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

*Release 1.0.2*

**Matthew Newville, Till Stensitzki, Renee Otten, and others**

**Feb 11, 2021**

# CONTENTS

Lmfit provides a high-level interface to non-linear optimization and curve fitting problems for Python. It builds on and extends many of the optimization methods of scipy.optimize. Initially inspired by (and named for) extending the Levenberg-Marquardt method from scipy.optimize.leastsq, lmfit now provides a number of useful enhancements to optimization and data fitting problems, including:

- Using `Parameter` objects instead of plain floats as variables. A `Parameter` has a value that can be varied during the fit or kept at a fixed value. It can have upper and/or lower bounds. A Parameter can even have a value that is constrained by an algebraic expression of other Parameter values. As a Python object, a Parameter can also have attributes such as a standard error, after a fit that can estimate uncertainties.
- Ease of changing fitting algorithms. Once a fitting model is set up, one can change the fitting algorithm used to find the optimal solution without changing the objective function.
- Improved estimation of confidence intervals. While scipy.optimize.leastsq will automatically calculate uncertainties and correlations from the covariance matrix, the accuracy of these estimates is sometimes questionable. To help address this, lmfit has functions to explicitly explore parameter space and determine confidence levels even for the most difficult cases. Additionally, lmfit will use the `numdifftools` package (if installed) to estimate parameter uncertainties and correlations for algorithms that do not natively support this in SciPy.
- Improved curve-fitting with the `Model` class. This extends the capabilities of scipy.optimize.curve_fit, allowing you to turn a function that models your data into a Python class that helps you parametrize and fit data with that model.
- Many *built-in models* for common lineshapes are included and ready to use.

The lmfit package is Free software, using an Open Source license. 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 provides simple tools to help you build 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 – this is not a tutorial on why or how to perform a minimization or fit data, but is rather aimed at explaining how to use lmfit to do these things.

In order to do a non-linear least-squares fit of a model to data or for any other optimization problem, 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 are to be minimized, typically in the least-squares sense. For many data fitting processes, the latter 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 ($\chi^2$) statistic is often defined as:

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

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

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

As a simple concrete example, one might want to model data with a decaying sine wave, and so write an objective function like this:

```python
from numpy import exp, sin

def residual(variables, x, data, eps_data):
    """Model a decaying sine wave and subtract data."""
    amp = variables[0]
    phaseshift = variables[1]
    freq = variables[2]
    decay = variables[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 this:

```python
from numpy import linspace, random
from scipy.optimize import leastsq
```

(continues on next page)

(continued from previous page)

```
# generate synthetic data with noise
x = linspace(0, 100)
eps_data = random.normal(size=x.size, scale=0.2)
data = 7.5 * sin(x*0.22 + 2.5) * exp(-x*x*0.01) + eps_data

variables = [10.0, 0.2, 3.0, 0.007]
out = leastsq(residual, variables, 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:

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

b) 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.

c) There is no simple, robust way to put bounds on values for the variables, or enforce mathematical relationships between the variables. In fact, the 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 due to the use of traditional arrays to hold the variables, which matches closely the implementation of the underlying Fortran code, but does not fit very well with Python's rich selection of objects and data structures. The key concept in lmfit is to define and use `Parameter` objects instead of plain floating point numbers as the variables for the fit. Using `Parameter` objects (or the closely related `Parameters` – a dictionary of `Parameter` objects), allows one to:

a) forget about the order of variables and refer to Parameters by meaningful names.

b) place bounds on Parameters as attributes, without worrying about preserving the order of arrays for variables and boundaries.

c) fix Parameters, without having to rewrite the objective function.

d) place algebraic constraints on Parameters.

To illustrate the value of this approach, we can rewrite the above example for the decaying sine wave as:

```
from numpy import exp, sin

from lmfit import minimize, Parameters


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

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

    return (data-model) / eps_data
```

(continues on next page)

(continued from previous page)

```
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 holds additional attributes to modify the value during the fit. For example, Parameters can be fixed or bounded. This can be done during definition:

```
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 parameter's value. It can also be done later by setting the corresponding attributes after they have been created:

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

Importantly, our objective function remains unchanged. This means the objective function can simply express the parameterized phenomenon to be modeled, and is separate from the choice of parameters to be varied in the fit.

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, control by the **user** performing the fit, instead of by the author of the objective function.

Finally, in addition to the `Parameters` approach to fitting data, lmfit allows switching optimization methods without changing the objective function, provides tools for generating fitting reports, and provides a better determination of Parameters confidence levels.

CHAPTER

TWO

# DOWNLOADING AND INSTALLATION

## 2.1 Prerequisites

Lmfit works with Python versions 3.6 and higher. Version 0.9.15 is the final version to support Python 2.7.

**Lmfit requires the following Python packages, with versions given:**

- NumPy version 1.18 or higher.
- SciPy version 1.3 or higher.
- asteval version 0.9.21 or higher.
- uncertainties version 3.0.1 or higher.

All of these are readily available on PyPI, and should be installed automatically if installing with `pip install lmfit`.

In order to run the test suite, the pytest package is required. Some functionality requires the emcee (version 3+), corner, pandas, Jupyter, matplotlib, dill, or numdifftools packages. These are not installed automatically, but we highly recommend each of these packages.

For building the documentation and generating the examples gallery, matplotlib, emcee (version 3+), corner, Sphinx, jupyter_sphinx, Pillow, sphinxcontrib-svg2pdfconverter, and cairosvg are required (the latter two only when generating the PDF document).

Please refer to `requirements-dev.txt` for a list of all dependencies that are needed if you want to participate in the development of lmfit.

## 2.2 Downloads

The latest stable version of lmfit is 1.0.2 and is available from PyPI. Check the *Release Notes* for a list of changes compared to earlier releases.

## 2.3 Installation

The easiest way to install lmfit is with:

```
pip install lmfit
```

For Anaconda Python, lmfit is not an official package, but several Anaconda channels provide it, allowing installation with (for example):

```
conda install -c conda-forge lmfit
```

## 2.4 Development Version

To get the latest development version from the lmfit GitHub repository, use:

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

and install using:

```
python setup.py install
```

We welcome all contributions to lmfit! If you cloned the repository for this purpose, please read CONTRIBUTING.md for more detailed instructions.

## 2.5 Testing

A battery of tests scripts that can be run with the pytest testing framework is distributed with lmfit in the `tests` folder. These are automatically run as part of the development process. For any release or any master branch from the git repository, running `pytest` 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 should also run for you. Some of these examples assume that matplotlib has been installed and is working correctly.

