Non-Linear Least-Squares Minimization and Curve-Fitting for Python

Release 0.8.0rc4

Matthew Newville, Till Stensitzki, and others

September 18, 2014

The lmfit python package provides a simple and flexible interface to non-linear optimization and curve fitting problems. Initially designed to extend the the Levenberg-Marquardt algorithm in `scipy.optimize.leastsq()`, lmfit supports most of the optimization methods from `scipy.optimize`. It also provides a simple way to apply this extension to *curve fitting* or *data modeling* problems.

The key concept in lmfit is the `Parameter` – the quantity to be optimized in all minimization problems in place of a plain floating point number. A `Parameter` has a value that can be varied in the fit, fixed, have upper and/or lower bounds. It can even have a value that is constrained by an algebraic expression of other Parameter values. Since `Parameters` live outside the core optimization routines, they can be used in **all** optimization routines from `scipy.optimize`. By using `Parameter` objects instead of plain variables, the objective function does not have to be modified to reflect every change of what is varied in the fit. This simplifies the writing of models, allowing general models that describe the phenomenon to be written, and gives the user more flexibility in using and testing variations of that model.

Lmfit supports several optimization methods from `scipy.optimize`. The default and best tested optimization method (and the origin of the name) is the Levenberg-Marquardt algorithm of `scipy.optimize.leastsq()`. An important feature of this method is that it automatically estimates uncertainties and correlations between fitted variables from the covariance matrix used in the fit. But, because this approach is sometimes questioned (and rightly so), lmfit also supports methods to do a brute force determination of the confidence intervals for a set of parameters.

Lmfit provides high-level curve-fitting or data modeling functionality through its `Model` class, which extends the capabilities of `scipy.optimize.curve_fit()`. This allows you to turn a function that models for your data into a python class that helps you parametrize and fit data with that model. Many pre-built models for common lineshapes are included and ready to use.

The lmfit package is an open-source project, and the software and this document are works in progress. If you are interested in participating in this effort please use the lmfit github repository.

CHAPTER

ONE

GETTING STARTED WITH NON-LINEAR LEAST-SQUARES FITTING

The lmfit package is designed to provide simple tools to help you build of complex fitting models for non-linear least-squares problems and apply these models to real data. This section gives an overview of the concepts and describes how to set up and perform simple fits. Some basic knowledge of Python, numpy, and modeling data are assumed.

To do a non-linear least-squares fit of a model to data or for a variety 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:

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

where $y_i^{\rm meas}$ is the set of measured data, $y_i^{\rm model}({\bf v})$ is the model calculation, ${\bf v}$ 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 $y_i^{\rm meas} - y_i^{\rm model}({\bf v})$, or the residual scaled by the data uncertainties, $[y_i^{\rm meas} - y_i^{\rm model}({\bf v})]/\epsilon_i$, 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.optimize`, 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. There are several practical challenges to using this approach, including:

1. The user has to keep track of the order of the variables, and their meaning – vars[0] is the amplitude, vars[2] is the frequency, and so on, although there is no intrinsic meaning to this order.

2. If the user wants to fix a particular variable (*not* vary it in the fit), the residual function has to be altered to have fewer variables, and have the corresponding constant value passed in some other way. While reasonable for simple cases, this quickly becomes a significant work for more complex models, and greatly complicates modeling for people not intimately familiar with the details of the fitting code.
3. There is no simple, robust way to put bounds on values for the variables, or enforce mathematical relationships between the variables. In fact, those optimization methods that do provide bounds, require bounds to be set for all variables with separate arrays that are in the same arbitrary order as variable values. Again, this is acceptable for small or one-off cases, but becomes painful if the fitting model needs to change.

These shortcomings are really do solely to the use of traditional arrays of variables, as matches closely the implementation of the Fortran code. The lmfit module overcomes these shortcomings by using a core reason for using Python – objects. The key concept for lmfit is to use `Parameter` objects instead of plain floating point numbers as the variables for the fit. By using `Parameter` objects (or the closely related `Parameters` – a dictionary of `Parameter` objects), one can

1. not care about the order of variables, but refer to Parameters by meaningful names.
2. place bounds on Parameters as attributes, without worrying about order.
3. fix Parameters, without having to rewrite the objective function.
4. place algebraic constraints on Parameters.

To illustrate the value of this approach, we can rewrite the above example as:

```
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))
```

At first look, we simply replaced a list of values with a dictionary, accessed by name – not a huge improvement. 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. 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)
```

(where `vary=False` will prevent the value from changing in the fit, and `min=-0.0` will set a lower bound on that parameters value) or after being defined by setting the corresponding attributes after they have been created:

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

Importantly, our function to be minimized remains unchanged.

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 objective function, but the approach here allows some external control, that is by the **user** of the objective function instead of just by the author of the objective function.

Finally, in addition to the `Parameters` approach to fitting data, lmfit allows you to easily switch optimization methods without rewriting your objective function, and provides tools for writing fitting reports and for better determining the confidence levels for Parameters.

CHAPTER

TWO

DOWNLOADING AND INSTALLATION

2.1 Prerequisites

The lmfit package requires Python, Numpy, and Scipy. Scipy version 0.13 or higher is recommended, but extensive testing on compatibility with various versions of scipy has not been done. Lmfit does work with Python 2.7, and 3.2 and 3.3. No testing has been done with Python 3.4, but as the package is pure Python, relying only on scipy and numpy, no significant troubles are expected. The nose frameworkt is required for running the test suite, and IPython and matplotib are recommended. If Pandas is available, it will be used in portions of lmfit.

2.2 Downloads

The latest stable version of lmfit is available from PyPi.

2.3 Installation

If you have pip installed, you can install lmfit with:

```
pip install lmfit
```

or, if you have Python Setup Tools installed, you install lmfit with:

```
easy_install -U lmfit
```

or, you can download the source kit, unpack it and install with:

```
python setup.py install
```

2.4 Development Version

To get the latest development version, use:

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

and install using:

```
python setup.py install
```

2.5 Testing

A battery of tests scripts that can be run with the nose testing framework is distributed with lmfit in the `tests` folder. These are routinely run on the development version. Running `nosetests` should run all of these tests to completion without errors or failures.

Many of the examples in this documentation are distributed with lmfit in the `examples` folder, and sould also run for you. Many of these require

2.6 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. Much of the work on improved unit testing and high-level model functions was done by Daniel B. Allen, with substantial input from Antonino Ingargiola. Many valuable suggestions for improvements have come from Christoph Deil. The implementation of parameter bounds as described in the MINUIT documentation is taken from Jonathan J. Helmus' leastsqbound code, with permission. The code for propagation of uncertainties is taken from Eric O. Le Bigot's uncertainties package, with permission. 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.

2.7 License

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

CHAPTER

THREE

PARAMETER AND PARAMETERS

This chapter describes `Parameter` objects which are fundamental to the lmfit approach to optimization. A `Parameter` contains the value adjusted in the optimization as well as several other properties that control what that value is. Most real use cases will use more than 1 `Parameter`, and so use the `Parameters` class, which provides an ordered dictionary of `Parameter` objects.

3.1 The `Parameter` class

class `Parameter` (*name=None*[, *value=None*[, *vary=True*[, *min=None*[, *max=None*[, *expr=None*]]]]])
create a Parameter object.

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`) [default `True`]) – 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.

After a fit, a Parameter for a fitted variable (that is with vary = `True`) will have the `value` attribute holding the best-fit value. Depending on the success of the fit and fitting algorithm used, it may also have attributes `stderr` and `correl`.

`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}
```

See *Bounds Implementation* for details on the math used to implement the bounds with `min` and `max`.

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.

`set` (*value=None*[, *vary=None*[, *min=None*[, *max=None*[, *expr=None*]]]])
set or update a Parameters value or other attributes.

Parameters

- **name** – parameter name
- **value** – the numerical value for the parameter
- **vary** – whether to vary the parameter or not.
- **min** – lower bound for value
- **max** – upper bound for value
- **expr** – mathematical expression to use to evaluate value during fit.

Each argument of `set()` has a default value of `None`, and will be set only if the provided value is not `None`. You can use this to update some Parameter attribute without affecting others, for example:

```
p1 = Parameter('a', value=2.0)
p2 = Parameter('b', value=0.0)
p1.set(min=0)
p2.set(vary=False)
```

to set a lower bound, or to set a Parameter as have a fixed value.

Note that to use this approach to lift a lower or upper bound, doing:

```
p1.set(min=0)
.....
# now lift the lower bound
p1.set(min=None)   # won't work!  lower bound NOT changed
```

won't work – this will not change the current lower bound. Instead you'll have to use `np.inf` to remove a lower or upper bound:

```
# now lift the lower bound
p1.set(min=-np.inf)   # will work!
```

Similarly, to clear an expression of a parameter, you need to pass an empty string, not `None`. You also need to give a value and explicitly tell it to vary:

```
p3 = Parameter('c', expr='(a+b)/2')
p3.set(expr=None)     # won't work!  expression NOT changed

# remove constraint expression
p3.set(value=1.0, vary=True, expr='')  # will work!  parameter now unconstrained
```

3.2 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 are for provided for convenient initialization of a `Parameters`, and one for extracting `Parameter` values into a plain dictionary.

add(*name*[, *value=None*[, *vary=True*[, *min=None*[, *max=None*[, *expr=None*]]]]])

add a named parameter. This 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
```

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'))
```

valuesdict (*self*)

return an ordered dictionary of name:value pairs containing the `name` and `value` of a Parameter.

This is distinct from the `Parameters` itself, as the dictionary values are not `Parameeter` objects, just the `value`. This can be a very convenient way to get updated values in a objective function.

3.3 Simple Example

Using `Parameters`` and `minimize()` function (discussed in the next chapter) might look like this:

```
#!/usr/bin/env python
#<examples/doc_basic.py>
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

#<end of examples/doc_basic.py>
```

Here, the objective function explicitly unpacks each Parameter value. This can be simplified using the `Parameters valuesdict()` method, which would make the objective function `fcn2min` above look like:

```
def fcn2min(params, x, data):
    """ model decaying sine wave, subtract data"""
    v = params.valuesdict()

    model = v['amp'] * np.sin(x * v['omega'] + v['shift']) * np.exp(-x*x*v['decay'])
    return model - data
```

The results are identical, and the difference is a stylisic choice.

CHAPTER

FOUR

PERFORMING FITS, ANALYZING OUTPUTS

As shown in the previous chapter, 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.

