Least-Squares Minimization with Constraints for Python

Release 0.7.4

Matthew Newville

January 22, 2014

CONTENTS

The lmfit Python package provides a simple, flexible interface to non-linear optimization or curve fitting problems. The package extends the optimization capabilities of scipy.optimize by replacing floating pointing values for the variables to be optimized with Parameter objects. These Parameters can be fixed or varied, have upper and/or lower bounds placed on its value, or written as an algebraic expression of other Parameters.

The principal advantage of using Parameters instead of simple variables is that the objective function does not have to be rewritten to reflect every change of what is varied in the fit, or what relationships or constraints are placed on the Parameters. This means a scientific programmer can write a general model that encapsulates the phenomenon to be optimized, and then allow user of that model to change what is varied and fixed, what range of values is acceptable for Parameters, and what constraints are placed on the model. The ease with which the model can be changed also allows one to easily test the significance of certain Parameters in a fitting model.

The lmfit package allows a choice of several optimization methods available from scipy.optimize. The default, and by far best tested optimization method used is the Levenberg-Marquardt algorithm from from MINPACK-1 as implemented in scipy.optimize.leastsq. This method is by far the most tested and best support method in lmfit, and much of this document assumes this algorithm is used unless explicitly stated. An important point for many scientific analysis is that this is only method that automatically estimates uncertainties and correlations between fitted variables from the covariance matrix calculated during the fit.

A few other optimization routines are also supported, including Nelder-Mead simplex downhill, Powell's method, COBYLA, Sequential Least Squares methods as implemented in scipy.optimize.fmin, and several others from scipy.optimize. In their native form, some of these methods setting allow upper or lower bounds on parameter variables, or adding constraints on fitted variables. By using Parameter objects, lmfit allows bounds and constraints for *all* of these methods, and makes it easy to swap between methods without hanging the objective function or set of Parameters.

Finally, because the approach derived from MINPACK-1 usin the covariance matrix to determine uncertainties is sometimes questioned (and sometimes rightly so), lmfit supports methods to do a brute force search of the confidence intervals and correlations for sets of parameters.

lmfit and this document are a work in progress.

CHAPTER

ONE

DOWNLOADING AND INSTALLATION

1.1 Prerequisites

The lmfit package requires Python, Numpy, and Scipy. Scipy version 0.11 or higher is recommended, but extensive testing on version compatibility has not been done. Initial tests do work with Python 3.2, but little testing with Python 3 has yet been done. Scipy seems to not yet be available for Python 3.3. No testing has been done with 64-bit architectures, but as this package is pure Python, no significant troubles are expected. Nose is a requirement for running the test suite.

If installed, the uncertainties package will be used for propagation of uncertainties to constrained parameters.

1.2 Downloads

The latest stable version is available from PyPI:

Download Option	Python Versions	Location
Source Kit	2.6, 2.7, 3.2	• lmfit-0.7.4.tar.gz (PyPI)
Win32 Installer	2.6	• lmfit-0.7.4.win32-py2.6.exe (PyPI)
Win32 Installer	2.7	• lmfit-0.7.4.win32-py2.7.exe (PyPI)
Win32 Installer	3.2	• lmfit-0.7.4.win32-py3.2.exe (PyPI)
Development Version	all	use lmfit github repository

if you have Python Setup Tools installed, you can download and install the lmfit-py Package simply with:

```
easy_install -U lmfit
```

1.3 Development Version

To get the latest development version, use:

```
git clone http://github.com/lmfit/lmfit-py.git
```

1.4 Installation

Installation from source on any platform is:

```
python setup.py install
```

1.5 Acknowledgements

LMFIT was originally written by Matthew Newville. Substantial code and documentation improvements, especially for improved estimates of confidence intervals was provided by Till Stensitzki. The implementation of parameter bounds as described in the MINUIT documentation is taken from Jonathan J. Helmus' leastsqbound code, with permission. Many valuable suggestions for improvements have come from Christoph Deil. The code obviously depends on, and owes a very large debt to the code in scipy.optimize. Several discussions on the scipy mailing lists have also led to improvements in this code.

1.6 License

The LMFIT-py code is distribution under the following license:

CHAPTER

TWO

GETTING STARTED WITH NON-LINEAR LEAST-SQUARES FITTING

The lmfit package is designed to provide a simple way to build complex fitting models and apply them to real data. This chapter describes how to set up and perform simple fits. Some basic knowledge of Python, Numpy, and modeling data are assumed.

To do a least-squares fit of a model to data, or for a host of other optimization problems, the main task is to write an *objective function* that takes the values of the fitting variables and calculates either a scalar value to be minimized or an array of values that is to be minimized in the least-squares sense. For many data fitting processes, the least-squares approach is used, and the objective function should return an array of (data-model), perhaps scaled by some weighting factor such as the inverse of the uncertainty in the data. For such a problem, the chi-square (χ^2) statistic is often defined as:

$$\chi^2 = \sum_i^N \frac{[y_i^{\rm meas} - y_i^{\rm model}(\mathbf{v})]^2}{\epsilon_i^2}$$

where $y_i^{\rm meas}$ is the set of measured data, $y_i^{\rm model}(\mathbf{v})$ is the model calculation, $\mathbf{v}$ is the set of variables in the model to be optimized in the fit, and ϵ_i is the estimated uncertainty in the data.

In a traditional non-linear fit, one writes an objective function that takes the variable values and calculates the residual $y_i^{\rm meas} - y_i^{\rm model}(\mathbf{v})$, or the residual scaled by the data uncertainties, $[y_i^{\rm meas} - y_i^{\rm model}(\mathbf{v})]/\epsilon_i$, or some other weighting factor. As a simple example, one might write an objective function like this:

```
def residual(vars, x, data, eps_data):
    amp = vars[0]
    phaseshift = vars[1]
    freq = vars[2]
    decay = vars[3]

    model = amp * sin(x * freq  + phaseshift) * exp(-x*x*decay)

    return (data-model)/eps_data
```

To perform the minimization with scipy, one would do:

```
from scipy.optimize import leastsq
vars = [10.0, 0.2, 3.0, 0.007]
out = leastsq(residual, vars, args=(x, data, eps_data))
```

Though it is wonderful to be able to use python for such optimization problems, and the scipy library is robust and easy to use, the approach here is not terribly different from how one would do the same fit in C or Fortran.

2.1 Using `Parameters` instead of Variables

As described above, there are several practical challenges in doing least-squares fits and other optimizations with the traditional implementation (Fortran, scipy.optimize.leastsq, and most other) in which a list of fitting variables to the function to be minimized. These challenges include:

1. The user has to keep track of the order of the variables, and their meaning – vars[2] is the frequency, and so on.
2. If the user wants to fix a particular variable (*not* vary it in the fit), the residual function has to be altered. While reasonable for simple cases, this quickly becomes significant work for more complex models, and greatly complicates modeling for people not intimately familiar with the code.
3. There is no simple, robust way to put bounds on values for the variables, or enforce mathematical relationships between the variables.

