Least-Squares Minimization with Constraints for Python

Release 0.6

Matthew Newville

August 14, 2012

CONTENTS

The lmfit Python package provides a simple, flexible interface to non-linear least-squares optimization, or curve fitting. By default, lmfit uses and builds upon the Levenberg-Marquardt minimization algorithm from MINPACK-1 as implemented in scipy.optimize.leastsq. Provisional support for some other optimization routines is included. Currently, the L-BFGS (limited memory Broyden-Fletcher-Goldfarb-Shanno) algorithm as implemented in scipy.optimize.l_bfgs_b andr the simulated annealing algorithm as implemented in scipy.optimize.anneal are both implemented and partially tested. However, the Levenberg-Marquardt algorithm is by far the most tested and appears to be the most robust for finding local minima of well-described models of scientific measurements, parts of this document may assume that it Levenberg-Marquardt algorithm is being discussed.

For any minimization problem, the programmer must provide an objective function that takes a set of values for the variables in the fit, and produces the residual function to be minimized in the least-squares sense.

The lmfit package allows models to be written in terms of a set of Parameters, which are extensions of simple numerical variables with the following properties:

- Parameters can be fixed or floated in the fit.
- Parameters can be bounded with a minimum and/or maximum value.
- Parameters can be written as simple mathematical expressions of other Parameters, using the asteval module (which is included with lmfit). These values will be re-evaluated at each step in the fit, so that the expression is satisfied. This gives a simple but flexible approach to constraining fit variables.

The main advantage of using Parameters instead of fit variables is that the objective function does not have to be rewritten for a change in what is varied or what constraints are placed on the fit. The programmer can write a fairly general model that encapsulates the phenomenon to be optimized, and then allow a user of the model to change what is varied and what constraints are placed on the model.

For the Levenberg-Marquardt algorithm, lmfit also calculates and reports the estimated uncertainties and correlation between fitted variables.

CHAPTER

ONE

DOWNLOADING AND INSTALLATION

1.1 Prerequisites

The lmfit package requires Python, Numpy, and Scipy. Extensive testing on version compatibility has not yet been done. Initial tests work with Python 3.2, but little testing with Python 3 has yet been done. No testing has been done with 64-bit architectures, but as this package is pure Python, no significant troubles are expected.

1.2 Downloads

The latest stable version is available from PyPI or CARS (Univ of Chicago):

Download Option	Python Versions	Location
Source Kit	2.6, 2.7, 3.2	• lmfit-0.6.tar.gz (PyPI) • lmfit-0.6.tar.gz (CARS)
Win32 Installer	2.6	• lmfit-0.6.win32-py2.6.exe (PyPI) • lmfit-0.6.win32-py2.6.exe (CARS)
Win32 Installer	2.7	• lmfit-0.6.win32-py2.7.exe (PyPI) • lmfit-0.6.win32-py2.7.exe (CARS)
Win32 Installer	3.2	• lmfit-0.6.win32-py3.2.exe (PyPI) • lmfit-0.6.win32-py3.2.exe (CARS)
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/newville/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 implemenation 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, but does assume some basic knowledge of Python, Numpy, and modeling data.

To model data in the least-squares sense, the most important step is writing a function that takes the values of the fitting variables and calculates a residual function (data-model) that is to be minimized in the least-squares sense

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

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 a function that takes the variable values and calculates the residual $y_i^{\rm meas} - y_i^{\rm model}(\mathbf{v})$, perhaps something like:

```
def residual(vars, x, 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)
```

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))
```

Though in python, and fairly easy to use, this 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 fit 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 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):
    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)

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))
```

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 later:

```
params['amp'].vary = True
params['decay'].max = 0.10
```

Now the fit will *not* vary the amplitude parameter, and will also impose a lower bound on the decay factor and an upper bound on the frequency. 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`** (*value=None*[, *vary=True*[, *min=None*[, *max=None*[, *name=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

- **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).
- **name** (`None` or string – will be overwritten during fit if `None`.) – parameter name
- **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

def residual(params, x, data=None):
    amp = params['amp'].value
    shift = params['phase_shift'].value
    omega = params['omega'].value
    decay = params['decay'].value

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

    return (data-model)

params = Parameters()
params.add('amp', value=10)
params.add('decay', value=0.007, vary=False)
params.add('phase_shift', value=0.2)
params.add('omega', value=3.0)

result = minimize(residual, params, args=(x, data))

print result.chisqr
print 'Best-Fit Values:'
for name, par in params.items():
    print '  %s = %.4f +/- %.4f ' % (name, par.value, par.stderr)
```

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*[, *engine='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.
- **engine** (*string*) – name of fitting engine to use. See *Choosing Different Fitting Engines* 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.

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 residual 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.

As the function will be passed in a dictionary of `Parameter` s, 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):
    # 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
    return (model - data)
```

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 Engines

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 `engine` keyword to the `minimize()` function or use the corresponding method name from the `Minimizer` class as listed in the *Table of Supported Fitting Engines*.