4.1 The `minimize()` function

The minimize function takes a objective function (the function that calculates the array to be minimized), a `Parameters` ordered dictionary, 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'*[, *scale_covar=True*[, *iter_cb=None*[, ***leastsq_kws*]]]]]])

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

Parameters

- **function** (*callable.*) – function to return fit residual. See *Writing a Fitting Function* for details.
- **params** (dict or `Parameters`.) – a `Parameters` dictionary. 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 (default `leastsq`)) – name of fitting method to use. See *Choosing Different Fitting Methods* for details
- **scale_covar** (bool (default `True`)) – whether to automatically scale covariance matrix (`leastsq` only)
- **iter_cb** (callable or `None`) – function to be called at each fit iteration
- **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.

If provided, the `iter_cb` function should take arguments of `params, iter, resid, *args, **kws`, where `params` will have the current parameter values, `iter` the iteration, `resid` the current residual array, and `*args` and `**kws` as passed to the objective function.

4.2 Writing a Fitting Function

An important component of a fit is writing a function to be minimized – the *objective function*. 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 way to do this is with `Parameters.valuesdict()`, as with:

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

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

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

    model = parvals['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 value of the 'period' Parameter. It might be wise to ensure 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 putting this directly in the function with:

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

is also a reasonable approach. Similarly, one could place bounds on the `decay` parameter to take values only between `-pi/2` and `pi/2`.

4.3 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 by providing 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`
Powell	`powell`		`Powell`
Conjugate Gradient	`cg`		`CG`
Newton-CG	`newton`		`Newton-CG`
COBYLA	`cobyla`		`COBYLA`
COBYLA	`cobyla`		`COBYLA`
Truncated Newton	`tnc`		`TNC`
Trust Newton-CGn	`trust-ncg`		`trust-ncg`
Dogleg	`dogleg`		`dogleg`
Sequential Linear Squares Programming	`slsqp`		`SLSQP`

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: Much of this documentation assumes that the Levenberg-Marquardt method is the method used. 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 this method.

4.4 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: $\chi^2_{\nu} = \chi^2/(N - N_{\rm varys})$
var_map	list of variable parameter names for rows/columns of covar
covar	covariance matrix (with rows/columns using var_map

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`.

4.5 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 automatically 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

`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.

4.6 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, or Minimizer object as returned by `minimize()`.
- **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]

If the first argument is a Minimizer object, as returned from `minimize()`, the report will include some goodness-of-fit statistics.

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

An example fit with report would be

```
#!/usr/bin/env python
#<examples/doc_withreport.py>

from __future__ import print_function
from lmfit import Parameters, minimize, fit_report
from numpy import random, linspace, pi, exp, sin, sign


p_true = Parameters()
p_true.add('amp', value=14.0)
p_true.add('period', value=5.46)
p_true.add('shift', value=0.123)
p_true.add('decay', value=0.032)

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

    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 = 1001
xmin = 0.
```

```
xmax = 250.0

random.seed(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)
print(fit_report(fit_params))

#<end of examples/doc_withreport.py>
```

which would write out:

```
[[Variables]]
    amp:       13.9121944 +/- 0.141202 (1.01%) (init= 13)
    decay:     0.03264538 +/- 0.000380 (1.16%) (init= 0.02)
    period:    5.48507044 +/- 0.026664 (0.49%) (init= 2)
    shift:     0.16203677 +/- 0.014056 (8.67%) (init= 0)
[[Correlations]] (unreported correlations are <  0.100)
    C(period, shift)             =  0.797
    C(amp, decay)                =  0.582
```

CHAPTER

FIVE

MODELING DATA AND CURVE FITTING

A common use of least-squares minimization is *curve fitting*, where one has a parametrized model function meant to explain some phenomena and wants to adjust the numerical values for the model to most closely match some data. With `scipy`, such problems are commonly solved with `scipy.optimize.curve_fit()`, which is a wrapper around `scipy.optimize.leastsq()`. Since Lmit's `minimize()` is also a high-level wrapper around `scipy.optimize.leastsq()` it can be used for curve-fitting problems, but requires more effort than using `scipy.optimize.curve_fit()`.

Here we discuss lmfit's `Model` class. This takes a model function – a function that calculates a model for some data – and provides methods to create parameters for that model and to fit data using that model function. This is closer in spirit to `scipy.optimize.curve_fit()`, but with the advantages of using `Parameters` and lmfit.

In addition to allowing you turn any model function into a curve-fitting method, Lmfit also provides canonical definitions for many known lineshapes such as Gaussian or Lorentzian peaks and Exponential decays that are widely used in many scientific domains. These are available in the `models` module that will be discussed in more detail in the next chapter (*Built-in Fitting Models in the models module*). We mention it here as you may want to consult that list before writing your own model. For now, we focus on turning python function into high-level fitting models with the `Model` class, and using these to fit data.

5.1 Example: Fit data to Gaussian profile

Let's start with a simple and common example of fitting data to a Gaussian peak. As we will see, there is a buit-in `GaussianModel` class that provides a model function for a Gaussian profile, but here we'll build our own. We start with a simple definition the model function:

```
>>> from numpy import sqrt, pi, exp, linspace
>>>
>>> def gaussian(x, amp, cen, wid):
...     return amp * exp(-(x-cen)**2 /wid)
...
```

that we want to use to fit to some data $y(x)$ represented by the arrays `y` and `x`. Using `scipy.optimize.curve_fit()` makes this easy to do, allowing us to do something like:

```
>>> from scipy.optimize import curve_fit
>>>
>>> x, y = read_data_from_somewhere(....)
>>>
>>> init_vals = [5, 5, 1]     # for [amp, cen, wid]
>>> best_vals, covar = curve_fit(gaussian, x, y, p0=init_vals)
>>> print best_vals
```

That is, we read in data from somewhere, make an initial guess of the model values, and run `scipy.optimize.curve_fit()` with the model function, data arrays, and initial guesses. The results returned are the optimal values for the parameters and the covariance matrix. It's simple and very useful. But it misses the benefits of lmfit.

To solve this with lmfit we would have to write an objective function. But such a function would be fairly simple (essentially, `data - model`, possibly with some weighting), and we would need to define and use appropriately named parameters. Though convenient, it is somewhat of a burden to keep the named parameter straight (on the other hand, with func:*scipy.optimize.curve_fit* you are required to remember the parameter order). After doing this a few times it appears as a recurring pattern, and we can imagine automating this process. That's where the `Model` class comes in.

The `Model` allows us to easily wrap a model function such as the `gaussian` function. This automatically generate the appropriate residual function, and determines the corresponding parameter names from the function signature itself:

```
>>> from lmfit import Model
>>> gmod = Model(gaussian)
>>> gmod.param_names
set(['amp', 'wid', 'cen'])
>>> gmod.independent_vars)
['x']
```

The Model `gmod` knows the names of the parameters and the independent variables. By default, the first argument of the function is taken as the independent variable, held in `independent_vars`, and the rest of the functions positional arguments (and, in certain cases, keyword arguments – see below) are used for Parameter names. Thus, for the `gaussian` function above, the parameters are named `amp`, `cen`, and `wid`, and `x` is the independent variable – all taken directly from the signature of the model function. As we will see below, you can specify what the independent variable is, and you can add or alter parameters too.

On creation of the model, parameters are *not* created. The model knows what the parameters should be named, but not anything about the scale and range of your data. You will normally have to make these parameters and assign initiald values and other attributes. To help you do this, each model has a `make_params()` method that will generate parameters with the expected names:

```
>>> params = gmod.make_params()
```

This creates the `Parameters` but doesn't necessarily give them initial values – again, the model has no idea what the scale should be. You can set initial values for parameters with keyword arguments to `make_params()`, as with:

```
>>> params = gmod.make_params(cen=5, amp=200, wid=1)
```

or assign them (and other parameter properties) after the `Parameters` has been created.

A `Model` has several methods associated with it. For example, one can use the `eval()` method to evaluate the model or the `fit()` method to fit data to this model with a `Parameter` object. Both of these methods can take explicit keyword arguments for the parameter values. For example, one could use `eval()` to calculate the predicted function:

```
>>> x = linspace(0, 10, 201)
>>> y = gmod.eval(x=x, amp=10, cen=6.2, wid=0.75)
```

Admittedly, this a slightly long-winded way to calculate a Gaussian function. But now that the model is set up, we can also use its `fit()` method to fit this model to data, as with:

```
result = gmod.fit(y, x=x, amp=5, cen=5, wid=1)
```

Putting everything together, the script to do such a fit (included in the `examples` folder with the source code) is:

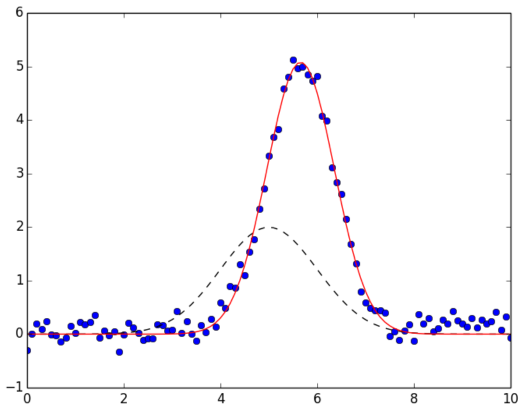
```
#!/usr/bin/env python
#<examples/doc_model1.py>
from numpy import sqrt, pi, exp, linspace, loadtxt
from lmfit import  Model
```

```
import matplotlib.pyplot as plt

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

def gaussian(x, amp, cen, wid):
    "1-d gaussian: gaussian(x, amp, cen, wid)"
    return (amp/(sqrt(2*pi)*wid)) * exp(-(x-cen)**2 /(2*wid**2))

gmod = Model(gaussian)
result = gmod.fit(y, x=x, amp=5, cen=5, wid=1)

print(result.fit_report())

plt.plot(x, y,         'bo')
plt.plot(x, result.init_fit, 'k--')
plt.plot(x, result.best_fit, 'r-')
plt.show()
#<end examples/doc_model1.py>
```

which is pretty compact and to the point. The returned `result` will be a `ModelFit` object. As we will see below, this has many components, including a `fit_report()` method, which will show:

```
[[Model]]
    gaussian
[[Fit Statistics]]
    # function evals   = 33
    # data points      = 101
    # variables        = 3
    chi-square         = 3.409
    reduced chi-square = 0.035
[[Variables]]
    amp:   8.88021829 +/- 0.113594 (1.28%) (init= 5)
    cen:   5.65866102 +/- 0.010304 (0.18%) (init= 5)
    wid:   0.69765468 +/- 0.010304 (1.48%) (init= 1)
[[Correlations]] (unreported correlations are <  0.100)
    C(amp, wid)                  =  0.577
```

The result will also have `init_fit` for the fit with the initial parameter values and a `best_fit` for the fit with the best fit parameter values. These can be used to generate the following plot:



which shows the data in blue dots, the best fit as a solid red line, and the initial fit as a dashed black line.