The lmfit module is designed to void these shortcomings.

The main idea of lmfit is to expand a numerical variable with a `Parameter`, which have more attributes than simply their value. Instead of a pass a list of numbers to the function to minimize, you create a `Parameters` object, add parameters to this object, and pass along this object to your function to be minimized. With this transformation, the above example would be translated to look like:

```
from lmfit import minimize, Parameters

def residual(params, x, data, eps_data):
    amp = params['amp'].value
    pshift = params['phase'].value
    freq = params['frequency'].value
    decay = params['decay'].value

    model = amp * sin(x * freq  + pshift) * exp(-x*x*decay)

    return (data-model)/eps_data

params = Parameters()
params.add('amp', value=10)
params.add('decay', value=0.007)
params.add('phase', value=0.2)
params.add('frequency', value=3.0)

out = minimize(residual, params, args=(x, data, eps_data))
```

So far, this simply looks like it replaced a list of values with a dictionary, accessed by name. But each of the named `Parameter` in the `Parameters` object hold additional attributes to modify the value during the fit. For example, Parameters can be fixed or bounded, and this can be done when being defined:

```
params = Parameters()
params.add('amp', value=10, vary=False)
params.add('decay', value=0.007, min=0.0)
params.add('phase', value=0.2)
params.add('frequency', value=3.0, max=10)
```

or after being defined by setting the corresponding attributes:

```
params['amp'].vary = False
params['decay'].min = 0.10
```

In either case, the fit will *not* vary the amplitude parameter. In addition, a lower bound will be placed on the decay factor, and upper bounds placed on two parameters. Importantly, our function to be minimized remains unchanged.

An important point here is that the *params* object can be copied and modified to make many user-level changes to the model and fitting process. Of course, most of the information about how your data is modeled goes into the fitting function, but the approach here allows some external control as well.

2.2 The `Parameter` class

class `Parameter` (*name=None*[, *value=None*[, *vary=True*[, *min=None*[, *max=None*[, *expr=None*]]]]])
create a Parameter object. These are the fundamental extension of a fit variable within lmfit, but you will probably create most of these with the `Parameters` class.

Parameters

- **name** (`None` or string – will be overwritten during fit if `None`.) – parameter name
- **value** – the numerical value for the parameter
- **vary** (boolean (`True`/`False`)) – whether to vary the parameter or not.
- **min** – lower bound for value (`None` = no lower bound).
- **max** – upper bound for value (`None` = no upper bound).
- **expr** (`None` or string) – mathematical expression to use to evaluate value during fit.

Each of these inputs is turned into an attribute of the same name. As above, one hands a dictionary of Parameters to the fitting routines. The name for the Parameter will be set to be consistent

After a fit, a Parameter for a fitted variable (ie with vary = `True`) will have the `value` attribute holding the best-fit value, and may (depending on the success of the fit) have obtain additional attributes.

`stderr`
the estimated standard error for the best-fit value.

`correl`
a dictionary of the correlation with the other fitted variables in the fit, of the form:

```
{'decay': 0.404, 'phase': -0.020, 'frequency': 0.102}
```

For details of the use of the bounds `min` and `max`, see *Bounds Implementation*.

The `expr` attribute can contain a mathematical expression that will be used to compute the value for the Parameter at each step in the fit. See *Using Mathematical Constraints* for more details and examples of this feature.

2.3 The `Parameters` class

class `Parameters`
create a Parameters object. This is little more than a fancy dictionary, with the restrictions that

1. keys must be valid Python symbol names (so that they can be used in expressions of mathematical constraints). This means the names must match `[a-z_][a-z0-9_]*` and cannot be a Python reserved word.
2. values must be valid `Parameter` objects.

Two methods for provided for convenience of initializing Parameters.

`add` (*name*[, *value=None*[, *vary=True*[, *min=None*[, *max=None*[, *expr=None*]]]]])
add a named parameter. This simply creates a `Parameter` object associated with the key *name*, with optional arguments passed to `Parameter`:

```
p = Parameters()
p.add('myvar', value=1, vary=True)
```

add_many (*self*, *paramlist*)
add a list of named parameters. Each entry must be a tuple with the following entries:

```
name, value, vary, min, max, expr
```

That is, this method is somewhat rigid and verbose (no default values), but can be useful when initially defining a parameter list so that it looks table-like:

```
p = Parameters()
#           (Name,  Value,  Vary,   Min,  Max,  Expr)
p.add_many(('amp1',    10,  True, None, None,  None),
           ('cen1',   1.2,  True,  0.5,  2.0,  None),
           ('wid1',   0.8,  True,  0.1, None,  None),
           ('amp2',   7.5,  True, None, None,  None),
           ('cen2',   1.9,  True,  1.0,  3.0,  None),
           ('wid2',  None, False, None, None, '2*wid1/3'))
```

2.4 Simple Example

Putting it all together, a simple example of using a dictionary of `Parameter` objects and `minimize()` might look like this:

```
from lmfit import minimize, Parameters, Parameter, report_fit
import numpy as np

# create data to be fitted
x = np.linspace(0, 15, 301)
data = (5. * np.sin(2 * x - 0.1) * np.exp(-x*x*0.025) +
        np.random.normal(size=len(x), scale=0.2) )

# define objective function: returns the array to be minimized
def fcn2min(params, x, data):
    """ model decaying sine wave, subtract data"""
    amp = params['amp'].value
    shift = params['shift'].value
    omega = params['omega'].value
    decay = params['decay'].value

    model = amp * np.sin(x * omega + shift) * np.exp(-x*x*decay)
    return model - data

# create a set of Parameters
params = Parameters()
params.add('amp',   value= 10,  min=0)
params.add('decay', value= 0.1)
params.add('shift', value= 0.0, min=-np.pi/2., max=np.pi/2)
params.add('omega', value= 3.0)


# do fit, here with leastsq model
result = minimize(fcn2min, params, args=(x, data))

# calculate final result
```

```
final = data + result.residual

# write error report
report_fit(params)

# try to plot results
try:
    import pylab
    pylab.plot(x, data, 'k+')
    pylab.plot(x, final, 'r')
    pylab.show()
except:
    pass
```

CHAPTER

THREE

PERFORMING FITS, ANALYZING OUTPUTS

As shown in the previous sections, a simple fit can be performed with the `minimize()` function. For more sophisticated modeling, the `Minimizer` class can be used to gain a bit more control, especially when using complicated constraints.

3.1 The `minimize()` function

The minimize function takes a function to minimize, a dictionary of `Parameter` , and several optional arguments. See *Writing a Fitting Function* for details on writing the function to minimize.

minimize(*function*, *params*, *args=None*, *kws=None*, *method='leastsq'*, ***leastsq_kws*)

find values for the params so that the sum-of-squares of the returned array from function is minimized.