Table of Supported Fitting Engines:

Engine	`engine` arg to `minimize()`	`Minimizer` method
Levenberg-Marquardt	`leastsq`	`leastsq()`
L-BFGS-B	`lbfgsb`	`lbfgsb()`
Simulated Annealing	`anneal`	`anneal()`

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* engine, 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* engine has completed succsessfully, 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[, **kws]]]]*)
creates a Minimizer, for fine-grain access to fitting methods and attributes.

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`.
- **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.
- **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.

The Minimizer object has a few public methods:

`leastsq` (***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` argument	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`	1000*(nvar+1)	maximum number of function calls (nvar= # of variables)

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

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

`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.

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 inverals 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 differnce 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 betweeen 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_errors(mi.params)
  a:     0.099713 +/- 0.000193 (inital=  0.100000)
  b:     1.988121 +/- 0.012165 (inital=  1.000000)
Correlations:
    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    0.09894   0.09894   0.09894   0.09971   0.10049   0.10049   0.10049
     99.70%    95.00%    67.40%     0.00%    67.40%    95.00%    99.70%
b    1.95151   1.96413   1.97643   1.98812   1.99981   2.01211   2.02473
```

As we can see, the estimated error is almost the same: it is not necessary to caclulate 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:

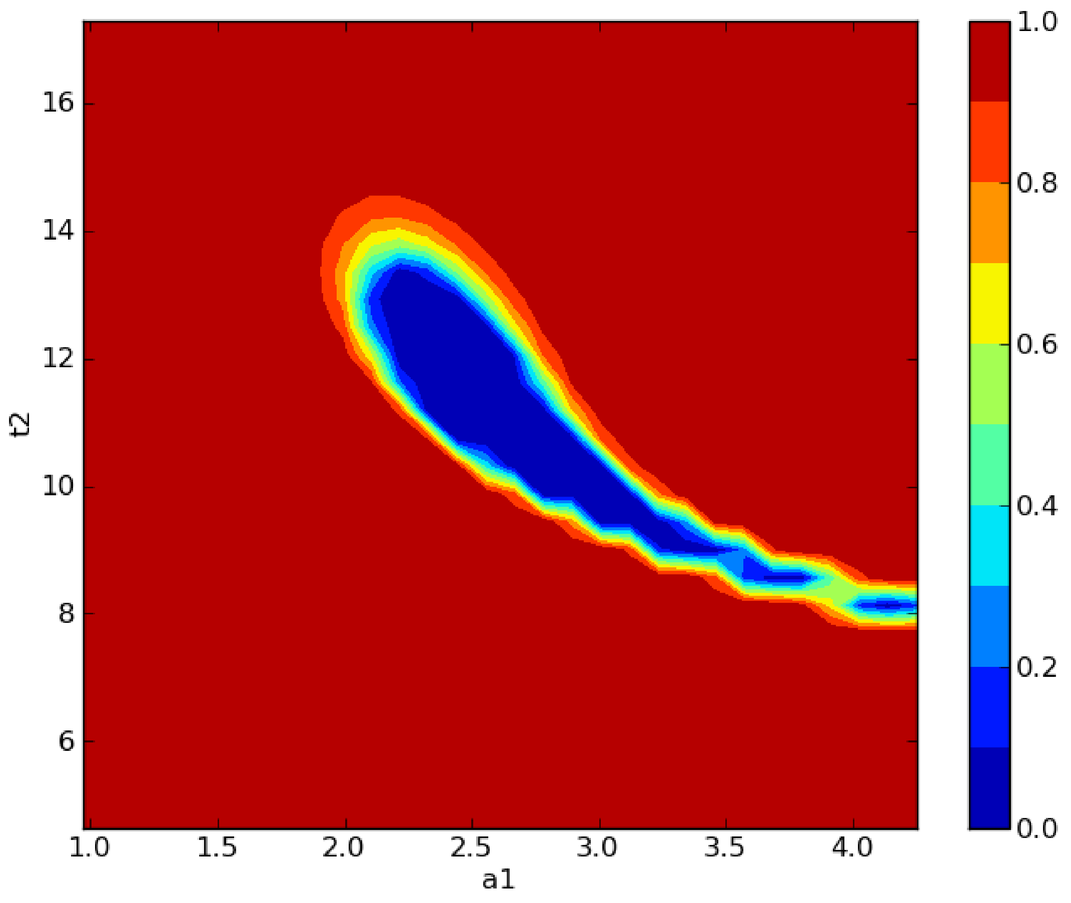
```
In [18]: mi=lmfit.minimize(residual, p)