We emphasize here that the fit to this model function was really performed with 2 lines of code:

```
gmod = Model(gaussian)
result = gmod.fit(y, x=x, amp=5, cen=5, wid=1)
```

These lines clearly express that we want to turn the `gaussian` function into a fitting model, and then fit the $y(x)$ data to this model, starting with values of 5 for `amp`, 5 for `cen` and 1 for `wid`, and compare well to `scipy.optimize.curve_fit()`:

```
best_vals, covar = curve_fit(gaussian, x, y, p0=[5, 5, 1])
```

except that all the other features of lmfit are included such as that the `Parameters` can have bounds and constraints and the result is a richer object that can be reused to explore the fit in more detail.

5.2 The `Model` class

The `Model` class provides a general way to wrap a pre-defined function as a fitting model.

class **`Model`** (*func*[, *independent_vars=None*[, *param_names=None*[, *missing=None*[, *prefix=''*[, *name=None*[, ***kws*]]]]]])

Create a model based on the user-supplied function. This uses introspection to automatically converting argument names of the function to Parameter names.

Parameters

- **func** (*callable*) – model function to be wrapped
- **independent_vars** (`None` (default) or list of strings.) – list of argument names to `func` that are independent variables.
- **param_names** (`None` (default) or list of strings) – list of argument names to `func` that should be made into Parameters.
- **missing** (one of `None` (default), 'none', 'drop', or 'raise'.) – how to handle missing values.
- **prefix** (*string*) – prefix to add to all parameter names to distinguish components.
- **name** (`None` or string.) – name for the model. When `None` (default) the name is the same as the model function (`func`).
- **kws** – addtional keyword arguments to pass to model function.

Of course, the model function will have to return an array that will be the same size as the data being modeled. Generally this is handled by also specifying one or more independent variables.

5.2.1 `Model` class Methods

`eval` (*params=None*[, ***kws*])

evaluate the model function for a set of parameters and inputs.

Parameters

- **params** (`None` (default) or Parameters) – parameters to use for fit.
- **kws** – addtional keyword arguments to pass to model function.

Returns ndarray for model given the parameters and other arguments.

If `params` is `None`, the values for all parameters are expected to be provided as keyword arguments. If `params` is given, and a keyword argument for a parameter value is also given, the keyword argument will be used.

Note that all non-parameter arguments for the model function – **including all the independent variables!** – will need to be passed in using keyword arguments.

`fit`(*data*[, *params=None*[, *weights=None*[, *method='leastsq'*[, *scale_covar=True*[, *iter_cb=None*[, ***kws*]]]]]])

perform a fit of the model to the `data` array with a set of parameters.

Parameters

- **data** (*ndarray-like*) – array of data to be fitted.
- **params** (`None` (default) or Parameters) – parameters to use for fit.
- **weights** (`None` (default) or ndarray-like.) – weights to use fit.
- **method** (string (default `leastsq`)) – name of fitting method to use. See *Choosing Different Fitting Methods* for details
- **scale_covar** (bool (default `True`)) – whether to automatically scale covariance matrix (`leastsq` only)
- **iter_cb** (callable or `None`) – function to be called at each fit iteration
- **verbose** (bool (default `True`)) – print a message when a new parameter is created due to a *hint*
- **kws** – addtional keyword arguments to pass to model function.

Returns `ModeFitResult` object.

If `params` is `None`, the internal `params` will be used. If it is supplied, these will replace the internal ones. If supplied, `weights` must is an ndarray-like object of same size and shape as `data`.

Note that other arguments for the model function (including all the independent variables!) will need to be passed in using keyword arguments.

`guess`(*data*, ***kws*)

Guess starting values for model parameters.

param data data array used to guess parameter values

type func ndarray

param kws addtional options to pass to model function.

return `Parameters` with guessed initial values for each parameter.

by default this is left to raise a `NotImplementedError`, but may be overwritten by subclasses. Generally, this method should take some values for `data` and use it to construct reasonable starting values for the parameters.

`make_params`(***kws*)

Create a set of parameters for model.

param kws optional keyword/value pairs to set initial values for parameters.

return `Parameters`.

The parameters may or may not have decent initial values for each parameter.

`set_param_hint`(*name*, *value=None*[, *min=None*[, *max=None*[, *vary=True*[, *expr=None*]]]])

set *hints* to use when creating parameters with `make_param()` for the named parameter. This is especially convenient for setting initial values. The `name` can include the models `prefix` or not.

Parameters

- **name** (*string*) – parameter name.
- **value** (*float*) – value for parameter
- **min** (`None` or float) – lower bound for parameter value
- **max** (`None` or float) – upper bound for parameter value
- **vary** (*boolean*) – whether to vary parameter in fit.
- **expr** (*string*) – mathematical expression for constraint

See *Using parameter hints*.

5.2.2 `Model` class Attributes

`components`
a list of instances of `Model` that make up a *composite model*. See *Creating composite models*. Normally, you will not need to use this, but is used by `Model` itself when constructing a composite model from two or more models.

`func`
The model function used to calculate the model.

`independent_vars`
list of strings for names of the independent variables.

`is_composite`
Boolean value for whether model is a composite model.

`missing`
describes what to do for missing values. The choices are

- `None`: Do not check for null or missing values (default)
- `'none'`: Do not check for null or missing values.
- **`'drop'`: Drop null or missing observations in data. If pandas is** installed, `pandas.isnull` is used, otherwise `numpy.isnan` is used.
- **`'raise'`: Raise a (more helpful) exception when data contains null** or missing values.

`name`
name of the model, used only in the string representation of the model. By default this will be taken from the model function.

`opts`
extra keyword arguments to pass to model function. Normally this will be determined internally and should not be changed.

`param_hints`
Dictionary of parameter hints. See *Using parameter hints*.

`param_names`
list of strings of parameter names.

`prefix`
prefix used for name-mangling of parameter names. The default is ‘’. If a particular `Model` has arguments `amplitude`, `center`, and `sigma`, these would become the parameter names. Using a prefix of `g1_` would convert these parameter names to `g1_amplitude`, `g1_center`, and `g1_sigma`. This can be essential to avoid name collision in composite models.

5.2.3 Determining parameter names and independent variables for a function

The `Model` created from the supplied function `func` will create a `Parameters` object, and names are inferred from the function arguments, and a residual function is automatically constructed.

By default, the independent variable is take as the first argument to the function. You can explicitly set this, of course, and will need to if the independent variable is not first in the list, or if there are actually more than one independent variables.

If not specified, Parameters are constructed from all positional arguments and all keyword arguments that have a default value that is numerical, except the independent variable, of course. Importantly, the Parameters can be modified after creation. In fact, you'll have to do this because none of the parameters have valid initial values. You can place bounds and constraints on Parameters, or fix their values.

5.2.4 Explicitly specifying `independent_vars`

As we saw for the Gaussian example above, creating a `Model` from a function is fairly easy. Let's try another:

```
>>> def decay(t, tau, N):
...    return N*np.exp(-t/tau)
...
>>> decay_model = Model(decay)
>>> print decay_model.independent_vars
['t']
>>> for pname, par in decay_model.params.items():
...     print pname, par
...
tau <Parameter 'tau', None, bounds=[None:None]>
N <Parameter 'N', None, bounds=[None:None]>
```

Here, `t` is assumed to be the independent variable because it is the first argument to the function. The other function arguments are used to create parameters for the model.

If you want `tau` to be the independent variable in the above example, you can say so:

```
>>> decay_model = Model(decay, independent_vars=['tau'])
>>> print decay_model.independent_vars
['tau']
>>> for pname, par in decay_model.params.items():
...     print pname, par
...
t <Parameter 't', None, bounds=[None:None]>
N <Parameter 'N', None, bounds=[None:None]>
```

You can also supply multiple values for multi-dimensional functions with multiple independent variables. In fact, the meaning of *independent variable* here is simple, and based on how it treats arguments of the function you are modeling:

independent variable a function argument that is not a parameter or otherwise part of the model, and that will be required to be explicitly provided as a keyword argument for each fit with `fit()` or evaluation with `eval()`.

Note that independent variables are not required to be arrays, or even floating point numbers.

5.2.5 Functions with keyword arguments

If the model function had keyword parameters, these would be turned into Parameters if the supplied default value was a valid number (but not `None`, `True`, or `False`).

```
>>> def decay2(t, tau, N=10, check_positive=False):
...     if check_small:
...         arg = abs(t)/max(1.e-9, abs(tau))
...     else:
...         arg = t/tau
...     return N*np.exp(arg)
...
>>> mod = Model(decay2)
>>> for pname, par in mod.params.items():
...      print pname, par
...
t <Parameter 't', None, bounds=[None:None]>
N <Parameter 'N', 10, bounds=[None:None]>
```

Here, even though `N` is a keyword argument to the function, it is turned into a parameter, with the default numerical value as its initial value. By default, it is permitted to be varied in the fit – the 10 is taken as an initial value, not a fixed value. On the other hand, the `check_positive` keyword argument, was not converted to a parameter because it has a boolean default value. In some sense, `check_positive` becomes like an independent variable to the model. However, because it has a default value it is not required to be given for each model evaluation or fit, as independent variables are.

5.2.6 Defining a `prefix` for the Parameters

As we will see in the next chapter when combining models, it is sometimes necessary to decorate the parameter names in the model, but still have them be correctly used in the underlying model function. This would be necessary, for example, if two parameters in a composite model (see *Creating composite models* or examples in the next chapter) would have the same name. To avoid this, we can add a `prefix` to the `Model` which will automatically do this mapping for us.

```
>>> def myfunc(x, amplitude=1, center=0, sigma=1):
...

>>> mod = Model(myfunc, prefix='f1_')
>>> for pname, par in mod.params.items():
...      print pname, par
...
f1_amplitude <Parameter 'f1_amplitude', None, bounds=[None:None]>
f1_center <Parameter 'f1_center', None, bounds=[None:None]>
f1_sigma <Parameter 'f1_sigma', None, bounds=[None:None]>
```

You would refer to these parameters as `f1_amplitude` and so forth, and the model will know to map these to the `amplitude` argument of `myfunc`.

5.2.7 Initializing model parameters

As mentioned above, the parameters created by `Model.make_params()` are generally created with invalid initial values of `None`. These values **must** be initialized in order for the model to be evaluated or used in a fit. There are four different ways to do this initialization that can be used in any combination:

1. You can supply initial values in the definition of the model function.
2. You can initialize the parameters when creating parameters with `make_params()`.
3. You can give parameter hints with `set_param_hint()`.
4. You can supply initial values for the parameters when you use the `eval()` or `fit()` methods.

Of course these methods can be mixed, allowing you to overwrite initial values at any point in the process of defining and using the model.

Initializing values in the function definition

To supply initial values for parameters in the definition of the model function, you can simply supply a default value:

```
>>> def myfunc(x, a=1, b=0):
>>>     ...
```

instead of using:

```
>>> def myfunc(x, a, b):
>>>     ...
```

This has the advantage of working at the function level – all parameters with keywords can be treated as options. It also means that some default initial value will always be available for the parameter.