Parameters

- **function** (*callable.*) – function to return fit residual. See *Writing a Fitting Function* for details.
- **params** (*dict*) – a dictionary of Parameters. Keywords must be strings that match `[a-z_][a-z0-9_]*` and is not a python reserved word. Each value must be `Parameter`.
- **args** (*tuple*) – arguments tuple to pass to the residual function as positional arguments.
- **kws** (*dict*) – dictionary to pass to the residual function as keyword arguments.
- **method** (*string*) – name of fitting method to use. See *Choosing Different Fitting Methods* for details
- **leastsq_kws** (*dict*) – dictionary to pass to scipy.optimize.leastsq

Returns Minimizer object, which can be used to inspect goodness-of-fit statistics, or to re-run fit.

For backward compatibility, the keyword *engine* is retained as a synonym for *method*, but this should be considered deprecated.

On output, the params will be updated with best-fit values and, where appropriate, estimated uncertainties and correlations. See *Goodness-of-Fit and estimated uncertainty and correlations* for further details.

3.2 Writing a Fitting Function

An important component of a fit is writing a function to be minimized in the least-squares sense. Since this function will be called by other routines, there are fairly stringent requirements for its call signature and return value. In principle, your function can be any python callable, but it must look like this:

func(params, *args, **kws):
calculate objective residual to be minimized from parameters.

Parameters

- **params** (*dict*) – parameters.
- **args** – positional arguments. Must match `args` argument to `minimize()`
- **kws** – keyword arguments. Must match `kws` argument to `minimize()`

Returns residual array (generally data-model) to be minimized in the least-squares sense.

Return type numpy array. The length of this array cannot change between calls.

A common use for the positional and keyword arguments would be to pass in other data needed to calculate the residual, including such things as the data array, dependent variable, uncertainties in the data, and other data structures for the model calculation.

The objective function should return the value to be minimized. For the Levenberg-Marquardt algorithm from `leastsq()`, this returned value **must** be an array, with a length greater than or equal to the number of fitting variables in the model. For the other methods, the return value can either be a scalar or an array. If an array is returned, the sum of squares of the array will be sent to the underlying fitting method, effectively doing a least-squares optimization of the return values.

Since the function will be passed in a dictionary of `Parameters`, it is advisable to unpack these to get numerical values at the top of the function. A simple example would look like:

```
def residual(pars, x, data=None, eps=None):
    # unpack parameters:
    #  extract .value attribute for each parameter
    amp = pars['amp'].value
    period = pars['period'].value
    shift = pars['shift'].value
    decay = pars['decay'].value

    if abs(shift) > pi/2:
        shift = shift - sign(shift)*pi

    if abs(period) < 1.e-10:
        period = sign(period)*1.e-10

    model = amp * sin(shift + x/period) * exp(-x*x*decay*decay)

    if data is None:
        return model
    if eps is None:
        return (model - data)
    return (model - data)/eps
```

In this example, `x` is a positional (required) argument, while the `data` array is actually optional (so that the function returns the model calculation if the data is neglected). Also note that the model calculation will divide `x` by the varied value of the 'period' Parameter. It might be wise to make sure this parameter cannot be 0. It would be possible to use the bounds on the `Parameter` to do this:

```
params['period'] = Parameter(value=2, min=1.e-10)
```

but might be wiser to put this directly in the function with:

```
if abs(period) < 1.e-10:
    period = sign(period)*1.e-10
```

3.3 Choosing Different Fitting Methods

By default, the Levenberg-Marquardt algorithm is used for fitting. While often criticized, including the fact it finds a *local* minima, this approach has some distinct advantages. These include being fast, and well-behaved for most curve-fitting needs, and making it easy to estimate uncertainties for and correlations between pairs of fit variables, as discussed in *Goodness-of-Fit and estimated uncertainty and correlations*.

Alternative algorithms can also be used. These include simulated annealing which promises a better ability to avoid local minima, and BFGS, which is a modification of the quasi-Newton method.

To select which of these algorithms to use, use the `method` keyword to the `minimize()` function or use the corresponding method name from the `Minimizer` class as listed in the *Table of Supported Fitting Methods*.

Table of Supported Fitting Methods:

Fitting Method	**method arg to minimize()**	**Minimizer method**	**method arg to scalar_minimize()**
Levenberg-Marquardt	`leastsq`	`leastsq()`	Not available
Nelder-Mead	`nelder`	`fmin()`	`Nelder-Mead`
L-BFGS-B	`lbfgsb`	`lbfgsb()`	`L-BFGS-B`
Simulated Annealing	`anneal`	`anneal()`	`Anneal`
Powell	`powell`		`Powell`
Conjugate Gradient	`cg`		`CG`
Newtown-CG	`newton`		`Newton-CG`
COBYLA	`cobyla`		`COBYLA`
Sequential Linear Squares Programming	`slsqp`		`SLSQP`

Note: Use of `scipy.optimize.minimize()` requires scipy 0.11 or higher.

Note: The objective function for the Levenberg-Marquardt method **must** return an array, with more elements than variables. All other methods can return either a scalar value or an array.

Warning: The Levenberg-Marquardt method is *by far* the most tested fit method, and much of this documentation assumes that this is the method used. For example, many of the fit statistics and estimates for uncertainties in parameters discussed in *Goodness-of-Fit and estimated uncertainty and correlations* are done only for the `leastsq` method.

In particular, the simulated annealing method appears to not work correctly.... understanding this is on the ToDo list.

3.4 Goodness-of-Fit and estimated uncertainty and correlations

On a successful fit using the *leastsq* method, several goodness-of-fit statistics and values related to the uncertainty in the fitted variables will be calculated. These are all encapsulated in the `Minimizer` object for the fit, as returned by `minimize()`. The values related to the entire fit are stored in attributes of the `Minimizer` object, as shown in *Table of Fit Results* while those related to each fitted variables are stored as attributes of the corresponding `Parameter`.

Table of Fit Results: These values, including the standard Goodness-of-Fit statistics, are all attributes of the `Minimizer` object returned by `minimize()`.

`Minimizer` Attribute	Description / Formula
nfev	number of function evaluations
success	boolean (`True`/`False`) for whether fit succeeded.
errorbars	boolean (`True`/`False`) for whether uncertainties were estimated.
message	message about fit success.
ier	integer error value from scipy.optimize.leastsq
lmdif_message	message from scipy.optimize.leastsq
nvarys	number of variables in fit $N_{\rm varys}$
ndata	number of data points: N
nfree ‘	degrees of freedom in fit: $N - N_{\rm varys}$
residual	residual array (return of `func()`: Resid
chisqr	chi-square: $\chi^2 = \sum_i^N [\rm Resid_i]^2$
redchi	reduced chi-square: $\chi^2_\nu = \chi^2/(N - N_{\rm varys})$

Note that the calculation of chi-square and reduced chi-square assume that the returned residual function is scaled properly to the uncertainties in the data. For these statistics to be meaningful, the person writing the function to be minimized must scale them properly.