## 2.6 Acknowledgements

```
Many people have contributed to lmfit.  The attribution of credit in a
project such as this is difficult to get perfect, and there are no doubt
important contributions that are missing or under-represented here.  Please
consider this file as part of the code and documentation that may have bugs
that need fixing.

Some of the largest and most important contributions (in approximate order
of size of the contribution to the existing code) are from:

  Matthew Newville wrote the original version and maintains the project.

  Renee Otten wrote the brute force method, implemented the basin-hopping
  and AMPGO global solvers, implemented uncertainty calculations for scalar
  minimizers and has greatly improved the code, testing, and documentation
  and overall project.
```

(continues on next page)

(continued from previous page)

```
  Till Stensitzki wrote the improved estimates of confidence intervals, and
  contributed many tests, bug fixes, and documentation.

  A. R. J. Nelson added differential_evolution, emcee, and greatly improved
  the code, docstrings, and overall project.

  Antonino Ingargiola wrote much of the high level Model code and has
  provided many bug fixes and improvements.

  Daniel B. Allan wrote much of the original version of the high level Model
  code, and many improvements to the testing and documentation.

  Austen Fox fixed many of the built-in model functions and improved the
  testing and documentation of these.

  Michal Rawlik added plotting capabilities for Models.


  The method used for placing bounds on parameters was derived from the
  clear description in the MINUIT documentation, and adapted from
  J. J. Helmus's python implementation in leastsqbounds.py.

  E. O. Le Bigot wrote the uncertainties package, a version of which was
  used by lmfit for many years, and is now an external dependency.

  The original AMPGO code came from Andrea Gavana and was adopted for
  lmfit.

  The propagation of parameter uncertainties to uncertainties in a Model
  was adapted from the excellent description at
  https://www.astro.rug.nl/software/kapteyn/kmpfittutorial.html#confidence-and-
↪prediction-intervals,
  which references the original work of: J. Wolberg, Data Analysis Using the
  Method of Least Squares, 2006, Springer.

Additional patches, bug fixes, and suggestions have come from Faustin
Carter, Christoph Deil, Francois Boulogne, Thomas Caswell, Colin Brosseau,
nmearl, Gustavo Pasquevich, Clemens Prescher, LiCode, Ben Gamari, Yoav
Roam, Alexander Stark, Alexandre Beelen, Andrey Aristov, Nicholas Zobrist,
Ethan Welty, Julius Zimmermann, Mark Dean, Arun Persaud, Ray Osborn, @lneuhaus,
Marcel Stimberg, Yoshiera Huang, Leon Foks, Sebastian Weigand, Florian LB,
Michael Hudson-Doyle, Ruben Verweij, @jedzill4, and many others.

The lmfit code obviously depends on, and owes a very large debt to the code
in scipy.optimize.  Several discussions on the SciPy-user and lmfit mailing
lists have also led to improvements in this code.
```

## 2.7 Copyright, Licensing, and Re-distribution

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

(continues on next page)

(continued from previous page)

# CHAPTER THREE

# GETTING HELP

If you have questions, comments, or suggestions for LMFIT, please use the mailing list. This provides an on-line conversation that is both archived well and can be searched easily with standard web searches. If you find a bug in the code or documentation, use GitHub Issues to submit a report. If you have an idea for how to solve the problem and are familiar with Python and GitHub, submitting a GitHub Pull Request would be greatly appreciated.

If you are unsure whether to use the mailing list or the Issue tracker, please start a conversation on the mailing list. That is, the problem you're having may or may not be due to a bug. If it is due to a bug, creating an Issue from the conversation is easy. If it is not a bug, the problem will be discussed and then the Issue will be closed. While one *can* search through closed Issues on GitHub, these are not so easily searched, and the conversation is not easily useful to others later. Starting the conversation on the mailing list with "How do I do this?" or "Why didn't this work?" instead of "This should work and doesn't" is generally preferred, and will better help others with similar questions. Of course, there is not always an obvious way to decide if something is a Question or an Issue, and we will try our best to engage in all discussions.

CHAPTER

FOUR

# FREQUENTLY ASKED QUESTIONS

A list of common questions.

## 4.1 What's the best way to ask for help or submit a bug report?

See *Getting Help*.

## 4.2 Why did my script break when upgrading from lmfit 0.8.3 to 0.9.0?

See *Version 0.9.0 Release Notes*.

## 4.3 I get import errors from IPython

If you see something like:

```
from IPython.html.widgets import Dropdown

ImportError: No module named 'widgets'
```

then you need to install the `ipywidgets` package, try: `pip install ipywidgets`.

## 4.4 How can I fit multi-dimensional data?

The fitting routines accept data arrays that are one-dimensional and double precision. So you need to convert the data and model (or the value returned by the objective function) to be one-dimensional. A simple way to do this is to use numpy.ndarray.flatten, for example:

```
def residual(params, x, data=None):
    ....
    resid = calculate_multidim_residual()
    return resid.flatten()
```

## 4.5 How can I fit multiple data sets?

As above, the fitting routines accept data arrays that are one-dimensional and double precision. So you need to convert the sets of data and models (or the value returned by the objective function) to be one-dimensional. A simple way to do this is to use numpy.concatenate. As an example, here is a residual function to simultaneously fit two lines to two different arrays. As a bonus, the two lines share the 'offset' parameter:

```
import numpy as np


def fit_function(params, x=None, dat1=None, dat2=None):
    model1 = params['offset'] + x * params['slope1']
    model2 = params['offset'] + x * params['slope2']

    resid1 = dat1 - model1
    resid2 = dat2 - model2
    return np.concatenate((resid1, resid2))
```

## 4.6 How can I fit complex data?

As with working with multi-dimensional data, you need to convert your data and model (or the value returned by the objective function) to be double precision, floating point numbers. The simplest approach is to use numpy.ndarray.view, perhaps like:

```
import numpy as np


def residual(params, x, data=None):
    ....
    resid = calculate_complex_residual()
    return resid.view(float)
```

Alternately, you can use the `lmfit.Model` class to wrap a fit function that returns a complex vector. It will automatically apply the above prescription when calculating the residual. The benefit to this method is that you also get access to the plot routines from the ModelResult class, which are also complex-aware.

## 4.7 How should I cite LMFIT?

See https://dx.doi.org/10.5281/zenodo.11813

## 4.8 I get errors from NaN in my fit. What can I do?

The solvers used by lmfit use NaN (see https://en.wikipedia.org/wiki/NaN) values as signals that the calculation cannot continue. If any value in the residual array (typically `(data-model)*weight`) is NaN, then calculations of chi-square or comparisons with other residual arrays to try find a better fit will also give NaN and fail. There is no sensible way for lmfit or any of the optimization routines to know how to handle such NaN values. They indicate that numerical calculations are not sensible and must stop.

This means that if your objective function (if using `minimize`) or model function (if using `Model`) generates a NaN, the fit will stop immediately. If your objective or model function generates a NaN, you really must handle that.

### 4.8.1 `nan_policy`

If you are using `lmfit.Model` and the NaN values come from your data array and are meant to indicate missing values, or if you using `lmfit.minimize()` with the same basic intention, then it might be possible to get a successful fit in spite of the NaN values. To do this, you can add a `nan_policy='omit'` argument to `lmfit.minimize()`, or when creating a `lmfit.Model`, or when running `lmfit.Model.fit()`.

In order for this to be effective, the number of NaN values cannot ever change during the fit. If the NaN values come from the data and not the calculated model, that should be the case.