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

In [20]: lmfit.printfuncs.report_errors(mi.params, show_correl=False)
  a1:     2.611014 +/- 0.327959 (inital=  5.000000)
  a2:    -4.512928 +/- 0.399194 (inital= -5.000000)
  t1:     1.569477 +/- 0.334505 (inital=  2.000000)
  t2:     10.961366 +/- 1.263868 (inital=  5.000000)
```

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=0)

In [22]: lmfit.printfuncs.report_ci(ci)
      95.00%    68.00%     0.00%    68.00%    95.00%
a1   2.11696   2.33696   2.61101   3.06631   4.28728
      95.00%    68.00%     0.00%    68.00%    95.00%
a2  -6.39492  -5.05982  -4.51293  -4.19528  -3.97850
      95.00%    68.00%     0.00%    68.00%    95.00%
t2   8.00414   9.62688  10.96137  12.17947  13.34824
      95.00%    68.00%     0.00%    68.00%    95.00%
t1   1.07036   1.28482   1.56948   1.97534   2.64341
```

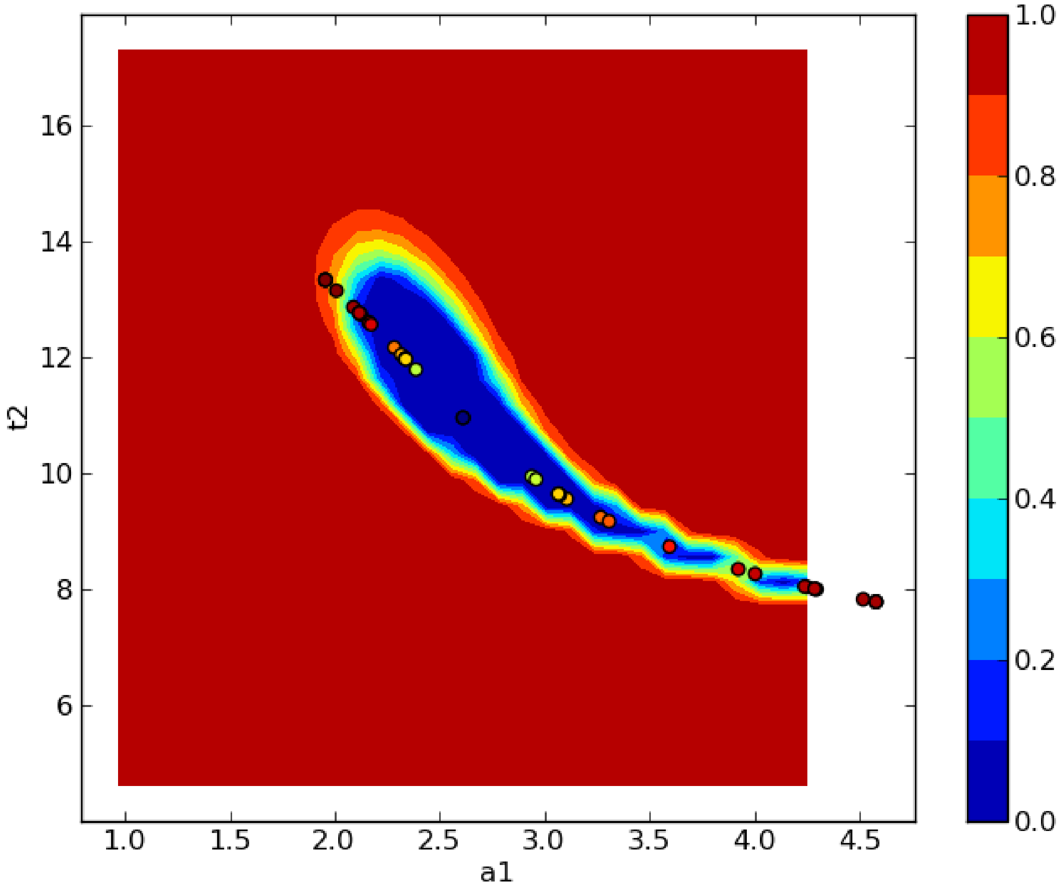
If you compare the calculated error estimates, you will see that the regular estimate is too small. Now let’s plot a confidence region:

```
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 0xa888d6c>

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 0xab7cb6c>

In [36]: plt.scatter(x2,y2,c=prob2,s=30)
Out[36]: <matplotlib.collections.PathCollection at 0xab933ec>
```



4.4 Documentation of methods

conf_interval(*minimizer*, *p_names=None*, *sigmas=(0.67400000000000004*, *0.94999999999999996*, *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 reoptimized 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 probality from the opimized chi-square. Default (`None`) uses built-in f_compare (F test).

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 quantils 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 stderrs 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 probality from the opimized chi-square. Default (`None`) uses built-in f_compare (F test).

Examples

```
>>> from lmfit.printfuncs import *
>>> 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

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

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

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

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

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

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 jacobian 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, prelminary 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 Advanced usage of Expressions in lmfit

The expression 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 lorenztian(x, amp, cen, wid):
    "lorenztian 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.

PYTHON MODULE INDEX

PYTHON MODULE INDEX

INDEX