After a fit using using the `leastsq()` method has completed successfully, standard errors for the fitted variables and correlations between pairs of fitted variables are automatically calculated from the covariance matrix. The standard error (estimated 1σ error-bar) go into the `stderr` attribute of the Parameter. The correlations with all other variables will be put into the `correl` attribute of the Parameter – a dictionary with keys for all other Parameters and values of the corresponding correlation.

In some cases, it may not be possible to estimate the errors and correlations. For example, if a variable actually has no practical effect on the fit, it will likely cause the covariance matrix to be singular, making standard errors impossible to estimate. Placing bounds on varied Parameters makes it more likely that errors cannot be estimated, as being near the maximum or minimum value makes the covariance matrix singular. In these cases, the `errorbars` attribute of the fit result (`Minimizer` object) will be `False`.

3.5 Using the `Minimizer` class

For full control of the fitting process, you'll want to create a `Minimizer` object, or at least use the one returned from the `minimize()` function.

class `Minimizer` (*function*, *params*, *fcn_args=None*, *fcn_kws=None*, *iter_cb=None*, *scale_covar=True*, ***kws*)

creates a Minimizer, for fine-grain access to fitting methods and attributes.

Parameters

- **function** (*callable.*) – objective function to return fit residual. See *Writing a Fitting Function* for details.
- **params** (*dict*) – a dictionary of Parameters. Keywords must be strings that match `[a-z_][a-z0-9_]*` and is not a python reserved word. Each value must be `Parameter`.
- **fcn_args** (*tuple*) – arguments tuple to pass to the residual function as positional arguments.
- **fcn_kws** (*dict*) – dictionary to pass to the residual function as keyword arguments.
- **iter_cb** (callable or `None`) – function to be called at each fit iteration
- **scale_covar** – flag for scaling covariance matrix and uncertainties to reduced chi-square (`leastsq` only)
- **kws** (*dict*) – dictionary to pass as keywords to the underlying scipy.optimize method.

Returns Minimizer object, which can be used to inspect goodness-of-fit statistics, or to re-run fit.

The Minimizer object has a few public methods:

leastsq(*scale_covar=True*, ***kws*)
perform fit with Levenberg-Marquardt algorithm. Keywords will be passed directly to 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

anneal(***kws*)
perform fit with Simulated Annealing. Keywords will be passed directly to scipy.optimize.anneal.

`anneal()` arg	Default Value	Description
schedule	`cauchy`	annealing schedule
maxiter	2000*(nvar+1)	maximum number of iterations

For me, this Simulated Annealing appears to never work.

lbfgsb(***kws*)
perform fit with L-BFGS-B algorithm. Keywords will be passed directly to scipy.optimize.fmin_l_bfgs_b.

`lbfgsb()` arg	Default Value	Description
factr	1000.0	
approx_grad	`True`	calculate approximations of gradient
maxfun	2000*(nvar+1)	maximum number of function calls (nvar= # of variables)

fmin(***kws*)
perform fit with Nelder-Mead downhill simplex algorithm. Keywords will be passed directly to scipy.optimize.fmin.

`fmin()` arg	Default Value	Description
ftol	1.e-4	function tolerance
xtol	1.e-4	parameter tolerance
maxfun	5000*(nvar+1)	maximum number of function calls (nvar= # of variables)

scalar_minimize(*method='Nelder-Mead'*, *hess=None*, *tol=None*, ***kws*)
perform fit with any of the scalar minimization algorithms supported by scipy.optimize.minimize.

`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

prepare_fit(***kws*)
prepares and initializes model and Parameters for subsequent fitting. This routine prepares the conversion of `Parameters` into fit variables, organizes parameter bounds, and parses, checks and "compiles" constrain expressions.

This is called directly by the fitting methods, and it is generally not necessary to call this function explicitly. An exception is when you would like to call your function to minimize prior to running one of the minimization routines, for example, to calculate the initial residual function. In that case, you might want to do something like:

```
myfit = Minimizer(my_residual, params,  fcn_args=(x,), fcn_kws={'data':data})

myfit.prepare_fit()
```

```
init = my_residual(p_fit, x)
pylab.plot(x, init, 'b--')

myfit.leastsq()
```

That is, this method should be called prior to your fitting function being called.

3.6 Getting and Printing Fit Reports

fit_report (*params*, *modelpars=None*, *show_correl=True*, *min_correl=0.1*)
generate and return text of report of best-fit values, uncertainties, and correlations from fit.

Parameters

- **params** – Parameters from fit.
- **modelpars** – Parameters with "Known Values" (optional, default None)
- **show_correl** – whether to show list of sorted correlations [`True`]
- **min_correl** – smallest correlation absolute value to show [0.1]

report_fit (*params*, *modelpars=None*, *show_correl=True*, *min_correl=0.1*)
print text of report from `fit_report()`.

An example fit with an error report:

```
p_true = Parameters()
p_true.add('amp', value=14.0)
p_true.add('period', value=5.33)
p_true.add('shift', value=0.123)
p_true.add('decay', value=0.010)

def residual(pars, x, data=None):
    amp = pars['amp'].value
    per = pars['period'].value
    shift = pars['shift'].value
    decay = pars['decay'].value

    if abs(shift) > pi/2:
        shift = shift - sign(shift)*pi
    model = amp*sin(shift + x/per) * exp(-x*x*decay*decay)
    if data is None:
        return model
    return (model - data)

n = 2500
xmin = 0.
xmax = 250.0
noise = random.normal(scale=0.7215, size=n)
x     = linspace(xmin, xmax, n)
data  = residual(p_true, x) + noise

fit_params = Parameters()
fit_params.add('amp', value=13.0)
fit_params.add('period', value=2)
fit_params.add('shift', value=0.0)
fit_params.add('decay', value=0.02)
```

```
out = minimize(residual, fit_params, args=(x,), kws={'data':data})

fit = residual(fit_params, x)
report_errors(fit_params)
```

would generate this report:

```
[[Variables]]
     amp:        13.969724 +/- 0.050145 (0.36%) initial =  13.000000
     decay:      0.009990 +/- 0.000042 (0.42%) initial =  0.020000
     period:     5.331423 +/- 0.002788 (0.05%) initial =  2.000000
     shift:      0.125333 +/- 0.004938 (3.94%) initial =  0.000000
[[Correlations]] (unreported correlations are <  0.100)
    C(period, shift)             =  0.800
    C(amp, decay)                =  0.576
```

CHAPTER

FOUR

CALCULATION OF CONFIDENCE INTERVALS

Since version *0.5*, lmfit is also capable of calculating the confidence intervals directly. For most models, it is not necessary: the estimation of the standard error from the estimated covariance matrix is normally quite good.