### 4.8.2 Common sources of NaN

If you are seeing errors due to NaN values, you will need to figure out where they are coming from and eliminate them. It is sometimes difficult to tell what causes NaN values. Keep in mind that all values should be assumed to be either scalar values or numpy arrays of double precision real numbers when fitting. Some of the most likely causes of NaNs are:

- taking `sqrt(x)` or `log(x)` where `x` is negative.
- doing `x**y` where `x` is negative. Since `y` is real, there will be a fractional component, and a negative number to a fractional exponent is not a real number.
- doing `x/y` where both `x` and `y` are 0.

If you use these very common constructs in your objective or model function, you should take some caution for what values you are passing these functions and operators. Many special functions have similar limitations and should also be viewed with some suspicion if NaNs are being generated.

A related problem is the generation of Inf (Infinity in floating point), which generally comes from `exp(x)` where `x` has values greater than 700 or so, so that the resulting value is greater than 1.e308. Inf is only slightly better than NaN. It will completely ruin the ability to do the fit. However, unlike NaN, it is also usually clear how to handle Inf, as you probably won't ever have values greater than 1.e308 and can therefore (usually) safely clip the argument passed to `exp()` to be smaller than about 700.

## 4.9 Why are Parameter values sometimes stuck at initial values?

In order for a Parameter to be optimized in a fit, changing its value must have an impact on the fit residual (`data-model` when curve fitting, for example). If a fit has not changed one or more of the Parameters, it means that changing those Parameters did not change the fit residual.

Normally (that is, unless you specifically provide a function for calculating the derivatives, in which case you probably would not be asking this question ;)), the fitting process begins by making a very small change to each Parameter value to determine which way and how large of a change to make for the parameter: This is the derivative or Jacobian (change in residual per change in parameter value). By default, the change made for each variable Parameter is to multiply its value by (1.0+1.0e-8) or so (unless the value is below about 1.e-15, in which case it adds 1.0e-8). If that small change does not change the residual, then the value of the Parameter will not be updated.

Parameter values that are "way off" are a common reason for Parameters being stuck at initial values. As an example, imagine fitting peak-like data with and `x` range of 0 to 10, peak centered at 6, and a width of 1 or 2 or so, as in the example at *doc_model_gaussian.py*. A Gaussian function with an initial value of for the peak center at 5 and an initial width or 5 will almost certainly find a good fit. An initial value of the peak center of -50 will end up being stuck with a "bad fit" because a small change in Parameters will still lead the modeled Gaussian to have no intensity over the actual range of the data. You should make sure that initial values for Parameters are reasonable enough to actually effect the fit. As it turns out in the example linked to above, changing the center value to any value between about 0 and 10 (that is, the data range) will result to a good fit.

Another common cause for Parameters being stuck at initial values is when the initial value is at a boundary value. For this case, too, a small change in the initial value for the Parameter will still leave the value at the boundary value and not show any real change in the residual.

If you're using bounds, make sure the initial values for the Parameters are not at the boundary values.

Finally, one reason for a Parameter to not change is that they are actually used as discrete values. This is discussed below in *Can Parameters be used for Array Indices or Discrete Values?*.

## 4.10 Why are uncertainties in Parameters sometimes not determined?

In order for Parameter uncertainties to be estimated, each variable Parameter must actually change the fit, and cannot be stuck at an initial value or at a boundary value. See *Why are Parameter values sometimes stuck at initial values?* for why values may not change from their initial values.

## 4.11 Can Parameters be used for Array Indices or Discrete Values?

The short answer is "No": variables in all of the fitting methods used in `lmfit` (and all of those available in `scipy.optimize`) are treated as continuous values, and represented as double precision floating point values. As an important example, you cannot have a variable that is somehow constrained to be an integer.

Still, it is a rather common question of how to fit data to a model that includes a breakpoint, perhaps

$$f(x; x_0, a, b, c) = \begin{cases} c & \text{for } x < x_0 \\ a + bx^2 & \text{for } x > x_0 \end{cases}$$

That you implement with a model function and use to fit data like this:

```python
import numpy as np

import lmfit


def quad_off(x, x0, a, b, c):
    model = a + b * x**2
    model[np.where(x < x0)] = c
    return model


x0 = 19
b = 0.02
a = 2.0
xdat = np.linspace(0, 100, 101)
ydat = a + b * xdat**2
ydat[np.where(xdat < x0)] = a + b * x0**2
ydat += np.random.normal(scale=0.1, size=xdat.size)

mod = lmfit.Model(quad_off)
pars = mod.make_params(x0=22, a=1, b=1, c=1)
```

(continues on next page)

(continued from previous page)

```
result = mod.fit(ydat, pars, x=xdat)
print(result.fit_report())
```

```
[[Model]]
    Model(quad_off)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 11
    # data points      = 101
    # variables        = 4
    chi-square         = 8.56937557
    reduced chi-square = 0.08834408
    Akaike info crit   = -241.159491
    Bayesian info crit = -230.699009
##  Warning: uncertainties could not be estimated:
    x0:  at initial value
[[Variables]]
    x0:  22.0000000 (init = 22)
    a:   2.01362058 (init = 1)
    b:   0.01999551 (init = 1)
    c:   9.40565127 (init = 1)
```

This will not result in a very good fit, as the value for `x0` cannot be found by making a small change in its value. Specifically, `model[np.where(x < x0)]` will give the same result for `x0=22` and `x0=22.001`, and so that value is not changed during the fit.

There are a couple ways around this problem. First, you may be able to make the fit depend on `x0` in a way that is not just discrete. That depends on your model function. A second option is to treat the break not as a hard break but as a more gentle transition with a sigmoidal function, such as an error function. Like the break-point, these will go from 0 to 1, but more gently and with some finite value leaking into neighboring points. The amount of leakage or width of the step can also be adjusted.

A simple modification of the above to use an error function would look like this and give better fit results:

```
import numpy as np
from scipy.special import erf

import lmfit


def quad_off(x, x0, a, b, c):
    m1 = a + b * x**2
    m2 = c * np.ones(len(x))
    # step up from 0 to 1 at x0: (erf(x-x0)+1)/2
    # step down from 1 to 0 at x0: (1-erf(x-x0))/2
    model = m1 * (erf(x-x0)+1)/2 + m2 * (1-erf(x-x0))/2
    return model


x0 = 19
b = 0.02
a = 2.0
xdat = np.linspace(0, 100, 101)
ydat = a + b * xdat**2
ydat[np.where(xdat < x0)] = a + b * x0**2
ydat += np.random.normal(scale=0.1, size=xdat.size)
```

(continues on next page)

(continued from previous page)

```
mod = lmfit.Model(quad_off)
pars = mod.make_params(x0=22, a=1, b=1, c=1)

result = mod.fit(ydat, pars, x=xdat)
print(result.fit_report())
```

```
[[Model]]
    Model(quad_off)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 46
    # data points      = 101
    # variables        = 4
    chi-square         = 0.95985842
    reduced chi-square = 0.00989545
    Akaike info crit   = -462.265090
    Bayesian info crit = -451.804608
[[Variables]]
    x0:  19.1417821 +/- 0.42122000 (2.20%) (init = 22)
    a:   2.02699936 +/- 0.01979431 (0.98%) (init = 1)
    b:   0.01999431 +/- 3.9152e-06 (0.02%) (init = 1)
    c:   9.25101120 +/- 0.02277975 (0.25%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(a, b)  = -0.828
    C(x0, a) =  0.160
    C(x0, b) = -0.128
```

The natural width of the error function is about 2 `x` units, but you can adjust this, shortening it with `erf((x-x0)*2)` to give a sharper transition for example.