Initializing values with `make_params()`

When creating parameters with `make_params()` you can specify initial values. To do this, use keyword arguments for the parameter names and initial values:

```
>>> mod = Model(myfunc)
>>> pars = mod.make_params(a=3, b=0.5)
```

Initializing values by setting parameter hints

After a model has been created, but prior to creating parameters with `make_params()`, you can set parameter hints. These allows you to set not only a default initial value but also to set other parameter attributes controlling bounds, whether it is varied in the fit, or a constraint expression. To set a parameter hint, you can use `set_param_hint()`, as with:

```
>>> mod = Model(myfunc)
>>> mod.set_param_hint('a', value = 1.0)
>>> mod.set_param_hint('b', value = 0.3, min=0, max=1.0)
>>> pars = mod.make_params()
```

Parameter hints are discussed in more detail in section *Using parameter hints*.

Initializing values when using a model

Finally, you can explicitly supply initial values when using a model. That is, as with `make_params()`, you can include values as keyword arguments to either the `eval()` or `fit()` methods:

```
>>> y1 = mod.eval(x=x, a=7.0, b=-2.0)

>>> out = mod.fit(x=x, pars, a=3.0, b=-0.0)
```

These approachess to initialization provide many opportunities for setting initial values for parameters. The methods can be combined, so that you can set parameter hints but then change the initial value explicitly with `fit()`.

5.2.8 Using parameter hints

After a model has been created, you can give it hints for how to create parameters with `make_params()`. This allows you to set not only a default initial value but also to set other parameter attributes controlling bounds, whether it is varied in the fit, or a constraint expression. To set a parameter hint, you can use `set_param_hint()`, as with:

```
>>> mod = Model(myfunc)
>>> mod.set_param_hint('a', value = 1.0)
>>> mod.set_param_hint('b', value = 0.3, min=0, max=1.0)
```

Parameter hints are stored in a model's `param_hints` attribute, which is simply a nested dictionary:

```
>>> print mod.param_hints
{'a': {'value': 1}, 'b': {'max': 1.0, 'value': 0.3, 'min': 0}}
```

You can change this dictionary directly, or with the `set_param_hint()` method. Either way, these parameter hints are used by `make_params()` when making parameters.

An important feature of parameter hints is that you can force the creation of new parameters with parameter hints. This can be useful to make derived parameters with constraint expressions. For example to get the full-width at half maximum of a Gaussian model, one could use a parameter hint of:

```
>>> mod = Model(gaussian)
>>> mod.set_param_hint('fwhm', expr='2.3548*sigma')
```

5.3 The `ModelFit` class

A `ModelFit` is the object returned by `Model.fit()`. It is a sublcass of `Minimizer`, and so contains many of the fit results. Of course, it knows the `Model` and the set of `Parameters` used in the fit, and it has methods to evaluate the model, to fit the data (or re-fit the data with changes to the parameters, or fit with different or modified data) and to print out a report for that fit.

While a `Model` encapsulates your model function, it is fairly abstract and does not contain the parameters or data used in a particular fit. A `ModelFit` *does* contain parameters and data as well as methods to alter and re-do fits. Thus the `Model` is the idealized model while the `ModelFit` is the messier, more complex (but perhaps more useful) object that represents a fit with a set of parameters to data with a model.

A `ModelFit` has several attributes holding values for fit results, and several methods for working with fits.

5.3.1 `ModelFit` methods

These methods are all inherited from `Minimize` or from `Model`.

`eval`(***kwargs*)
evaluate the model using the best-fit parameters and supplied independent variables. The `**kwargs` arguments can be used to update parameter values and/or independent variables.

`fit`(*data=None*[, *params=None*[, *weights=None*[, *method=None*[, ***kwargs*]]]])
fit (or re-fit), optionally changing `data`, `params`, `weights`, or `method`, or changing the independent variable(s) with the `**kwargs` argument. See `Model.fit()` for argument descriptions, and note that any value of `None` defaults to the last used value.

`fit_report`(*modelpars=None*[, *show_correl=True*[, *min_correl=0.1*]])
return a printable fit report for the fit with fit statistics, best-fit values with uncertainties and correlations. As with `fit_report()`.

Parameters

- **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]

5.3.2 `ModelFit` attributes

`best_fit`
ndarray result of model function, evaluated at provided independent variables and with best-fit parameters.

`best_values`
dictionary with parameter names as keys, and best-fit values as values.

`chisqr`
floating point best-fit chi-square statistic.

`covar`
ndarray (square) covariance matrix returned from fit.

`data`
ndarray of data to compare to model.

`errorbars`
boolean for whether error bars were estimated by fit.

`ier`
integer returned code from `scipy.optimize.leastsq()`.

`init_fit`
ndarray result of model function, evaluated at provided independent variables and with initial parameters.

`init_params`
initial parameters.

`init_values`
dictionary with parameter names as keys, and initial values as values.

`iter_cb`
optional callable function, to be called at each fit iteration. This must take take arguments of `params, iter, resid, *args, **kws`, where `params` will have the current parameter values, `iter` the iteration, `resid` the current residual array, and `*args` and `**kws` as passed to the objective function.

`jacfcn`
optional callable function, to be called to calculate jacobian array.

`lmdif_message`
string message returned from `scipy.optimize.leastsq()`.

`message`
string message returned from `minimize()`.

`method`
string naming fitting method for `minimize()`.

`model`
instance of `Model` used for model.

`ndata`
integer number of data points.

`nfev`
integer number of function evaluations used for fit.

nfree
integer number of free paramaeters in fit.

nvarys
integer number of independent, freely varying variables in fit.

params
Parameters used in fit. Will have best-fit values.

redchi
floating point reduced chi-square statistic

residual
ndarray for residual.

scale_covar
boolean flag for whether to automatically scale covariance matrix.

success
boolean value of whether fit succeeded.

weights
ndarray (or `None`) of weighting values used in fit.

5.4 Creating composite models

One of the most interesting features of the `Model` class is that models can be added together to give a composite model, with parameters from the component models all being available to influence the total sum of the separat component models. This will become even more useful in the next chapter, when pre-built subclasses of `Model` are discussed.

For now, we'll consider a simple example will build a model of a Gaussian plus a line. Obviously, we could build a model that included both components:

```python
def gaussian_plus_line(x, amp, cen, wid, slope, intercept):
    "line + 1-d gaussian"

    gauss = (amp/(sqrt(2*pi)*wid)) * exp(-(x-cen)**2 /(2*wid**2))
    line = slope * x + intercept
    return gauss + line
```

and use that with:

```python
mod = Model(gaussian_plus_line)
```

but, of course, we already had a function for a gaussian function, and maybe we'll discover that a linear background isn't sufficient and we'd have to alter the model again. As an alternative we could just define a linear function:

```python
def line(x, slope, intercept):
    "a line"
    return slope * x + intercept
```

and build a composite model with:

```python
mod = Model(gaussian) + Model(line)
```

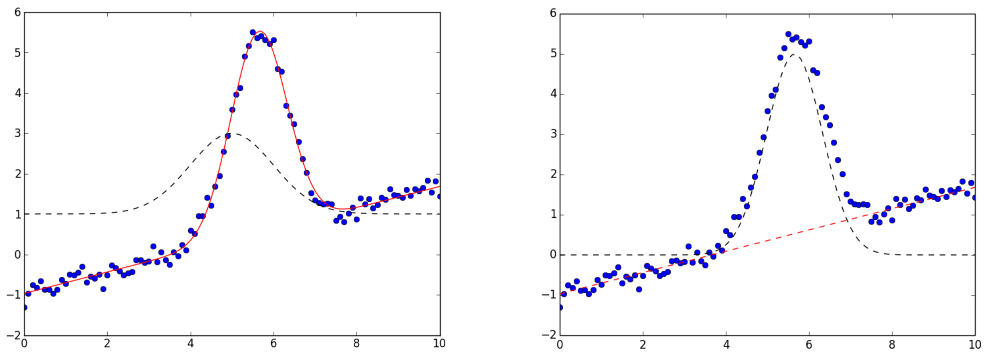
This model has parameters for both component models, and can be used as:

```python
#!/usr/bin/env python
#<examples/model_doc2.py>
from numpy import sqrt, pi, exp, linspace, loadtxt
from lmfit import Model

import matplotlib.pyplot as plt

data = loadtxt('model1d_gauss.dat')
x = data[:, 0]
y = data[:, 1] + 0.25*x - 1.0

def gaussian(x, amp, cen, wid):
    "1-d gaussian: gaussian(x, amp, cen, wid)"
    return (amp/(sqrt(2*pi)*wid)) * exp(-(x-cen)**2 /(2*wid**2))

def line(x, slope, intercept):
    "line"
    return slope * x + intercept

mod = Model(gaussian) + Model(line)
pars  = mod.make_params( amp=5, cen=5, wid=1, slope=0, intercept=1)

print mod

for k, v in pars.items():
    print k, v

result = mod.fit(y, pars, x=x)

print(result.fit_report())

plt.plot(x, y,         'bo')
plt.plot(x, result.init_fit, 'k--')
plt.plot(x, result.best_fit, 'r-')
plt.show()
#<end examples/model_doc2.py>
```

which prints out the results:

```
[[Model]]
 Composite Model:
    gaussian
    line
[[Fit Statistics]]
    # function evals   = 44
    # data points      = 101
    # variables        = 5
    chi-square         = 2.579
    reduced chi-square = 0.027
[[Variables]]
    amp:         8.45931061 +/- 0.124145 (1.47%) (init= 5)
    cen:         5.65547872 +/- 0.009176 (0.16%) (init= 5)
    intercept:  -0.96860201 +/- 0.033522 (3.46%) (init= 1)
    slope:       0.26484403 +/- 0.005748 (2.17%) (init= 0)
    wid:         0.67545523 +/- 0.009916 (1.47%) (init= 1)
[[Correlations]] (unreported correlations are <  0.100)
    C(amp, wid)                  =  0.666
    C(cen, intercept)            =  0.129
```

and shows the plot on the left.



On the left, data is shown in blue dots, the total fit is shown in solid red line, and the initial fit is shown as a black dashed line. In the figure on the right, the data is again shown in blue dots, and the Gaussian component shown as a black dashed line, and the linear component shown as a red dashed line. These components were generated after the fit using the Models `eval()` method:

```
comp_gauss = mod.components[0].eval(x=x)
comp_line  = mod.components[1].eval(x=x)
```

Note that we have to pass in `x` here, but not any of the final values for the parameters – the current values for `mod.params` will be used, and these will be the best-fit values after a fit. While the model does store the best parameters and the estimate of the data in `mod.best_fit`, it does not actually store the data it fit to or the independent variables – here, `x` for that data. That means you can easily apply this model to other data sets, or evaluate the model at other values of `x`. You may want to do this to give a finer or coarser spacing of data point, or to extrapolate the model outside the fitting range. This can be done with:

```
xwide = np.linspace(-5, 25, 3001)
predicted = mod.eval(x=xwide)
```

A final note: In this example, the argument names for the model functions do not overlap. If they had, the `prefix` argument to `Model` would have allowed us to identify which parameter went with which component model. As we will see in the next chapter, using composite models with the built-in models provides a simple way to build up complex models.