But for some models, e.g. a sum of two exponentials, the approximation begins to fail. For this case, lmfit has the function `conf_interval()` to calculate confidence intervals directly. This is substantially slower than using the errors estimated from the covariance matrix, but the results are more robust.

4.1 Method used for calculating confidence intervals

The F-test is used to compare our null model, which is the best fit we have found, with an alternate model, where one of the parameters is fixed to a specific value. The value is changed until the difference between χ_0^2 and χ_f^2 can't be explained by the loss of a degree of freedom within a certain confidence.

$$F(P_{fix}, N-P) = \left(\frac{\chi_f^2}{\chi_0^2} - 1\right)\frac{N-P}{P_{fix}}$$

N is the number of data-points, P the number of parameter of the null model. P_{fix} is the number of fixed parameters (or to be more clear, the difference of number of parameters between our null model and the alternate model).

A log-likelihood method will be added soon.

4.2 A basic example

First we create a toy problem:

```
In [1]: import lmfit

In [2]: import numpy as np

In [3]: x = np.linspace(0.3,10,100)

In [4]: y = 1/(0.1*x)+2+0.1*np.random.randn(x.size)

In [5]: p = lmfit.Parameters()

In [6]: p.add_many(('a',0.1),('b',1))

In [7]: def residual(p):
   ...:       a = p['a'].value
   ...:       b = p['b'].value
```

```
   ...:     return 1/(a*x)+b-y
   ...:
```

We have to fit it, before we can generate the confidence intervals.

```
In [8]: mi = lmfit.minimize(residual, p)

In [9]: mi.leastsq()
Out[9]: True

In [10]: lmfit.printfuncs.report_fit(mi.params)
[[Variables]]
     a:     0.09997481 +/- 0.0001938203 (0.19%) initial =  0.09997481
     b:     1.991444 +/- 0.01213395 (0.61%) initial =  1.991444
[[Correlations]] (unreported correlations are <  0.100)
    C(a, b)                      =  0.601
```

Now it just a simple function call to start the calculation:

```
In [11]: ci = lmfit.conf_interval(mi)

In [12]: lmfit.printfuncs.report_ci(ci)
     99.70%    95.00%    67.40%     0.00%    67.40%    95.00%    99.70%
a   0.09939   0.09959   0.09978   0.09997   0.10017   0.10036   0.10057
b   1.95451   1.96736   1.97947   1.98414   2.00342   2.01552   2.02838
```

As we can see, the estimated error is almost the same: it is not necessary to calculate ci's for this problem.

4.3 An advanced example

Now we look at a problem, where calculating the error from approximated covariance can lead to wrong results:

```
In [14]: y = 3*np.exp(-x/2.)-5*np.exp(-x/10.)+0.2*np.random.randn(x.size)

In [15]: p = lmfit.Parameters()

In [16]: p.add_many(('a1', 5), ('a2', -5), ('t1', 2), ('t2', 5))

In [17]: def residual(p):
   ....:     a1, a2, t1, t2 = [i.value for i in p.values()]
   ....:     return a1*np.exp(-x/t1)+a2*np.exp(-x/t2)-y
   ....:
```

Now lets fit it:

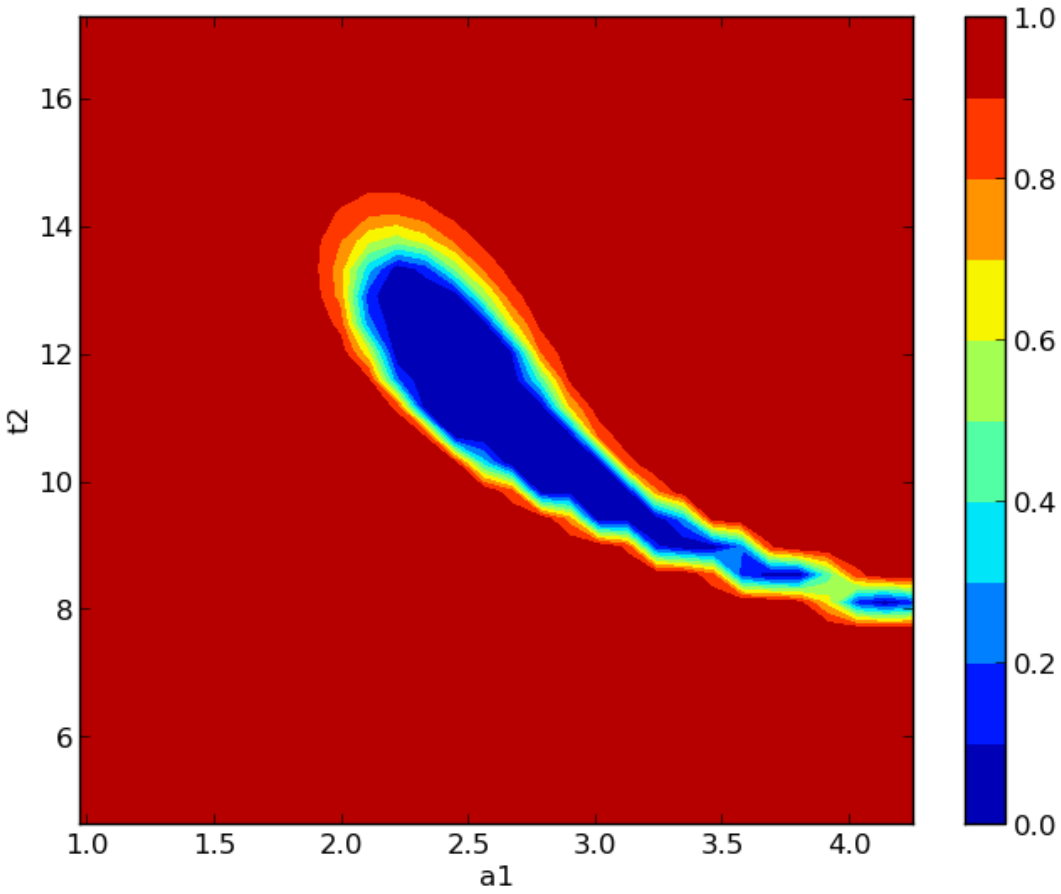
```
In [18]: mi = lmfit.minimize(residual, p)

In [19]: mi.leastsq()
Out[19]: True

In [20]: lmfit.printfuncs.report_fit(mi.params, show_correl=False)
[[Variables]]
     a1:     2.611014 +/- 0.3279648 (12.56%) initial =  2.611014
     a2:    -4.512928 +/- 0.3991997 (8.85%) initial = -4.512928
     t1:     1.569477 +/- 0.3345078 (21.31%) initial =  1.569477
     t2:     10.96137 +/- 1.263874 (11.53%) initial =  10.96137
```