CHAPTER

FIVE

# PARAMETER AND PARAMETERS

This chapter describes the `Parameter` object, which is a key concept of lmfit.

A `Parameter` is the quantity to be optimized in all minimization problems, replacing the plain floating point number used in the optimization routines from `scipy.optimize`. A `Parameter` has a value that can either be varied in the fit or held at a fixed value, and can have lower and/or upper bounds placed on the value. It can even have a value that is constrained by an algebraic expression of other Parameter values. Since `Parameter` objects 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, or whether bounds can be applied. This simplifies the writing of models, allowing general models that describe the phenomenon and gives the user more flexibility in using and testing variations of that model.

Whereas a `Parameter` expands on an individual floating point variable, the optimization methods actually still need an ordered group of floating point variables. In the `scipy.optimize` routines this is required to be a one-dimensional numpy.ndarray. In lmfit, this one-dimensional array is replaced by a `Parameters` object, which works as an ordered dictionary of `Parameter` objects with a few additional features and methods. That is, while the concept of a `Parameter` is central to lmfit, one normally creates and interacts with a `Parameters` instance that contains many `Parameter` objects. For example, the objective functions you write for lmfit will take an instance of `Parameters` as its first argument. A table of parameter values, bounds, and other attributes can be printed using `Parameters.pretty_print()`.

## 5.1 The `Parameter` class

**class Parameter**(*name*, *value=None*, *vary=True*, *min=- inf*, *max=inf*, *expr=None*, *brute_step=None*, *user_data=None*)

A Parameter is an object that can be varied in a fit.

It is a central component of lmfit, and all minimization and modeling methods use Parameter objects.

A Parameter has a *name* attribute, and a scalar floating point *value*. It also has a *vary* attribute that describes whether the value should be varied during the minimization. Finite bounds can be placed on the Parameter's value by setting its *min* and/or *max* attributes. A Parameter can also have its value determined by a mathematical expression of other Parameter values held in the *expr* attribute. Additional attributes include *brute_step* used as the step size in a brute-force minimization, and *user_data* reserved exclusively for user's need.

After a minimization, a Parameter may also gain other attributes, including *stderr* holding the estimated standard error in the Parameter's value, and *correl*, a dictionary of correlation values with other Parameters used in the minimization.

**Parameters**

- **name** (*str*) – Name of the Parameter.
- **value** (*float, optional*) – Numerical Parameter value.

- **`vary`** (*`bool`*, *`optional`*) – Whether the Parameter is varied during a fit (default is True).
- **`min`** (*`float`*, *`optional`*) – Lower bound for value (default is `-numpy.inf`, no lower bound).
- **`max`** (*`float`*, *`optional`*) – Upper bound for value (default is `numpy.inf`, no upper bound).
- **`expr`** (*`str`*, *`optional`*) – Mathematical expression used to constrain the value during the fit (default is None).
- **`brute_step`** (*`float`*, *`optional`*) – Step size for grid points in the *brute* method (default is None).
- **`user_data`** (*`optional`*) – User-definable extra attribute used for a Parameter (default is None).

**`stderr`**

The estimated standard error for the best-fit value.

**Type** float

**`correl`**

A dictionary of the correlation with the other fitted Parameters of the form:

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

**Type** dict

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*, *brute_step=None*)

Set or update Parameter attributes.

**Parameters**

- **`value`** (*`float`*, *`optional`*) – Numerical Parameter value.
- **`vary`** (*`bool`*, *`optional`*) – Whether the Parameter is varied during a fit.
- **`min`** (*`float`*, *`optional`*) – Lower bound for value. To remove a lower bound you must use `-numpy.inf`.
- **`max`** (*`float`*, *`optional`*) – Upper bound for value. To remove an upper bound you must use `numpy.inf`.
- **`expr`** (*`str`*, *`optional`*) – Mathematical expression used to constrain the value during the fit. To remove a constraint you must supply an empty string.
- **`brute_step`** (*`float`*, *`optional`*) – Step size for grid points in the *brute* method. To remove the step size you must use `0`.

### Notes

Each argument to *set()* has a default value of None, which will leave the current value for the attribute unchanged. Thus, to lift a lower or upper bound, passing in None will not work. Instead, you must set these to `-numpy.inf` or `numpy.inf`, as with:

```
par.set(min=None)          # leaves lower bound unchanged
par.set(min=-numpy.inf)    # removes lower bound
```

Similarly, to clear an expression, pass a blank string, (not None!) as with:

```
par.set(expr=None)   # leaves expression unchanged
par.set(expr='')     # removes expression
```

Explicitly setting a value or setting `vary=True` will also clear the expression.

Finally, to clear the brute_step size, pass `0`, not None:

```
par.set(brute_step=None)   # leaves brute_step unchanged
par.set(brute_step=0)      # removes brute_step
```

## 5.2 The `Parameters` class

**class Parameters**(*asteval=None*, *usersyms=None*)

An ordered dictionary of Parameter objects.

It should contain all Parameter objects that are required to specify a fit model. All minimization and Model fitting routines in lmfit will use exactly one Parameters object, typically given as the first argument to the objective function.

All keys of a Parameters() instance must be strings and valid Python symbol names, so that the name must match `[a-z_][a-z0-9_]*` and cannot be a Python reserved word.

All values of a Parameters() instance must be Parameter objects.

A Parameters() instance includes an *asteval* Interpreter used for evaluation of constrained Parameters.

Parameters() support copying and pickling, and have methods to convert to and from serializations using json strings.

**Parameters**

- **asteval** (`asteval.Interpreter`, optional) – Instance of the *asteval.Interpreter* to use for constraint expressions. If None (default), a new interpreter will be created. **Warning: deprecated**, use *usersyms* if possible!
- **usersyms** (*dict, optional*) – Dictionary of symbols to add to the `asteval.Interpreter` (default is None).

**add**(*name*, *value=None*, *vary=True*, *min=- inf*, *max=inf*, *expr=None*, *brute_step=None*)

Add a Parameter.

**Parameters**

- **name** (*str*) – Name of parameter. Must match `[a-z_][a-z0-9_]*` and cannot be a Python reserved word.
- **value** (*float, optional*) – Numerical Parameter value, typically the *initial value*.

- **vary** (*bool, optional*) – Whether the Parameter is varied during a fit (default is True).
- **min** (*float, optional*) – Lower bound for value (default is -numpy.inf, no lower bound).
- **max** (*float, optional*) – Upper bound for value (default is numpy.inf, no upper bound).
- **expr** (*str, optional*) – Mathematical expression used to constrain the value during the fit (default is None).
- **brute_step** (*float, optional*) – Step size for grid points in the *brute* method (default is None).