5.4.1 Model names for composite models

By default a *Model* object has a *name* attribute containing the name of the model function. This name can be overridden when building a model:

```
my_model = Model(gaussian, name='my_gaussian')
```

or by assigning the *name* attribute:

```
my_model = Model(gaussian)
my_model.name = 'my_gaussian'
```

This name is used in the object representation (for example when printing):

```
<lmfit.Model: my_gaussian>
```

A composite model will have the name *'composite_fun'* by default, but as noted, we can overwrite it with a more meaningful string. This can be useful when dealing with multiple models.

For example, let assume we want to fit some bi-modal data. We initially try two Gaussian peaks:

```
model = GaussianModel(prefix='p1_') + GaussianModel(prefix='p2_')
model.name = '2-Gaussians model'
```

Here, instead of the standard name *'composite_func'*, we assigned a more meaningful name. Now, if we want to also fit with two Lorentzian peaks we can do similarly:

```
model2 = LorentzianModel(prefix='p1_') + LorentzianModel(prefix='p2_')
model2.name = '2-Lorentzians model'
```

It is evident that assigning names will help to easily distinguish the different models.

CHAPTER

SIX

BUILT-IN FITTING MODELS IN THE MODELS MODULE

Lmfit provides several builtin fitting models in the `models` module. These pre-defined models each subclass from the `Model` class of the previous chapter and wrap relatively well-known functional forms, such as Gaussians, Lorentzian, and Exponentials that are used in a wide range of scientific domains. In fact, all the models are all based on simple, plain python functions defined in the `lineshapes` module. In addition to wrapping a function into a `Model`, these models also provide a `guess()` method that is intended to give a reasonable set of starting values from a data array that closely approximates the data to be fit.

As shown in the previous chapter, a key feature of the `Model` class is that models can easily be combined to give a composite `Model`. Thus while some of the models listed here may seem pretty trivial (notably, `ConstantModel` and `LinearModel`), the main point of having these is to be able to used in composite models. For example, a Lorentzian plus a linear background might be represented as:

```
>>> from lmfit.models import LinearModel, LorentzianModel
>>> peak = LorentzianModel()
>>> background  = LinearModel()
>>> model = peak + background
```

All the models listed below are one dimensional, with an independent variable named `x`. Many of these models represent a function with a distinct peak, and so share common features. To maintain uniformity, common parameter names are used whenever possible. Thus, most models have a parameter called `amplitude` that represents the overall height (or area of) a peak or function, a `center` parameter that represents a peak centroid position, and a `sigma` parameter that gives a characteristic width. Some peak shapes also have a parameter `fwhm`, typically constrained by `sigma` to give the full width at half maximum.

After a list of builtin models, a few examples of their use is given.

6.1 Peak-like models

There are many peak-like models available. These include `GaussianModel`, `LorentzianModel`, `VoigtModel` and some less commonly used variations. The `guess()` methods for all of these make a fairly crude guess for the value of `amplitude`, but also set a lower bound of 0 on the value of `sigma`.

6.1.1 `GaussianModel`

class `GaussianModel`

A model based on a Gaussian or normal distribution lineshape. Parameter names: `amplitude`, `center`, and `sigma`. In addition, a constrained parameter `fwhm` is included.

$$f(x; A, \mu, \sigma) = \frac{A}{\sigma\sqrt{2\pi}} e^{[-(x-\mu)^2/2\sigma^2]}$$

where the parameter `amplitude` corresponds to A, `center` to μ, and `sigma` to σ. The Full-Width at Half-Maximum is $2\sigma\sqrt{2\ln 2}$, approximately 2.3548σ

6.1.2 `LorentzianModel`

class `LorentzianModel`

A model based on a Lorentzian or Cauchy-Lorentz distribution function. Parameter names: `amplitude`, `center`, and `sigma`. In addition, a constrained parameter `fwhm` is included.

$$f(x; A, \mu, \sigma) = \frac{A}{\pi} \big[\frac{\sigma}{(x - \mu)^2 + \sigma^2}\big]$$

where the parameter `amplitude` corresponds to A, `center` to μ, and `sigma` to σ. The Full-Width at Half-Maximum is 2σ.

6.1.3 `VoigtModel`

class `VoigtModel`

A model based on a Voigt distribution function. Parameter names: `amplitude`, `center`, and `sigma`. A `gamma` parameter is also available. By default, it is constrained to have value equal to `sigma`, though this can be varied independently. In addition, a constrained parameter `fwhm` is included. The definition for the Voigt function used here is

$$f(x; A, \mu, \sigma, \gamma) = \frac{A \textrm{Re}[w(z)]}{\sigma\sqrt{2 \pi}}$$

where

$$\begin{aligned} z &= \frac{x-\mu +i\gamma}{\sigma\sqrt{2}} \\ w(z) &= e^{-z^2}\textrm{erfc}(-iz) \end{aligned}$$

and `erfc()` is the complimentary error function. As above, `amplitude` corresponds to A, `center` to μ, and `sigma` to σ. The parameter `gamma` corresponds to γ. If `gamma` is kept at the default value (constrained to `sigma`), the full width at half maximum is approximately 3.6013σ.

6.1.4 `PseudoVoigtModel`

class `PseudoVoigtModel`

a model based on a pseudo-Voigt distribution function, which is a weighted sum of a Gaussian and Lorentzian distribution functions with the same values for `amplitude` (A), `center` (μ) and `sigma` (σ), and a parameter `fraction` (α) in

$$f(x; A, \mu, \sigma, \alpha) = (1-\alpha)\frac{A}{\pi} \big[\frac{\sigma}{(x - \mu)^2 + \sigma^2}\big] + \frac{\alpha A}{\pi} \big[\frac{\sigma}{(x - \mu)^2 + \sigma^2}\big]$$

The `guess()` function always gives a starting value for `fraction` of 0.5

6.1.5 `Pearson7Model`

class **`Pearson7Model`**

A model based on a Pearson VII distribution. This is a Lorenztian-like distribution function. It has the usual parameters `amplitude` (A), `center` (μ) and `sigma` (σ), and also an `exponent` (m) in

$$f(x; A, \mu, \sigma, m) = \frac{A}{\sigma\beta(m-\frac{1}{2}, \frac{1}{2})}[1 + \frac{(x-\mu)^2}{\sigma^2}]^{-m}$$

where β is the beta function (see `scipy.special.beta()` in `scipy.special`). The `guess()` function always gives a starting value for `exponent` of 1.5.

6.1.6 `StudentsTModel`

class **`StudentsTModel`**

A model based on a Student's t distribution function, with the usual parameters `amplitude` (A), `center` (μ) and `sigma` (σ) in

$$f(x; A, \mu, \sigma) = \frac{A\Gamma(\frac{\sigma+1}{2})}{\sqrt{\sigma\pi}\,\Gamma(\frac{\sigma}{2})}\Big[1 + \frac{(x-\mu)^2}{\sigma}\Big]^{-\frac{\sigma+1}{2}}$$

where $\Gamma(x)$ is the gamma function.

6.1.7 `BreitWignerModel`

class **`BreitWignerModel`**

A model based on a Breit-Wigner-Fano function. It has the usual parameters `amplitude` (A), `center` (μ) and `sigma` (σ), plus `q` (q) in

$$f(x; A, \mu, \sigma, q) = \frac{A(q\sigma/2 + x - \mu)^2}{(\sigma/2)^2 + (x-\mu)^2}$$

6.1.8 `LognormalModel`

class **`LognormalModel`**

A model based on the Log-normal distribution function. It has the usual parameters `amplitude` (A), `center` (μ) and `sigma` (σ) in

$$f(x; A, \mu, \sigma) = \frac{Ae^{-(\ln(x)-\mu)/2\sigma^2}}{x}$$

6.1.9 `DampedOcsillatorModel`

class **`DampedOcsillatorModel`**

A model based on the Damped Harmonic Oscillator Amplitude. It has the usual parameters `amplitude` (A), `center` (μ) and `sigma` (σ) in

$$f(x; A, \mu, \sigma) = \frac{A}{\sqrt{[1-(x/\mu)^2]^2 + (2\sigma x/\mu)^2}}$$

6.1.10 `ExponentialGaussianModel`

class `ExponentialGaussianModel`

A model of an Exponentially modified Gaussian distribution. It has the usual parameters `amplitude` (A), `center` (μ) and `sigma` (σ), and also `gamma` (γ) in

$$f(x; A, \mu, \sigma, \gamma) = \frac{A\gamma}{2} \exp\bigl[\gamma(\mu - x + \sigma^2/2)\bigr] \mathrm{erfc}\bigl[\frac{\mu + \gamma\sigma^2 - x}{\sqrt{2}\sigma}\bigr]$$

where `erfc()` is the complimentary error function.

6.1.11 `DonaichModel`

class `DonaichModel`

A model of an Doniach Sunjic asymmetric lineshape, used in photo-emission. With the usual parameters `amplitude` (A), `center` (μ) and `sigma` (σ), and also `gamma` (γ) in

$$f(x; A, \mu, \sigma, \gamma) = A\frac{\cos\bigl[\pi\gamma/2 + (1-\gamma)\arctan{(x - \mu)}/\sigma\bigr]}{\bigl[1 + (x-\mu)/\sigma\bigr]^{(1-\gamma)/2}}$$

6.2 Linear and Polynomial Models

These models correspond to polynomials of some degree. Of course, lmfit is a very inefficient way to do linear regression (see `numpy.polyfit()` or `scipy.stats.linregress()`), but these models may be useful as one of many components of composite model.

6.2.1 `ConstantModel`

class `ConstantModel`

a class that consists of a single value, `c`. This is constant in the sense of having no dependence on the independent variable `x`, not in the sense of being non-varying. To be clear, `c` will be a variable Parameter.

6.2.2 `LinearModel`

class `LinearModel`

a class that gives a linear model:

$$f(x; m, b) = mx + b$$

with parameters `slope` for m and `intercept` for b.

6.2.3 `QuadraticModel`

class `QuadraticModel`

a class that gives a quadratic model:

$$f(x; a, b, c) = ax^2 + bx + c$$

with parameters `a`, `b`, and `c`.

6.2.4 `ParabolicModel`

class `ParabolicModel`

same as `QuadraticModel`.