Again we call `conf_interval()`, this time with tracing and only for 1- and 2-sigma:

```
In [21]: ci, trace = lmfit.conf_interval(mi, sigmas=[0.68,0.95], trace=True, verbose=False)

In [22]: lmfit.printfuncs.report_ci(ci)
      95.00%    68.00%     0.00%    68.00%    95.00%
a1   2.11682   2.33695   2.61101   3.06638   4.28694
a2  -6.39449  -5.05978  -4.20174  -4.19527  -3.97856
t2   8.00415   9.62699  12.17331  12.17903  13.34866
t1   1.07010   1.28481   1.37407   1.97509   2.64342
```

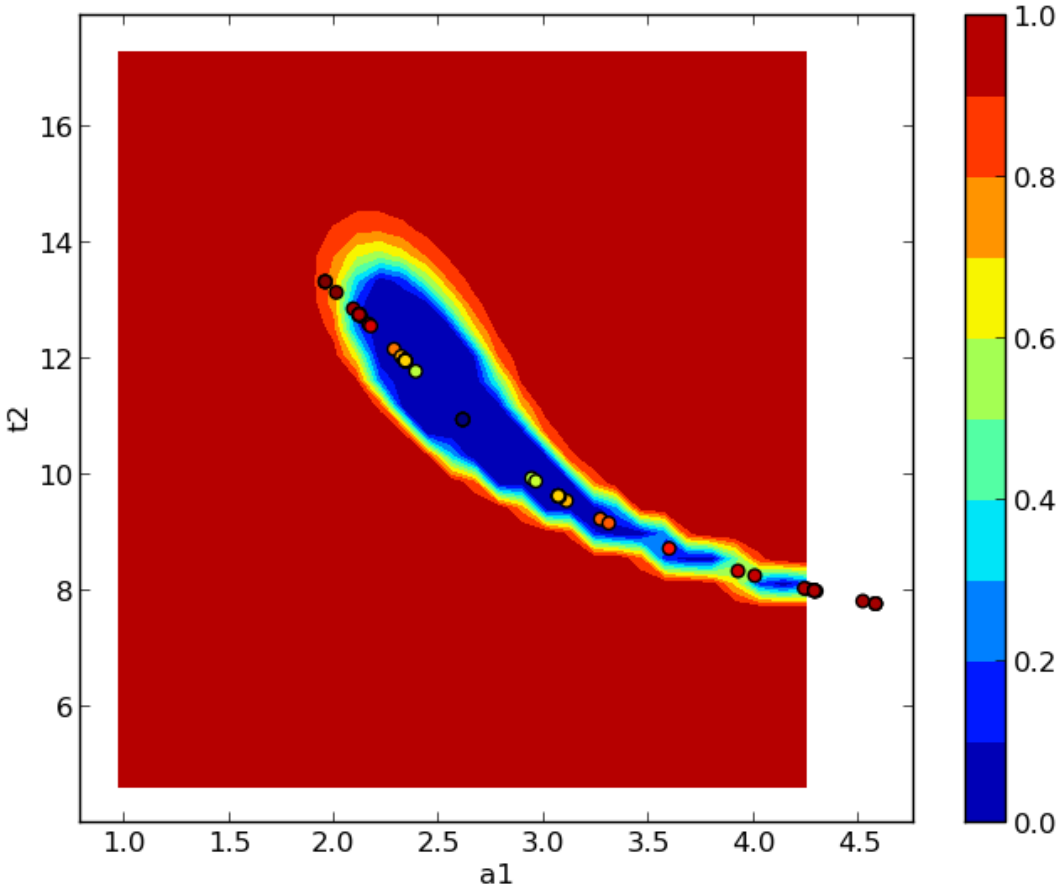
If you compare the calculated error estimates, you will see that the regular estimate is too small. Now let's plot a confidence region:

```
In [23]: import matplotlib.pylab as plt

In [24]: x, y, grid = lmfit.conf_interval2d(mi,'a1','t2',30,30)

In [25]: plt.contourf(x, y, grid, np.linspace(0,1,11))
Out[25]: <matplotlib.contour.QuadContourSet instance at 0xa69ff0c>

In [26]: plt.xlabel('a1');

In [27]: plt.colorbar();

In [28]: plt.ylabel('t2');
```



Remember the trace? It shows the dependence between two parameters.

```
In [33]: x, y, prob = trace['a1']['a1'], trace['a1']['t2'],trace['a1']['prob']

In [34]: x2, y2, prob2 = trace['t2']['t2'], trace['t2']['a1'],trace['t2']['prob']

In [35]: plt.scatter(x, y, c=prob ,s=30)
Out[35]: <matplotlib.collections.PathCollection at 0xa69fccc>

In [36]: plt.scatter(x2, y2, c=prob2, s=30)
Out[36]: <matplotlib.collections.PathCollection at 0xa69f7ac>
```



4.4 Documentation of methods

conf_interval (*minimizer*, *p_names=None*, *sigmas=(0.674, 0.95, 0.997)*, *trace=False*, *maxiter=200*, *verbose=False*, *prob_func=None*)

Calculates the confidence interval for parameters from the given minimizer.

The parameter for which the ci is calculated will be varied, while the remaining parameters are re-optimized for minimizing chi-square. The resulting chi-square is used to calculate the probability with a given statistic e.g. F-statistic. This function uses a 1d-rootfinder from scipy to find the values resulting in the searched confidence region.

Parameters **minimizer** : Minimizer

The minimizer to use, should be already fitted via leastsq.

p_names : list, optional

Names of the parameters for which the ci is calculated. If None, the ci is calculated for every parameter.

sigmas : list, optional

The probabilities (1-alpha) to find. Default is 1,2 and 3-sigma.

trace : bool, optional

Defaults to False, if true, each result of a probability calculation is saved along with the parameter. This can be used to plot so called "profile traces".

Returns **output** : dict

A dict, which contains a list of (sigma, vals)-tuples for each name.

trace_dict : dict

Only if trace is set true. Is a dict, the key is the parameter which was fixed.The values are again a dict with the names as keys, but with an additional key 'prob'. Each contains an array of the corresponding values.

Other Parameters **maxiter** : int

Maximum of iteration to find an upper limit.

prob_func : `None` or callable

Function to calculate the probability from the optimized chi-square. Default (`None`) uses built-in f_compare (F test).

verbose: bool :

print extra debuggin information. Default is `False`.

See also:

`conf_interval2d`

Examples

```
>>> from lmfit.printfuncs import *
>>> mini = minimize(some_func, params)
>>> mini.leastsq()
True
>>> report_errors(params)
... #report
>>> ci = conf_interval(mini)
>>> report_ci(ci)
... #report
```

Now with quantiles for the sigmas and using the trace.

```
>>> ci, trace = conf_interval(mini, sigmas=(0.25, 0.5, 0.75, 0.999), trace=True)
>>> fixed = trace['para1']['para1']
>>> free = trace['para1']['not_para1']
>>> prob = trace['para1']['prob']
```

This makes it possible to plot the dependence between free and fixed.

conf_interval2d(*minimizer*, *x_name*, *y_name*, *nx=10*, *ny=10*, *limits=None*, *prob_func=None*)

Calculates confidence regions for two fixed parameters.

