

I

i (CompartmentEqualityConstraint attribute), 12
iter() (in module lmfit_varpro.result), 13

J

j (CompartmentEqualityConstraint attribute), 12

L

least_squares() (SeparableModelResult method), 29
leastsq() (SeparableModelResult method), 29
lmfit_varpro.constraints (module), 11
lmfit_varpro.qr_decomposition (module), 12
lmfit_varpro.result (module), 13
lmfit_varpro.separable_model (module), 33
lmfit_varpro.util (module), 34

M

minimize() (SeparableModelResult method), 30

P

parameter (CompartmentEqualityConstraint attribute), 12
penalty() (SeparableModelResult method), 31
prepare_fit() (SeparableModelResult method), 31

Q

qr_coefficents() (in module lmfit_varpro.qr_decomposition), 12
qr_residual() (in module lmfit_varpro.qr_decomposition), 12

R

retrieve_e_matrix() (SeparableModel method), 34
retrieve_e_matrix_from_c() (SeparableModel method), 34

S

scalar_minimize() (SeparableModelResult method), 32
SeparableModel (class in lmfit_varpro.separable_model), 33
SeparableModelResult (class in lmfit_varpro.result), 13

U

unprepare_fit() (SeparableModelResult method), 33

V

values (SeparableModelResult attribute), 33

W

weight (CompartmentEqualityConstraint attribute), 12