6.2.5 `PolynomialModel`

class `PolynomialModel`(*degree*)

a class that gives a polynomial model up to `degree` (with maximum value of 7).

$$f(x; c_0, c_1, \ldots, c_7) = \sum_{i=0,7} c_i x^i$$

with parameters `c0`, `c1`, ..., `c7`. The supplied `degree` will specify how many of these are actual variable parameters. This uses `numpy.polyval()` for its calculation of the polynomial.

6.3 Step-like models

6.3.1 `StepModel`

class `StepModel`(*form='linear'*)

A model based on a Step function, with four choices for functional form. The step function starts with a value 0, and ends with a value of A (`amplitude`), rising to $A/2$ at μ (`center`), with σ (`sigma`) setting the characteristic width. The supported functional forms are `linear` (the default), `atan` or `arctan` for an arc-tangent function, `erf` for an error function, or `logistic` for a logistic function. The forms are

$$\begin{aligned} f(x; A, \mu, \sigma, \mathrm{form} = '\mathrm{linear}') &= A \min\left[1, \max\left(0, \alpha\right)\right] \\ f(x; A, \mu, \sigma, \mathrm{form} = '\mathrm{arctan}') &= A[1/2 + \arctan\left(\alpha\right)/\pi] \\ f(x; A, \mu, \sigma, \mathrm{form} = '\mathrm{erf}') &= A[1 + \mathrm{erf}(\alpha)]/2 \\ f(x; A, \mu, \sigma, \mathrm{form} = '\mathrm{logistic}') &= A[1 - \frac{1}{1 + e^{\alpha}}] \end{aligned}$$

where $\alpha = (x - \mu)/\sigma$.

6.3.2 `RectangleModel`

class `RectangleModel`(*form='linear'*)

A model based on a Step-up and Step-down function of the same form. The same choices for functional form as for `StepModel` are supported, with `linear` as the default. The function starts with a value 0, and ends with a value of A (`amplitude`), rising to $A/2$ at μ_1 (`center1`), with σ_1 (`sigma1`) setting the characteristic width. It drops to rising to $A/2$ at μ_2 (`center2`), with characteristic width σ_2 (`sigma2`).

$$\begin{aligned} f(x; A, \mu, \sigma, \mathrm{form} = '\mathrm{linear}') &= A\{\min\left[1, \max\left(0, \alpha_1\right)\right] + \min\left[-1, \max\left(0, \alpha_2\right)\right]\} \\ f(x; A, \mu, \sigma, \mathrm{form} = '\mathrm{arctan}') &= A[\arctan\left(\alpha_1\right) + \arctan\left(\alpha_2\right)]/\pi \\ f(x; A, \mu, \sigma, \mathrm{form} = '\mathrm{erf}') &= A[\mathrm{erf}(\alpha_1) + \mathrm{erf}(\alpha_2)]/2 \\ f(x; A, \mu, \sigma, \mathrm{form} = '\mathrm{logistic}') &= A[1 - \frac{1}{1 + e^{\alpha_1}} - \frac{1}{1 + e^{\alpha_2}}] \end{aligned}$$

where $\alpha_1 = (x - \mu_1)/\sigma_1$ and $\alpha_2 = -(x - \mu_2)/\sigma_2$.

6.4 Exponential and Power law models

6.4.1 `ExponentialModel`

class `ExponentialModel`

A model based on an exponential decay function. With parameters named `amplitude` (A), and `decay` (τ), this has the form:

$$f(x; A, \tau) = Ae^{-x/\tau}$$

6.4.2 `PowerLawModel`

class `PowerLawModel`

A model based on a Power Law. With parameters named `amplitude` (A), and `exponent` (k), this has the form:

$$f(x; A, k) = Ax^k$$

6.5 Example 1: Fit Peaked data to Gaussian, Lorentzian, and Voigt profiles

Here, we will fit data to three similar lineshapes, in order to decide which might be the better model. We will start with a Gaussian profile, as in the previous chapter, but use the built-in `GaussianModel` instead of one we write ourselves. This is a slightly different version from the one in previous example in that the parameter names are different, and have built-in default values. So, we'll simply use:

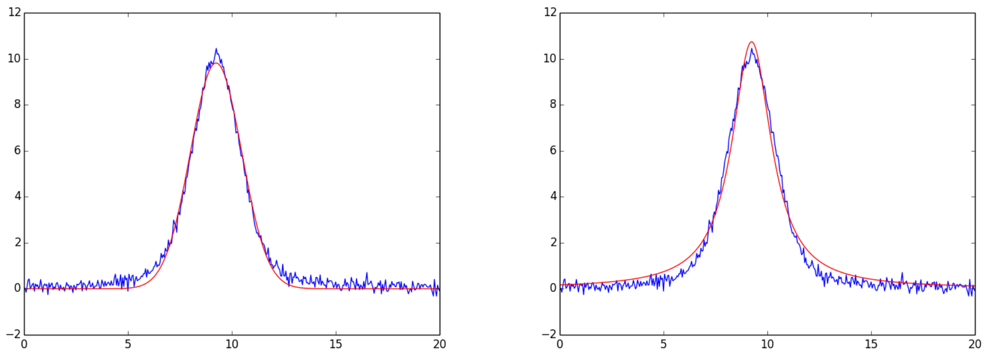
```
from numpy import loadtxt
from lmfit.models import GaussianModel

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

mod = GaussianModel()
pars = mod.guess(y, x=x)
out  = mod.fit(y, pars, x=x)
print(out.fit_report(min_correl=0.25))
```

which prints out the results:

```
[[Model]]
    gaussian
[[Fit Statistics]]
    # function evals   = 21
    # data points      = 401
    # variables        = 3
    chi-square         = 29.994
    reduced chi-square = 0.075
[[Variables]]
    amplitude:   30.3135571 +/- 0.157126 (0.52%) (init= 29.08159)
    center:      9.24277049 +/- 0.007374 (0.08%) (init= 9.25)
    fwhm:        2.90156963 +/- 0.017366 (0.60%)  == '2.3548200*sigma'
    sigma:       1.23218319 +/- 0.007374 (0.60%) (init= 1.35)
```

```
[[Correlations]] (unreported correlations are <  0.250)
    C(amplitude, sigma)          =  0.577
```

[We see a few interesting differences from the results of the previous chapter. First, the parameter names are longer. Second, there is a `fwhm` parameter, defined as $\sim 2.355\sigma$. And third, the automated initial guesses are pretty good. A plot of the fit shows not such a great fit:


Fit to peak with Gaussian (left) and Lorentzian (right) models.

suggesting that a different peak shape, with longer tails, should be used. Perhaps a Lorentzian would be better? To do this, we simply replace `GaussianModel` with `LorentzianModel` to get a `LorentzianModel`:

```python
from lmfit.models import LorentzianModel
mod = LorentzianModel()
pars = mod.guess(y, x=x)
out  = mod.fit(y, pars, x=x)
print(out.fit_report(min_correl=0.25))
```

Predictably, the first thing we try gives results that are worse:

```
[[Model]]
    lorentzian
[[Fit Statistics]]
    # function evals   = 25
    # data points      = 401
    # variables        = 3
    chi-square         = 53.754
    reduced chi-square = 0.135
[[Variables]]
    amplitude:   38.9728645 +/- 0.313857 (0.81%) (init= 36.35199)
    center:      9.24438944 +/- 0.009275 (0.10%) (init= 9.25)
    fwhm:        2.30969034 +/- 0.026312 (1.14%)  == '2.0000000*sigma'
    sigma:       1.15484517 +/- 0.013156 (1.14%) (init= 1.35)
[[Correlations]] (unreported correlations are <  0.250)
    C(amplitude, sigma)          =  0.709
```

with the plot shown on the right in the figure above.

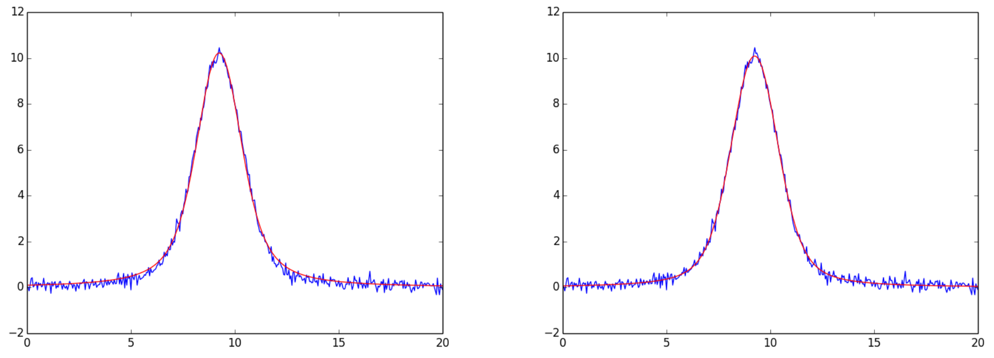
A Voigt model does a better job. Using `VoigtModel`, this is as simple as:

```python
from lmfit.models import VoigtModel
mod = VoigtModel()
pars = mod.guess(y, x=x)
out  = mod.fit(y, pars, x=x)
print(out.fit_report(min_correl=0.25))
```

which gives:

```
[[Model]]
    voigt
[[Fit Statistics]]
    # function evals   = 17
    # data points      = 401
    # variables        = 3
    chi-square         = 14.545
    reduced chi-square = 0.037
[[Variables]]
    amplitude:   35.7554017 +/- 0.138614 (0.39%) (init= 43.62238)
    center:      9.24411142 +/- 0.005054 (0.05%) (init= 9.25)
    fwhm:        2.62951718 +/- 0.013269 (0.50%)  == '3.6013100*sigma'
    gamma:       0.73015574 +/- 0.003684 (0.50%)  == 'sigma'
    sigma:       0.73015574 +/- 0.003684 (0.50%) (init= 0.8775)
[[Correlations]] (unreported correlations are <  0.250)
    C(amplitude, sigma)          =  0.651
```

with the much better value for χ^2 and the obviously better match to the data as seen in the figure below (left).



Fit to peak with Voigt model (left) and Voigt model with `gamma` varying independently of `sigma` (right).