The method is explained in *conf_interval*: here we are fixing two parameters.

Parameters **minimizer** : minimizer

The minimizer to use, should be already fitted via leastsq.

x_name : string

The name of the parameter which will be the x direction.

y_name : string

The name of the parameter which will be the y direction.

nx, ny : ints, optional

Number of points.

limits : tuple: optional

Should have the form ((x_upper, x_lower),(y_upper, y_lower)). If not given, the default is 5 std-errs in each direction.

Returns **x** : (nx)-array

x-coordinates

y : (ny)-array

y-coordinates

grid : (nx,ny)-array

grid contains the calculated probabilities.

Other Parameters **prob_func** : `None` or callable

Function to calculate the probability from the optimized chi-square. Default (`None`) uses built-in f_compare (F test).

Examples

```
>>> mini = minimize(some_func, params)
>>> mini.leastsq()
True
>>> x,y,gr = conf_interval2d('para1','para2')
>>> plt.contour(x,y,gr)
```

CHAPTER

FIVE

BOUNDS IMPLEMENTATION

This section describes the implementation of `Parameter` bounds. The MINPACK-1 implementation used in scipy.optimize.leastsq for the Levenberg-Marquardt algorithm does not explicitly support bounds on parameters, and expects to be able to fully explore the available range of values for any Parameter. Simply placing hard constraints (that is, resetting the value when it exceeds the desired bounds) prevents the algorithm from determining the partial derivatives, and leads to unstable results.

Instead of placing such hard constraints, bounded parameters are mathematically transformed using the formulation devised (and documented) for MINUIT. This is implemented following (and borrowing heavily from) the leastsqbound from J. J. Helmus. Parameter values are mapped from internally used, freely variable values $P_{\rm internal}$ to bounded parameters $P_{\rm bounded}$. When both `min` and `max` bounds are specified, the mapping is

$$\begin{aligned} P_{\rm internal} &= \arcsin\big(\frac{2(P_{\rm bounded} - \min)}{(\max - \min)} - 1\big) \\ P_{\rm bounded} &= \min + \big(\sin(P_{\rm internal}) + 1\big)\frac{(\max - \min)}{2} \end{aligned}$$

With only an upper limit `max` supplied, but `min` left unbounded, the mapping is:

$$\begin{aligned} P_{\rm internal} &= \sqrt{(\max - P_{\rm bounded} + 1)^2 - 1} \\ P_{\rm bounded} &= \max + 1 - \sqrt{P_{\rm internal}^2 + 1} \end{aligned}$$

With only a lower limit `min` supplied, but `max` left unbounded, the mapping is:

$$\begin{aligned} P_{\rm internal} &= \sqrt{(P_{\rm bounded} - \min + 1)^2 - 1} \\ P_{\rm bounded} &= \min - 1 + \sqrt{P_{\rm internal}^2 + 1} \end{aligned}$$

With these mappings, the value for the bounded Parameter cannot exceed the specified bounds, though the internally varied value can be freely varied.

It bears repeating that code from leastsqbound was adopted to implement the transformation described above. The challenging part (Thanks again to Jonathan J. Helmus!) here is to re-transform the covariance matrix so that the uncertainties can be estimated for bounded Parameters. This is included by using the derivate $dP_{\rm internal}/dP_{\rm bounded}$ from the equations above to re-scale the Jacobin matrix before constructing the covariance matrix from it. Tests show that this re-scaling of the covariance matrix works quite well, and that uncertainties estimated for bounded are quite reasonable. Of course, if the best fit value is very close to a boundary, the derivative estimated uncertainty and correlations for that parameter may not be reliable.

The MINUIT documentation recommends caution in using bounds. Setting bounds can certainly increase the number of function evaluations (and so computation time), and in some cases may cause some instabilities, as the range of acceptable parameter values is not fully explored. On the other hand, preliminary tests suggest that using `max` and `min` to set clearly outlandish bounds does not greatly affect performance or results.

CHAPTER

SIX

USING MATHEMATICAL CONSTRAINTS

While being able to fix variables and place upper and lower bounds on their values are key parts of lmfit, the ability to place mathematical constraints on parameters is also highly desirable. This section describes how to do this, and what sort of parameterizations are possible – see the asteval for further documentation.

6.1 Overview

Just as one can place bounds on a Parameter, or keep it fixed during the fit, so too can one place mathematical constraints on parameters. The way this is done with lmfit is to write a Parameter as a mathematical expression of the other parameters and a set of pre-defined operators and functions. The constraint expressions are simple Python statements, allowing one to place constraints like:

```
pars = Parameters()
pars.add('frac_curve1', value=0.5, min=0, max=1)
pars.add('frac_curve2', expr='1-frac_curve1')
```

as the value of the *frac_curve1* parameter is updated at each step in the fit, the value of *frac_curve2* will be updated so that the two values are constrained to add to 1.0. Of course, such a constraint could be placed in the fitting function, but the use of such constraints allows the end-user to modify the model of a more general-purpose fitting function.

Nearly any valid mathematical expression can be used, and a variety of built-in functions are available for flexible modeling.

6.2 Supported Operators, Functions, and Constants

The mathematical expressions used to define constrained Parameters need to be valid python expressions. As you'd expect, the operators '+', '-', '*', '/', '**', are supported. In fact, a much more complete set can be used, including Python's bit- and logical operators:

```
+, -, *, /, **, &, |, ^, <<, >>, %, and, or,
==, >, >=, <, <=, !=, ~, not, is, is not, in, not in
```

The values for *e* (2.7182818...) and *pi* (3.1415926...) are available, as are several supported mathematical and trigonometric function:

```
abs, acos, acosh, asin, asinh, atan, atan2, atanh, ceil,
copysign, cos, cosh, degrees, exp, fabs, factorial,
floor, fmod, frexp, fsum, hypot, isinf, isnan, ldexp,
log, log10, log1p, max, min, modf, pow, radians, sin,
sinh, sqrt, tan, tanh, trunc
```

In addition, all Parameter names will be available in the mathematical expressions. Thus, with parameters for a few peak-like functions:

```
pars = Parameters()
pars.add('amp_1', value=0.5, min=0, max=1)
pars.add('cen_1', value=2.2)
pars.add('wid_1', value=0.2)
```

The following expression are all valid:

```
pars.add('amp_2', expr='(2.0 - amp_1**2)')
pars.add('cen_2', expr='cen_1 * wid_2 / max(wid_1, 0.001)')
pars.add('wid_2', expr='sqrt(pi)*wid_1')
```

In fact, almost any valid Python expression is allowed. A notable example is that Python's 1-line *if expression* is supported:

```
pars.add('bounded', expr='param_a if test_val/2. > 100 else param_b')
```

which is equivalent to the more familiar:

```
if test_val/2. > 100:
    bounded = param_a
else:
    bounded = param_b
```

6.3 Using Inequality Constraints

A rather common question about how to set up constraints that use an inequality, say, $x + y \leq 10$. This can be done with algebraic constraints by recasting the problem, as $x + y = \delta$ and $\delta \leq 10$. That is, first, allow x to be held by the freely varying parameter *x*. Next, define a parameter *delta* to be variable with a maximum value of 10, and define parameter *y* as *delta* - *x*:

```
pars = Parameters()
pars.add('x',     value = 5, vary=True)
pars.add('delta', value = 5, max=10, vary=True)
pars.add('y',     expr='delta-x')
```

The essential point is that an inequality still implies that a variable (here, *delta*) is needed to describe the constraint. The secondary point is that upper and lower bounds can be used as part of the inequality to make the definitions more convenient.

6.4 Advanced usage of Expressions in lmfit

The expression used in a constraint is converted to a Python Abstract Syntax Tree, which is an intermediate version of the expression – a syntax-checked, partially compiled expression. Among other things, this means that Python's own parser is used to parse and convert the expression into something that can easily be evaluated within Python. It also means that the symbols in the expressions can point to any Python object.

In fact, the use of Python's AST allows a nearly full version of Python to be supported, without using Python's built-in `eval()` function. The asteval module actually supports most Python syntax, including for- and while-loops, conditional expressions, and user-defined functions. There are several unsupported Python constructs, most notably the class statement, so that new classes cannot be created, and the import statement, which helps make the asteval module safe from malicious use.

One important feature of the asteval module is that you can add domain-specific functions into the it, for later use in constraint expressions. To do this, you would use the `asteval` attribute of the `Minimizer` class, which contains a complete AST interpreter. The asteval interpreter uses a flat namespace, implemented as a single dictionary. That means you can preload any Python symbol into the namespace for the constraints:

```
def lorentzian(x, amp, cen, wid):
    "lorentzian function: wid = half-width at half-max"
    return (amp  / (1 + ((x-cen)/wid)**2))

fitter = Minimizer()
fitter.asteval.symtable['lorenztian'] = lorenztian
```

and this `lorenztian()` function can now be used in constraint expressions.

CHAPTER

SEVEN

SIMPLE BUILTIN FITTING MODELS

It is common to want to fit some 1-dimensional data set to a simple peak or line shape, such as Gaussians, Lorentzian, and Voigt peaks, Exponential decays, and so on. These are used in a wide range of spectroscopic techniques as well as in basic mathematical analysis. In an effort to make make simple things truly simple, the lmfit provides a few simple wrappers for doing such fits in its *models1d* module.

7.1 Example

Let's start with a very simple example. We'll read data from a simple datafile, and fit it to a Gaussian peak. A script to do this could be:

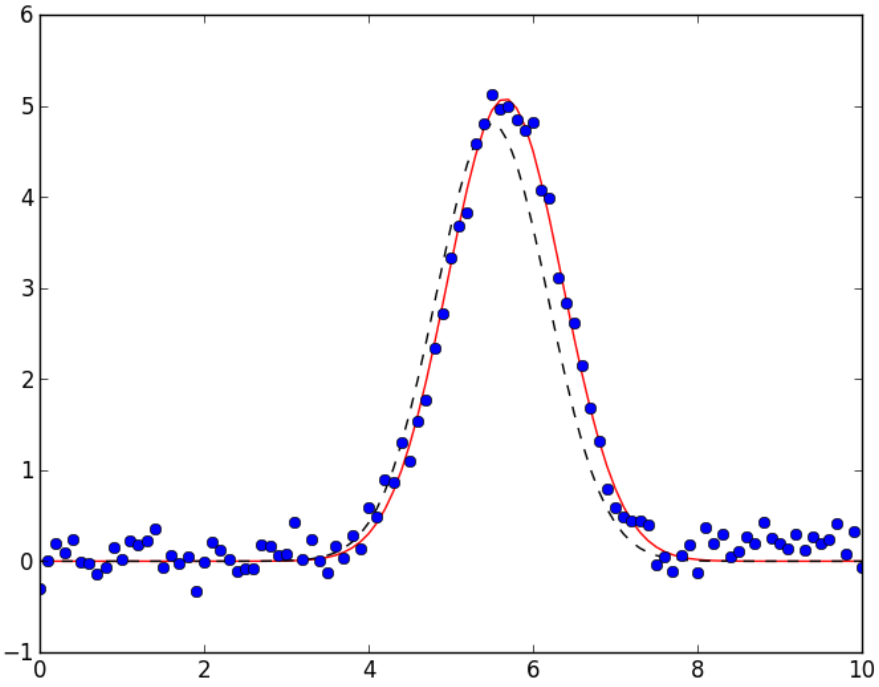
```
import numpy as np
from lmfit.models1d import  GaussianModel
import matplotlib.pyplot as plt

data = np.loadtxt('model1d_gauss.dat')
x = data[:, 0]
y = data[:, 1]

model = GaussianModel()
model.guess_starting_values(y, x=x)
# model.params['amplitude'].value=6.0

init_fit = model.model(x=x)
model.fit(y, x=x)

print model.fit_report(min_correl=0.25)

final_fit = model.model(x=x)

plt.plot(x, final_fit, 'r-')
plt.plot(x, init_fit, 'k--')
plt.plot(x, y,         'bo')
plt.show()
```

First, we read in the data for 'x' and 'y', then build a Gaussian model. This 'model' contains all the Parameters for a Gaussian line shape. We then expliticly tell the model to make initial guesses for the Parameters based on the data arrays, and save the model predicted with these initial Parameter values. We then perform the actual fit, and print out and display the results. The printed output will be (approximately):

```
[[Variables]]
     amplitude:     8.880222 +/- 0.113597 (1.28%) initial =  8.182302
     center:        5.65866 +/- 0.01030533 (0.18%) initial =  5.5
```

```
     fwhm:          1.642853 +/- 0.02426699 (1.48%) == '2.35482*sigma'
     sigma:         0.6976553 +/- 0.01030524 (1.48%) initial =  0.6794575
[[Correlations]] (unreported correlations are <  0.250)
    C(amplitude, sigma)          =  0.577
```

and the resulting plot will look like:



which shows a good fit (the data were simulated).

You can see here that the model created Parameters named 'amplitude', 'center', 'fwhm', and 'sigma' for the Gaussian model.

You can also see from the results that the starting guess were a pretty good estimate for this simple data set. In fact, it's generally possible to not bother running `guess_starting_values()` explicitly. If this method has not been run already, `fit()` will run it for you. Good reasons to run this method yourself are if want to save the initial estimate of the data, or to alter the starting values by hand.

7.2 classes in the `models1d` module

Several fitting models are available

class `GaussianModel`

class `LorentzianModel`

class `VoigtModel`

class `PeakModel`

class `ExponentialModel`

class **ExponentialModel**

class **StepModel**

class **RectangleModel**

C

C