The Voigt function has a γ parameter (`gamma`) that can be distinct from `sigma`. The default behavior used above constrains `gamma` to have exactly the same value as `sigma`. If we allow these to vary separately, does the fit improve? To do this, we have to change the `gamma` parameter from a constrained expression and give it a starting value:

```
mod = VoigtModel()
pars = mod.guess(y, x=x)
pars['gamma'].set(value=0.7, vary=True, expr='')

out  = mod.fit(y, pars, x=x)
print(out.fit_report(min_correl=0.25))
```

which gives:

```
[[Model]]
    voigt
[[Fit Statistics]]
    # function evals   = 21
    # data points      = 401
    # variables        = 4
    chi-square         = 10.930
    reduced chi-square = 0.028
[[Variables]]
    amplitude:   34.1914716 +/- 0.179468 (0.52%) (init= 43.62238)
    center:      9.24374845 +/- 0.004419 (0.05%) (init= 9.25)
    fwhm:        3.22385491 +/- 0.050974 (1.58%)  == '3.6013100*sigma'
    gamma:       0.52540157 +/- 0.018579 (3.54%) (init= 0.7)
    sigma:       0.89518950 +/- 0.014154 (1.58%) (init= 0.8775)
[[Correlations]] (unreported correlations are <  0.250)
    C(amplitude, gamma)          =  0.821
```

and the fit shown on the right above.

Comparing the two fits with the Voigt function, we see that χ^2 is definitely improved with a separately varying `gamma` parameter. In addition, the two values for `gamma` and `sigma` differ significantly – well outside the estimated uncertainties. Even more compelling, reduced χ^2 is improved even though a fourth variable has been added to the fit. In the simplest statistical sense, this suggests that `gamma` is a significant variable in the model.

This example shows how easy it can be to alter and compare fitting models for simple problems. The example is included in the `doc_peakmodels.py` file in the examples directory.

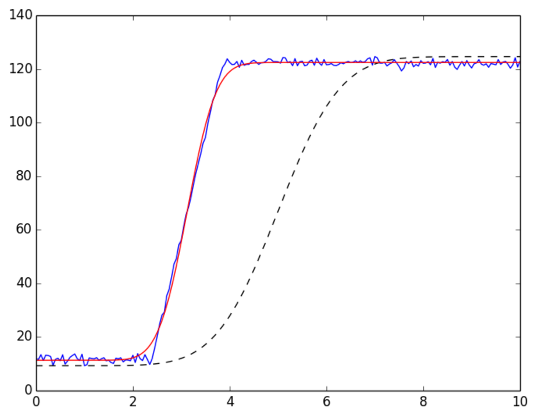
6.6 Example 2: Fit data to a Composite Model with pre-defined models

Here, we repeat the point made at the end of the last chapter that instances of `Model` class can be added them together to make a *composite model*. But using the large number of built-in models available, this is very simple. An example of a simple fit to a noisy step function plus a constant:

```python
#!/usr/bin/env python
#<examples/doc_stepmodel.py>
import numpy as np
from lmfit.models import StepModel, LinearModel

import matplotlib.pyplot as plt

x = np.linspace(0, 10, 201)
y = np.ones_like(x)
y[:48] = 0.0
y[48:77] = np.arange(77-48)/(77.0-48)
y = 110.2 * (y + 9e-3*np.random.randn(len(x))) + 12.0 + 2.22*x

step_mod = StepModel(form='erf', prefix='step_')
line_mod = LinearModel(prefix='line_')

pars =  line_mod.make_params(intercept=y.min(), slope=0)
pars += step_mod.guess(y, x=x, center=2.5)

mod = step_mod + line_mod
out = mod.fit(y, pars, x=x)

print(out.fit_report())

plt.plot(x, y)
plt.plot(x, out.init_fit, 'k--')
plt.plot(x, out.best_fit, 'r-')
plt.show()
#<end examples/doc_stepmodel.py>
```

After constructing step-like data, we first create a `StepModel` telling it to use the `erf` form (see details above), and a `ConstantModel`. We set initial values, in one case using the data and `guess()` method for the intial step function paramaters, and `make_params()` arguments for the linear component. After making a composite model, we run `fit()` and report the results, which give:

```
[[Model]]
 Composite Model:
    step(prefix='step_',form='erf')
    linear(prefix='line_')
[[Fit Statistics]]
    # function evals   = 49
    # data points      = 201
    # variables        = 5
    chi-square         = 633.465
    reduced chi-square = 3.232
[[Variables]]
    line_intercept:   11.5685248 +/- 0.285611 (2.47%) (init= 10.72406)
    line_slope:       2.03270159 +/- 0.096041 (4.72%) (init= 0)
    step_amplitude:   112.270535 +/- 0.674790 (0.60%) (init= 136.3006)
    step_center:      3.12343845 +/- 0.005370 (0.17%) (init= 2.5)
    step_sigma:       0.67468813 +/- 0.011336 (1.68%) (init= 1.428571)
[[Correlations]] (unreported correlations are <  0.100)
    C(step_amplitude, step_sigma)   =  0.564
    C(line_intercept, step_center)   =  0.428
    C(step_amplitude, step_center)   =  0.109
```

with a plot of 

6.7 Example 3: Fitting Multiple Peaks – and using Prefixes

As shown above, many of the models have similar parameter names. For composite models, this could lead to a problem of having parameters for different parts of the model having the same name. To overcome this, each `Model` can have a `prefix` attribute (normally set to a blank string) that will be put at the beginning of each parameter name. To illustrate, we fit one of the classic datasets from the NIST StRD suite involving a decaying exponential and two gaussians.

```python
#!/usr/bin/env python
#<examples/doc_nistgauss.py>
import numpy as np
from lmfit.models import GaussianModel, ExponentialModel
import sys
import matplotlib.pyplot as plt

dat = np.loadtxt('NIST_Gauss2.dat')
```

```
x = dat[:, 1]
y = dat[:, 0]

exp_mod = ExponentialModel(prefix='exp_')
pars = exp_mod.guess(y, x=x)

gauss1  = GaussianModel(prefix='g1_')
pars.update( gauss1.make_params())

pars['g1_center'].set(105, min=75, max=125)
pars['g1_sigma'].set(15, min=3)
pars['g1_amplitude'].set(2000, min=10)

gauss2  = GaussianModel(prefix='g2_')

pars.update(gauss2.make_params())

pars['g2_center'].set(155, min=125, max=175)
pars['g2_sigma'].set(15, min=3)
pars['g2_amplitude'].set(2000, min=10)

mod = gauss1 + gauss2 + exp_mod


init = mod.eval(pars, x=x)
plt.plot(x, y)
plt.plot(x, init, 'k--')

out = mod.fit(y, pars, x=x)

print(out.fit_report(min_correl=0.5))

plt.plot(x, out.best_fit, 'r-')
plt.show()
#<end examples/doc_nistgauss.py>
```

where we give a separate prefix to each model (they all have an `amplitude` parameter). The `prefix` values are attached transparently to the models.

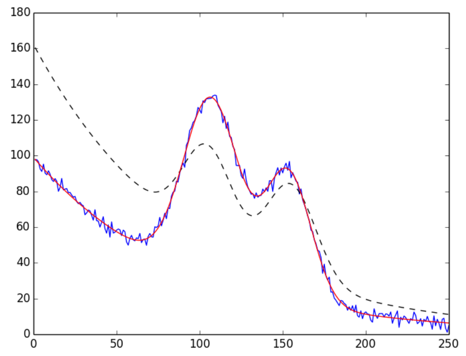
MN—-: Note that the calls to `make_param()` used the bare name, without the prefix. We could have used them, but because we used the individual model `gauss1` and `gauss2`, there was no need.

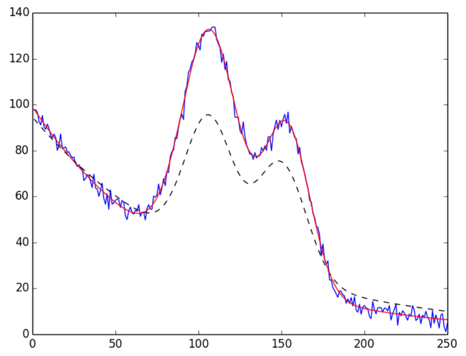
Note also in the example here that we explicitly set bounds on many of the parameter values.

The fit results printed out are:

```
[[Model]]
 Composite Model:
    gaussian(prefix='g1_')
    gaussian(prefix='g2_')
    exponential(prefix='exp_')
[[Fit Statistics]]
    # function evals   = 55
    # data points      = 250
    # variables        = 8
    chi-square         = 1247.528
    reduced chi-square = 5.155
[[Variables]]
    exp_amplitude:   99.0183291 +/- 0.537487 (0.54%) (init= 162.2102)
    exp_decay:       90.9508788 +/- 1.103104 (1.21%) (init= 93.24905)
    g1_amplitude:    4257.77384 +/- 42.38354 (1.00%) (init= 2000)
    g1_center:       107.030955 +/- 0.150068 (0.14%) (init= 105)
    g1_fwhm:         39.2609205 +/- 0.377907 (0.96%)  == '2.3548200*g1_sigma'
    g1_sigma:        16.6725781 +/- 0.160482 (0.96%) (init= 15)
    g2_amplitude:    2493.41747 +/- 36.16907 (1.45%) (init= 2000)
    g2_center:       153.270103 +/- 0.194665 (0.13%) (init= 155)
    g2_fwhm:         32.5128760 +/- 0.439860 (1.35%)  == '2.3548200*g2_sigma'
    g2_sigma:        13.8069474 +/- 0.186791 (1.35%) (init= 15)
[[Correlations]] (unreported correlations are <  0.500)
    C(g1_amplitude, g1_sigma)    =  0.824
    C(g2_amplitude, g2_sigma)    =  0.815
    C(g1_sigma, g2_center)       =  0.684
    C(g1_amplitude, g2_center)   =  0.648
    C(g1_center, g2_center)      =  0.621
    C(g1_center, g1_sigma)       =  0.507
```

We get a very good fit to this challenging problem (described at the NIST site as of average difficulty, but the tests there are generally hard) by applying reasonable initial guesses and putting modest but explicit bounds on the parameter values. This fit is shown on the left:





One final point on setting initial values. From looking at the data itself, we can see the two Gaussian peaks are reasonably well separated but do overlap. Furthermore, we can tell that the initial guess for the decaying exponential component was poorly estimated because we used the full data range. We can simplify the initial parameter values by using this, and by defining an `index_of()` function to limit the data range. That is, with:

```
def index_of(arrval, value):
    "return index of array *at or below* value "
    if value < min(arrval):  return 0
    return max(np.where(arrval<=value)[0])

ix1 = index_of(x,  75)
ix2 = index_of(x, 135)
ix3 = index_of(x, 175)

exp_mod.guess(y[:ix1], x=x[:ix1])
gauss1.guess(y[ix1:ix2], x=x[ix1:ix2])
gauss2.guess(y[ix2:ix3], x=x[ix2:ix3])
```

we can get a better initial estimate, and the fit converges in fewer steps, getting to identical values (to the precision printed out in the report), and without any bounds on parameters at all:

```
[[Model]]
 Composite Model:
    gaussian(prefix='g1_')
    gaussian(prefix='g2_')
    exponential(prefix='exp_')
[[Fit Statistics]]
    # function evals   = 46
    # data points      = 250
    # variables        = 8
    chi-square         = 1247.528
    reduced chi-square = 5.155
[[Variables]]
    exp_amplitude:   99.0183281 +/- 0.537487 (0.54%) (init= 94.53724)
    exp_decay:       90.9508863 +/- 1.103105 (1.21%) (init= 111.1985)
    g1_amplitude:    4257.77321 +/- 42.38338 (1.00%) (init= 2126.432)
    g1_center:       107.030954 +/- 0.150067 (0.14%) (init= 106.5)
    g1_fwhm:         39.2609141 +/- 0.377905 (0.96%)  == '2.3548200*g1_sigma'
    g1_sigma:        16.6725754 +/- 0.160481 (0.96%) (init= 14.5)
    g2_amplitude:    2493.41766 +/- 36.16948 (1.45%) (init= 1878.892)
    g2_center:       153.270100 +/- 0.194667 (0.13%) (init= 150)
    g2_fwhm:         32.5128777 +/- 0.439866 (1.35%)  == '2.3548200*g2_sigma'
    g2_sigma:        13.8069481 +/- 0.186794 (1.35%) (init= 15)
[[Correlations]] (unreported correlations are <  0.500)
    C(g1_amplitude, g1_sigma)    =  0.824
    C(g2_amplitude, g2_sigma)    =  0.815
    C(g1_sigma, g2_center)       =  0.684
    C(g1_amplitude, g2_center)   =  0.648
    C(g1_center, g2_center)      =  0.621
    C(g1_center, g1_sigma)       =  0.507
```

This example is in the file `doc_nistgauss2.py` in the examples folder, and the fit result shown on the right above shows an improved initial estimate of the data.

CHAPTER

SEVEN

CALCULATION OF CONFIDENCE INTERVALS

The lmfit `confidence` module allows you to explicitly calculate confidence intervals for variable parameters. 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.

7.1 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).

Adding a log-likelihood method is under consideration.

7.2 A basic example

First we create an example problem:

```
>>> import lmfit
>>> import numpy as np
>>> x = np.linspace(0.3,10,100)
>>> y = 1/(0.1*x)+2+0.1*np.random.randn(x.size)
>>> p = lmfit.Parameters()
>>> p.add_many(('a', 0.1), ('b', 1))
>>> def residual(p):
...     a = p['a'].value
...     b = p['b'].value
...     return 1/(a*x)+b-y
```

before we can generate the confidence intervals, we have to run a fit, so that the automated estimate of the standard errors can be used as a starting point:

```
>>> mi = lmfit.minimize(residual, p)
>>> lmfit.printfuncs.report_fit(mi.params)
[Variables]]
    a:   0.09943895 +/- 0.000193 (0.19%) (init= 0.1)
    b:   1.98476945 +/- 0.012226 (0.62%) (init= 1)
[[Correlations]] (unreported correlations are <  0.100)
    C(a, b)                      =  0.601
```

Now it is just a simple function call to calculate the confidence intervals:

```
>>> ci = lmfit.conf_interval(mi)
>>> lmfit.printfuncs.report_ci(ci)
      99.70%    95.00%    67.40%     0.00%    67.40%    95.00%    99.70%
a    0.09886   0.09905   0.09925   0.09944   0.09963   0.09982   0.10003
b    1.94751   1.96049   1.97274   1.97741   1.99680   2.00905   2.02203
```

This shows the best-fit values for the parameters in the *0.00%* column, and parameter values that are at the varying confidence levels given by steps in σ. As we can see, the estimated error is almost the same, and the uncertainties are well behaved: Going from 1 σ (68% confidence) to 3 σ (99.7% confidence) uncertainties is fairly linear. It can also be seen that the errors are fairy symmetric around the best fit value. For this problem, it is not necessary to calculate confidence intervals, and the estimates of the uncertainties from the covariance matrix are sufficient.

7.3 An advanced example

Now we look at a problem where calculating the error from approximated covariance can lead to misleading result – two decaying exponentials. In fact such a problem is particularly hard for the Levenberg-Marquardt method, so we fitst estimate the results using the slower but robust Nelder-Mead method, and *then* use Levenberg-Marquardt to estimate the uncertainties and correlations:

```
>>> x = np.linspace(1, 10, 250)
>>> np.random.seed(0)
>>> y = 3.0*np.exp(-x/2) -5.0*np.exp(-(x-0.1)/10.) + 0.1*np.random.randn(len(x))
>>>
>>> p = lmfit.Parameters()
>>> p.add_many(('a1', 4.), ('a2', 4.), ('t1', 3.), ('t2', 3.))
>>>
>>> def residual(p):
...     v = p.valuesdict()
...     return v['a1']*np.exp(-x/v['t1']) + v['a2']*np.exp(-(x-0.1)/v['t2'])-y
>>>
>>> # first solve with Nelder-Mead
>>> mi = lmfit.minimize(residual, p, method='Nelder')
>>> # then solve with Levenberg-Marquardt
>>> mi = lmfit.minimize(residual, p)
>>>
>>> lmfit.printfuncs.report_fit(mi.params, min_correl=0.5)

[[Variables]]
    a1:   2.98622120 +/- 0.148671 (4.98%) (init= 2.986237)
    a2:  -4.33526327 +/- 0.115275 (2.66%) (init=-4.335256)
    t1:   1.30994233 +/- 0.131211 (10.02%) (init= 1.309932)
    t2:   11.8240350 +/- 0.463164 (3.92%) (init= 11.82408)
[[Correlations]] (unreported correlations are <  0.500)
    C(a2, t2)                    =  0.987
```

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

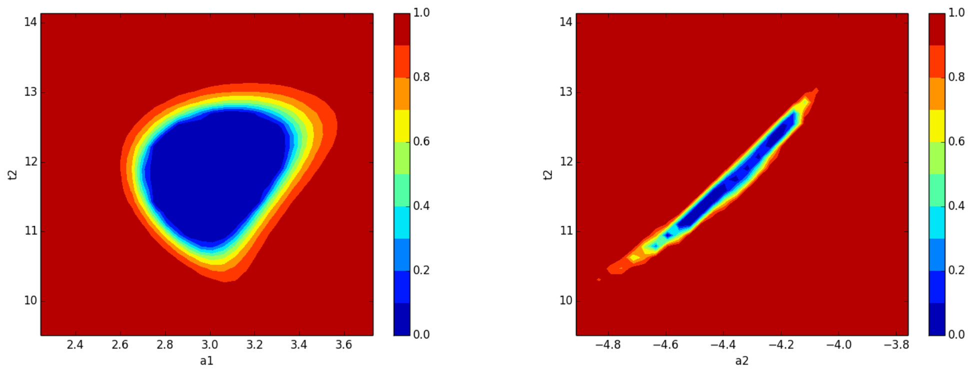
```
>>> ci, trace = lmfit.conf_interval(mi, sigmas=[0.68,0.95], trace=True, verbose=False)
>>> lmfit.printfuncs.report_ci(ci)

      95.00%     68.00%      0.00%     68.00%     95.00%
a1   2.71850    2.84525    2.98622    3.14874    3.34076
a2  -4.63180   -4.46663   -4.35429   -4.22883   -4.14178
t2  10.82699   11.33865   11.78219   12.28195   12.71094
t1   1.08014    1.18566    1.38044    1.45566    1.62579
```

Comparing these two different estimates, we see that the estimate for *a1* is reasonably well approximated from the covariance matrix, but the estimates for *a2* and especially for *t1*, and *t2* are very asymmetric and that going from 1 σ (68% confidence) to 2 σ (95% confidence) is not very predictable.

Now let's plot a confidence region:

```
>>> import matplotlib.pylab as plt
>>> x, y, grid = lmfit.conf_interval2d(mi,'a1','t2',30,30)
>>> plt.contourf(x, y, grid, np.linspace(0,1,11))
>>> plt.xlabel('a1')
>>> plt.colorbar()
>>> plt.ylabel('t2')
>>> plt.show()
```

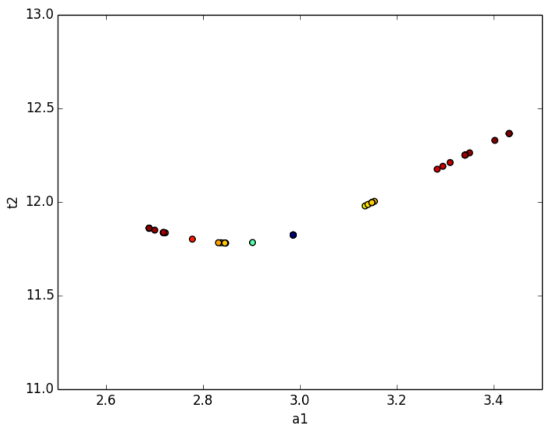
which shows the figure on the left below for `a1` and `t2`, and for `a2` and `t2` on the right:



Neither of these plots is very much like an ellipse, which is implicitly assumed by the approach using the covariance matrix.

Remember the trace? It shows also shows the dependence between two parameters:

```
>>> x, y, prob = trace['a1']['a1'], trace['a1']['t2'],trace['a1']['prob']
>>> x2, y2, prob2 = trace['t2']['t2'], trace['t2']['a1'],trace['t2']['prob']
>>> plt.scatter(x, y, c=prob ,s=30)
>>> plt.scatter(x2, y2, c=prob2, s=30)
>>> plt.gca().set_xlim((1, 5))
>>> plt.gca().set_ylim((5, 15))
>>> plt.xlabel('a1')
>>> plt.ylabel('t2')
>>> plt.show()
```



which shows the trace of values:

7.4 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

EIGHT

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 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

NINE

USING MATHEMATICAL CONSTRAINTS

Being able to fix variables to a constant value or place upper and lower bounds on their values can greatly simplify modeling real data. These capabilities are key to lmfit's Parameters. In addition, it is sometimes highly desirable to place mathematical constraints on parameter values. For example, one might want to require that two Gaussian peaks have the same width, or have amplitudes that are constrained to add to some value. Of course, one could rewrite the objective or model function to place such requirements, but this is somewhat error prone, and limits the flexibility so that exploring constraints becomes laborious.

To simplify the setting of constraints, Parameters can be assigned a mathematical expression of other Parameters, builtin constants, and builtin mathematical functions that will be used to determine its value. The expressions used for constraints are evaluated using the asteval module, which uses Python syntax, and evaluates the constraint expressions in a safe and isolated namespace.

This approach to mathematical constraints allows one to not have to write a separate model function for two Gaussians where the two `sigma` values are forced to be equal, or where amplitudes are related. Instead, one can write a more general two Gaussian model (perhaps using `GaussianModel`) and impose such constraints on the Parameters for a particular fit.

9.1 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.

9.2 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
```

9.3 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.

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

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

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

PYTHON MODULE INDEX

PYTHON MODULE INDEX

INDEX