### Examples

```
>>> params = Parameters()
>>> params.add('xvar', value=0.50, min=0, max=1)
>>> params.add('yvar', expr='1.0 - xvar')
```

which is equivalent to:

```
>>> params = Parameters()
>>> params['xvar'] = Parameter(name='xvar', value=0.50, min=0, max=1)
>>> params['yvar'] = Parameter(name='yvar', expr='1.0 - xvar')
```

**add_many** (**parlist*)
Add many parameters, using a sequence of tuples.

**Parameters** ***parlist** (sequence of tuple or Parameter) – A sequence of tuples, or a sequence of *Parameter* instances. If it is a sequence of tuples, then each tuple must contain at least a *name*. The order in each tuple must be (name, value, vary, min, max, expr, brute_step).

### Examples

```
>>>  params = Parameters()
# add with tuples: (NAME VALUE VARY MIN  MAX  EXPR  BRUTE_STEP)
>>> params.add_many(('amp', 10, True, None, None, None, None),
...                 ('cen', 4, True, 0.0, None, None, None),
...                 ('wid', 1, False, None, None, None, None),
...                 ('frac', 0.5))
# add a sequence of Parameters
>>> f = Parameter('par_f', 100)
>>> g = Parameter('par_g', 2.)
>>> params.add_many(f, g)
```

**pretty_print** (*oneline=False, colwidth=8, precision=4, fmt='g', columns=['value', 'min', 'max', 'stderr', 'vary', 'expr', 'brute_step']*)
Pretty-print of parameters data.

**Parameters**

- **oneline** (*bool, optional*) – If True prints a one-line parameters representation (default is False).

- **colwidth** (*int, optional*) – Column width for all columns specified in *columns* (default is 8).
- **precision** (*int, optional*) – Number of digits to be printed after floating point (default is 4).
- **fmt** (*{'g', 'e', 'f'}, optional*) – Single-character numeric formatter. Valid values are: *'g'* floating point and exponential (default), *'e'* exponential, or *'f'* floating point.
- **columns** (list of str, optional) – List of *Parameter* attribute names to print (default is to show all attributes).

**valuesdict**()

Return an ordered dictionary of parameter values.

**Returns** An ordered dictionary of `name`:`value` pairs for each Parameter.

**Return type** OrderedDict

**dumps**(***kws*)

Represent Parameters as a JSON string.

**Parameters** ****kws** (*optional*) – Keyword arguments that are passed to *json.dumps*.

**Returns** JSON string representation of Parameters.

**Return type** str

**See also:**

*dump*, *loads*, *load*, `json.dumps`

**dump**(*fp*, ***kws*)

Write JSON representation of Parameters to a file-like object.

**Parameters**

- **fp** (*file-like object*) – An open and *.write()*-supporting file-like object.
- ****kws** (*optional*) – Keyword arguments that are passed to *dumps*.

**Returns** Return value from *fp.write()*: the number of characters written.

**Return type** int

**See also:**

*dumps*, *load*, `json.dump`

**eval**(*expr*)

Evaluate a statement using the *asteval* Interpreter.

**Parameters** **expr** (*str*) – An expression containing parameter names and other symbols recognizable by the *asteval* Interpreter.

**Returns** The result of evaluating the expression.

**Return type** float

**loads**(*s*, ***kws*)

Load Parameters from a JSON string.

**Parameters** ****kws** (*optional*) – Keyword arguments that are passed to *json.loads*.

**Returns** Updated Parameters from the JSON string.

**Return type** *Parameters*

### Notes

Current Parameters will be cleared before loading the data from the JSON string.

**See also:**

*dump*, *dumps*, *load*, `json.loads`

**load**(*fp*, ***kws*)
Load JSON representation of Parameters from a file-like object.

**Parameters**

- **fp** (*file-like object*) – An open and *.read()*-supporting file-like object.
- ****kws** (*optional*) – Keyword arguments that are passed to *loads*.

**Returns** Updated Parameters loaded from *fp*.

**Return type** *Parameters*

**See also:**

*dump*, *loads*, `json.load`

## 5.3 Simple Example

A basic example making use of `Parameters` and the `minimize()` function (discussed in the next chapter) might look like this:

```
# <examples/doc_parameters_basic.py>
import numpy as np

from lmfit import Minimizer, Parameters, report_fit

# create data to be fitted
x = np.linspace(0, 15, 301)
data = (5.0 * np.sin(2.0*x - 0.1) * np.exp(-x*x*0.025) +
        np.random.normal(size=x.size, scale=0.2))


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

(continues on next page)

(continued from previous page)

```python
# do fit, here with the default leastsq algorithm
minner = Minimizer(fcn2min, params, fcn_args=(x, data))
result = minner.minimize()

# calculate final result
final = data + result.residual

# write error report
report_fit(result)

# try to plot results
try:
    import matplotlib.pyplot as plt
    plt.plot(x, data, 'k+')
    plt.plot(x, final, 'r')
    plt.show()
except ImportError:
    pass
# <end of examples/doc_parameters_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:

```python
def fcn2min(params, x, data):
    """Model a decaying sine wave and 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 stylistic choice.

CHAPTER

SIX

# PERFORMING FITS AND 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 or comparing results from related fits.

## 6.1 The `minimize()` function

The `minimize()` function is a wrapper around `Minimizer` for running an optimization problem. It takes an objective function (the function that calculates the array to be minimized), a `Parameters` object, and several optional arguments. See *Writing a Fitting Function* for details on writing the objective function.

**minimize**(*fcn*, *params*, *method='leastsq'*, *args=None*, *kws=None*, *iter_cb=None*, *scale_covar=True*, *nan_policy='raise'*, *reduce_fcn=None*, *calc_covar=True*, *max_nfev=None*, ***fit_kws*)

Perform the minimization of the objective function.

The minimize function takes an objective function to be minimized, a dictionary (`Parameters` ; Parameters) containing the model parameters, and several optional arguments including the fitting method.

**Parameters**

- **fcn** (`callable`) – Objective function to be minimized. When method is *'leastsq'* or '*least_squares*', the objective function should return an array of residuals (difference between model and data) to be minimized in a least-squares sense. With the scalar methods the objective function can either return the residuals array or a single scalar value. The function must have the signature:

```
fcn(params, *args, **kws)
```

- **params** (`Parameters`) – Contains the Parameters for the model.
- **method** (`str, optional`) – Name of the fitting method to use. Valid values are:
  - *'leastsq'*: Levenberg-Marquardt (default)
  - *'least_squares'*: Least-Squares minimization, using Trust Region Reflective method
  - *'differential_evolution'*: differential evolution
  - *'brute'*: brute force method
  - *'basinhopping'*: basinhopping
  - *'ampgo'*: Adaptive Memory Programming for Global Optimization
  - '*nelder*': Nelder-Mead
  - *'lbfgsb'*: L-BFGS-B

  - *'powell'*: Powell
  - *'cg'*: Conjugate-Gradient
  - *'newton'*: Newton-CG
  - *'cobyla'*: Cobyla
  - *'bfgs'*: BFGS
  - *'tnc'*: Truncated Newton
  - *'trust-ncg'*: Newton-CG trust-region
  - *'trust-exact'*: nearly exact trust-region
  - *'trust-krylov'*: Newton GLTR trust-region
  - *'trust-constr'*: trust-region for constrained optimization
  - *'dogleg'*: Dog-leg trust-region
  - *'slsqp'*: Sequential Linear Squares Programming
  - *'emcee'*: Maximum likelihood via Monte-Carlo Markov Chain
  - *'shgo'*: Simplicial Homology Global Optimization
  - *'dual_annealing'*: Dual Annealing optimization

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

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

- **args** (*tuple, optional*) – Positional arguments to pass to *fcn*.
- **kws** (*dict, optional*) – Keyword arguments to pass to *fcn*.
- **iter_cb** (*callable, optional*) – Function to be called at each fit iteration. This function should have the signature:

  ```
  iter_cb(params, iter, resid, *args, **kws),
  ```

  where *params* will have the current parameter values, *iter* the iteration number, *resid* the current residual array, and **args* and ***kws* as passed to the objective function.

- **scale_covar** (*bool, optional*) – Whether to automatically scale the covariance matrix (default is True).
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – Specifies action if *fcn* (or a Jacobian) returns NaN values. One of:
  - *'raise'* : a *ValueError* is raised
  - *'propagate'* : the values returned from *userfcn* are un-altered
  - *'omit'* : non-finite values are filtered
- **reduce_fcn** (*str or callable, optional*) – Function to convert a residual array to a scalar value for the scalar minimizers. See Notes in *Minimizer*.
- **calc_covar** (*bool, optional*) – Whether to calculate the covariance matrix (default is True) for solvers other than *'leastsq'* and *'least_squares'*. Requires the *numdifftools* package to be installed.

- **max_nfev** (*int or None, optional*) – Maximum number of function evaluations (default is None). The default value depends on the fitting method.
- ****fit_kws** (*dict, optional*) – Options to pass to the minimizer being used.

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

**Return type** *MinimizerResult*

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

### Notes

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.

A common use for *args* and *kws* 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.

On output, *params* will be unchanged. The best-fit values and, where appropriate, estimated uncertainties and correlations, will all be contained in the returned `MinimizerResult`. See *MinimizerResult – the optimization result* for further details.

This function is simply a wrapper around `Minimizer` and is equivalent to:

```
fitter = Minimizer(fcn, params, fcn_args=args, fcn_kws=kws,
                   iter_cb=iter_cb, scale_covar=scale_covar,
                   nan_policy=nan_policy, reduce_fcn=reduce_fcn,
                   calc_covar=calc_covar, **fit_kws)
fitter.minimize(method=method)
```

## 6.2 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** (`Parameters`) – 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.ndarray. 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 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 shown below:

```
from numpy import exp, sign, sin, pi


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 bounds on the `Parameter` to do this:

```
params['period'] = Parameter(name='period', 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`.

## 6.3 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 *MinimizerResult – the optimization result*.

Alternative algorithms can also be used by providing the `method` keyword to the *`minimize()`* function or *`Minimizer.minimize()`* class as listed in the *Table of Supported Fitting Methods*. If you have the `numdifftools` package installed, lmfit will try to estimate the covariance matrix and determine parameter uncertainties and correlations if `calc_covar` is `True` (default).

Table of Supported Fitting Methods:

| Fitting Method | `method` arg to *`minimize()`* or *`Minimizer.minimize()`* |
|---|---|
| Levenberg-Marquardt | `leastsq` or `least_squares` |
| Nelder-Mead | `nelder` |
| L-BFGS-B | `lbfgsb` |
| Powell | `powell` |
| Conjugate Gradient | `cg` |
| Newton-CG | `newton` |
| COBYLA | `cobyla` |
| BFGS | `bfgsb` |
| Truncated Newton | `tnc` |
| Newton CG trust-region | `trust-ncg` |
| Exact trust-region | `trust-exact` |
| Newton GLTR trust-region | `trust-krylov` |
| Constrained trust-region | `trust-constr` |
| Dogleg | `dogleg` |
| Sequential Linear Squares Programming | `slsqp` |
| Differential Evolution | `differential_evolution` |
| Brute force method | `brute` |
| Basinhopping | `basinhopping` |
| Adaptive Memory Programming for Global Optimization | `ampgo` |
| Simplicial Homology Global Ooptimization | `shgo` |
| Dual Annealing | `dual_annealing` |
| Maximum likelihood via Monte-Carlo Markov Chain | `emcee` |

**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. The Monte-Carlo Markov Chain or `emcee` method has two different operating methods when the objective function returns a scalar value. See the documentation for `emcee`.

**Warning:** Much of this documentation assumes that the Levenberg-Marquardt (`leastsq`) method is used. Many of the fit statistics and estimates for uncertainties in parameters discussed in *MinimizerResult – the optimization result* are done only unconditionally for this (and the `least_squares`) method. Lmfit versions newer than

0.9.11 provide the capability to use `numdifftools` to estimate the covariance matrix and calculate parameter uncertainties and correlations for other methods as well.

## 6.4 `MinimizerResult` – the optimization result

New in version 0.9.0.

An optimization with *`minimize()`* or *`Minimizer.minimize()`* will return a *`MinimizerResult`* object. This is an otherwise plain container object (that is, with no methods of its own) that simply holds the results of the minimization. These results will include several pieces of informational data such as status and error messages, fit statistics, and the updated parameters themselves.

Importantly, the parameters passed in to *`Minimizer.minimize()`* will be not be changed. To find the best-fit values, uncertainties and so on for each parameter, one must use the *`MinimizerResult.params`* attribute. For example, to print the fitted values, bounds and other parameter attributes in a well-formatted text tables you can execute:

```
result.params.pretty_print()
```

with `results` being a `MinimizerResult` object. Note that the method *`pretty_print()`* accepts several arguments for customizing the output (e.g., column width, numeric format, etcetera).

**class MinimizerResult**(***kws*)

The results of a minimization.

Minimization results include data such as status and error messages, fit statistics, and the updated (i.e., best-fit) parameters themselves in the *`params`* attribute.

The list of (possible) *MinimizerResult* attributes is given below:

**params**

The best-fit parameters resulting from the fit.

**Type** *Parameters*

**status**

Termination status of the optimizer. Its value depends on the underlying solver. Refer to *message* for details.

**Type** int

**var_names**

Ordered list of variable parameter names used in optimization, and useful for understanding the values in *`init_vals`* and *`covar`*.

**Type** list

**covar**

Covariance matrix from minimization, with rows and columns corresponding to *`var_names`*.

**Type** numpy.ndarray

**init_vals**

List of initial values for variable parameters using *`var_names`*.

**Type** list

**init_values**

Dictionary of initial values for variable parameters.

**Type** dict

**nfev**
Number of function evaluations.

**Type** int

**success**
True if the fit succeeded, otherwise False.

**Type** bool

**errorbars**
True if uncertainties were estimated, otherwise False.

**Type** bool

**message**
Message about fit success.

**Type** str

**call_kws**
Keyword arguments sent to underlying solver.

**Type** dict

**ier**
Integer error value from scipy.optimize.leastsq (*'leastsq'* method only).

**Type** int

**lmdif_message**
Message from scipy.optimize.leastsq (*'leastsq'* method only).

**Type** str

**nvarys**
Number of variables in fit: $N_{\rm varys}$.

**Type** int

**ndata**
Number of data points: $N$.

**Type** int

**nfree**
Degrees of freedom in fit: $N - N_{\rm varys}$.

**Type** int

**residual**
Residual array $\rm Resid_i$. Return value of the objective function when using the best-fit values of the parameters.

**Type** numpy.ndarray

**chisqr**
Chi-square: $\chi^2 = \sum_i^N [{\rm Resid}_i]^2$.

**Type** float

**redchi**
Reduced chi-square: $\chi^2_\nu = \chi^2/(N - N_{\rm varys})$.

**Type** float

**aic**
Akaike Information Criterion statistic: $N \ln(\chi^2/N) + 2N_{\rm varys}$.

**Type** float

**bic**
Bayesian Information Criterion statistic: $N \ln(\chi^2/N) + \ln(N) N_{\rm varys}$.

**Type** float

**flatchain**
A flatchain view of the sampling chain from the *emcee* method.

**Type** pandas.DataFrame

**show_candidates**()
`pretty_print()` representation of candidates from the *brute* fitting method.

### 6.4.1 Goodness-of-Fit Statistics

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

| Attribute Name | Description / Formula |
|---|---|
| nfev | number of function evaluations |
| 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, returned by the objective function: $\{\rm Resid_i\}$ |
| chisqr | chi-square: $\chi^2 = \sum_i^N [{\rm Resid}_i]^2$ |
| redchi | reduced chi-square: $\chi^2_{\nu} = \chi^2/(N - N_{\rm varys})$ |
| aic | Akaike Information Criterion statistic (see below) |
| bic | Bayesian Information Criterion statistic (see below) |
| var_names | ordered list of variable parameter names used for init_vals and covar |
| covar | covariance matrix (with rows/columns using var_names) |
| init_vals | list of initial values for variable parameters |
| call_kws | dict of keyword arguments sent to underlying solver |

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 the `leastsq()` or `least_squares()` method has completed successfully, standard errors for the fitted variables and correlations between pairs of fitted variables are automatically calculated from the covariance matrix. For other methods, the `calc_covar` parameter (default is `True`) in the *Minimizer* class determines whether or not to use the `numdifftools` package to estimate the covariance matrix. The standard error (estimated $1\sigma$ error-bar) goes 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`.

### 6.4.2 Akaike and Bayesian Information Criteria

The *MinimizerResult* includes the traditional chi-square and reduced chi-square statistics:

$$\begin{aligned}\chi^2 &= \sum_i^N r_i^2 \\ \chi^2_\nu &= \chi^2/(N - N_{\rm varys})\end{aligned}$$

where $r$ is the residual array returned by the objective function (likely to be `(data-model)/uncertainty` for data modeling usages), $N$ is the number of data points (`ndata`), and $N_{\rm varys}$ is number of variable parameters.

Also included are the Akaike Information Criterion, and Bayesian Information Criterion statistics, held in the `aic` and `bic` attributes, respectively. These give slightly different measures of the relative quality for a fit, trying to balance quality of fit with the number of variable parameters used in the fit. These are calculated as:

$$\begin{aligned}{\rm aic} &= N \ln(\chi^2/N) + 2 N_{\rm varys} \\ {\rm bic} &= N \ln(\chi^2/N) + \ln(N) N_{\rm varys}\end{aligned}$$

When comparing fits with different numbers of varying parameters, one typically selects the model with lowest reduced chi-square, Akaike information criterion, and/or Bayesian information criterion. Generally, the Bayesian information criterion is considered the most conservative of these statistics.

### 6.4.3 Uncertainties in Variable Parameters, and their Correlations

As mentioned above, when a fit is complete the uncertainties for fitted Parameters as well as the correlations between pairs of Parameters are usually calculated. This happens automatically either when using the default `leastsq()` method, the `least_squares()` method, or for most other fitting methods if the highly-recommended `numdifftools` package is available. The estimated standard error (the $1\sigma$ uncertainty) for each variable Parameter will be contained in the `stderr`, while the `correl` attribute for each Parameter will contain a dictionary of the correlation with each other variable Parameter.

These estimates of the uncertainties are done by inverting the Hessian matrix which represents the second derivative of fit quality for each variable parameter. There are situations for which the uncertainties cannot be estimated, which generally indicates that this matrix cannot be inverted because one of the fit is not actually sensitive to one of the variables. This can happen if a Parameter is stuck at an upper or lower bound, if the variable is simply not used by the fit, or if the value for the variable is such that it has no real influence on the fit.

In principle, the scale of the uncertainties in the Parameters is closely tied to the goodness-of-fit statistics chi-square and reduced chi-square (`chisqr` and `redchi`). The standard errors or $1\sigma$ uncertainties are those that increase chi-square by 1. Since a "good fit" should have `redchi` of around 1, this requires that the data uncertainties (and to some extent the sampling of the N data points) is correct. Unfortunately, it is often not the case that one has high-quality estimates of the data uncertainties (getting the data is hard enough!). Because of this common situation, the uncertainties reported and held in `stderr` are not those that increase chi-square by 1, but those that increase chi-square by reduced chi-square. This is equivalent to rescaling the uncertainty in the data such that reduced chi-square would be 1. To be clear, this rescaling is done by default because if reduced chi-square is far from 1, this rescaling often makes the reported uncertainties sensible, and if reduced chi-square is near 1 it does little harm. If you have good scaling of the data uncertainty and believe the scale of the residual array is correct, this automatic rescaling can be turned off using `scale_covar=False`.

Note that the simple (and fast!) approach to estimating uncertainties and correlations by inverting the second derivative matrix assumes that the components of the residual array (if, indeed, an array is used) are distributed around 0 with a normal (Gaussian distribution), and that a map of probability distributions for pairs would be elliptical – the size of the of ellipse gives the uncertainty itself and the eccentricity of the ellipse gives the correlation. This simple approach to assessing uncertainties ignores outliers, highly asymmetric uncertainties, or complex correlations between Parameters.

In fact, it is not too hard to come up with problems where such effects are important. Our experience is that the automated results are usually the right scale and quite reasonable as initial estimates, but a more thorough exploration of the Parameter space using the tools described in *Minimizer.emcee() - calculating the posterior probability distribution of parameters* and *An advanced example for evaluating confidence intervals* can give a more complete understanding of the distributions and relations between Parameters.

## 6.5 Getting and Printing Fit Reports

**fit_report**(*inpars*, *modelpars=None*, *show_correl=True*, *min_correl=0.1*, *sort_pars=False*)

Generate a report of the fitting results.

The report contains the best-fit values for the parameters and their uncertainties and correlations.

**Parameters**

- **inpars** (`Parameters`) – Input Parameters from fit or MinimizerResult returned from a fit.
- **modelpars** (`Parameters, optional`) – Known Model Parameters.
- **show_correl** (*bool, optional*) – Whether to show list of sorted correlations (default is True).
- **min_correl** (*float, optional*) – Smallest correlation in absolute value to show (default is 0.1).
- **sort_pars** (*bool or callable, optional*) – Whether to show parameter names sorted in alphanumerical order. If False (default), then the parameters will be listed in the order they were added to the Parameters dictionary. If callable, then this (one argument) function is used to extract a comparison key from each list element.

**Returns** Multi-line text of fit report.

**Return type** str

An example using this to write out a fit report would be:

```
# <examples/doc_fitting_withreport.py>
from numpy import exp, linspace, pi, random, sign, sin

from lmfit import Parameters, fit_report, minimize

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):
    """Model a decaying sine wave and subtract data."""
    vals = pars.valuesdict()
    amp = vals['amp']
    per = vals['period']
    shift = vals['shift']
    decay = vals['decay']

    if abs(shift) > pi/2:
```

(continues on next page)

(continued from previous page)

```
        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


random.seed(0)
x = linspace(0.0, 250., 1001)
noise = random.normal(scale=0.7215, size=x.size)
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})

print(fit_report(out))
# <end examples/doc_fitting_withreport.py>
```

which would give as output:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 83
    # data points      = 1001
    # variables        = 4
    chi-square         = 498.811759
    reduced chi-square = 0.50031270
    Akaike info crit   = -689.222517
    Bayesian info crit = -669.587497
[[Variables]]
    amp:     13.9121945 +/- 0.14120288 (1.01%) (init = 13)
    period:  5.48507045 +/- 0.02666492 (0.49%) (init = 2)
    shift:   0.16203677 +/- 0.01405661 (8.67%) (init = 0)
    decay:   0.03264538 +/- 3.8014e-04 (1.16%) (init = 0.02)
[[Correlations]] (unreported correlations are < 0.100)
    C(period, shift) =  0.797
    C(amp, decay)    =  0.582
    C(amp, shift)    = -0.297
    C(amp, period)   = -0.243
    C(shift, decay)  = -0.182
    C(period, decay) = -0.150
```

To be clear, you can get at all of these values from the fit result `out` and `out.params`. For example, a crude printout of the best fit variables and standard errors could be done as

```
print('-------------------------------')
print('Parameter    Value       Stderr')
for name, param in out.params.items():
    print('{:7s} {:11.5f} {:11.5f}'.format(name, param.value, param.stderr))
```

```
-------------------------------
```

(continues on next page)

(continued from previous page)

```
Parameter    Value       Stderr
amp        13.91219     0.14120
period      5.48507     0.02666
shift       0.16204     0.01406
decay       0.03265     0.00038
```

## 6.6 Using a Iteration Callback Function

An iteration callback function is a function to be called at each iteration, just after the objective function is called. The iteration callback allows user-supplied code to be run at each iteration, and can be used to abort a fit.

**iter_cb(params, iter, resid, *args, **kws):**
User-supplied function to be run at each iteration.

**Parameters**

- **params** (*Parameters*) – Parameters.
- **iter** (*int*) – Iteration number.
- **resid** (*numpy.ndarray*) – Residual array.
- **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** None for normal behavior, any value like `True` to abort the fit.

Normally, the iteration callback would have no return value or return `None`. To abort a fit, have this function return a value that is `True` (including any non-zero integer). The fit will also abort if any exception is raised in the iteration callback. When a fit is aborted this way, the parameters will have the values from the last iteration. The fit statistics are not likely to be meaningful, and uncertainties will not be computed.

## 6.7 Using the `Minimizer` class

For full control of the fitting process, you will want to create a *Minimizer* object.

**class Minimizer**(*userfcn*, *params*, *fcn_args=None*, *fcn_kws=None*, *iter_cb=None*, *scale_covar=True*, *nan_policy='raise'*, *reduce_fcn=None*, *calc_covar=True*, *max_nfev=None*, ***kws*)
A general minimizer for curve fitting and optimization.

**Parameters**

- **userfcn** (*callable*) – Objective function that returns the residual (difference between model and data) to be minimized in a least-squares sense. This function must have the signature:

```
userfcn(params, *fcn_args, **fcn_kws)
```

- **params** (*Parameters*) – Contains the Parameters for the model.
- **fcn_args** (*tuple, optional*) – Positional arguments to pass to *userfcn*.
- **fcn_kws** (*dict, optional*) – Keyword arguments to pass to *userfcn*.

- **iter_cb** (*callable, optional*) – Function to be called at each fit iteration. This function should have the signature:

  ```
  iter_cb(params, iter, resid, *fcn_args, **fcn_kws)
  ```

  where *params* will have the current parameter values, *iter* the iteration number, *resid* the current residual array, and **fcn_args* and ***fcn_kws* are passed to the objective function.

- **scale_covar** (*bool, optional*) – Whether to automatically scale the covariance matrix (default is True).

- **nan_policy** (*{'raise', 'propagate', 'omit}, optional*) – Specifies action if *userfcn* (or a Jacobian) returns NaN values. One of:
  - *'raise'* : a *ValueError* is raised (default)
  - *'propagate'* : the values returned from *userfcn* are un-altered
  - *'omit'* : non-finite values are filtered

- **reduce_fcn** (*str or callable, optional*) – Function to convert a residual array to a scalar value for the scalar minimizers. Optional values are (where *r* is the residual array):
  - None : sum-of-squares of residual (default)

    = (r*r).sum()
  - *'negentropy'* : neg entropy, using normal distribution

    = rho*log(rho).sum()`, where rho = exp(-r*r/2)/(sqrt(2*pi))
  - *'neglogcauchy'* : neg log likelihood, using Cauchy distribution

    = -log(1/(pi*(1+r*r))).sum()
  - callable : must take one argument (*r*) and return a float.

- **calc_covar** (*bool, optional*) – Whether to calculate the covariance matrix (default is True) for solvers other than `'leastsq'` and `'least_squares'`. Requires the `numdifftools` package to be installed.

- **max_nfev** (*int or None, optional*) – Maximum number of function evaluations (default is None). The default value depends on the fitting method.

- ****kws** (*dict, optional*) – Options to pass to the minimizer being used.

### Notes

The objective function should return the value to be minimized. For the Levenberg-Marquardt algorithm from `leastsq()` or `least_squares()`, 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. If the objective function returns non-finite values then a *ValueError* will be raised because the underlying solvers cannot deal with them.

A common use for the *fcn_args* and *fcn_kws* 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 Minimizer object has a few public methods:

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

Perform the minimization.

**Parameters**

- **`method`** (`str, optional`) – Name of the fitting method to use. Valid values are:
  - *'leastsq'*: Levenberg-Marquardt (default)
  - *'least_squares'*: Least-Squares minimization, using Trust Region Reflective method
  - *'differential_evolution'*: differential evolution
  - *'brute'*: brute force method
  - *'basinhopping'*: basinhopping
  - *'ampgo'*: Adaptive Memory Programming for Global Optimization
  - '*nelder*': Nelder-Mead
  - *'lbfgsb'*: L-BFGS-B
  - *'powell'*: Powell
  - *'cg'*: Conjugate-Gradient
  - *'newton'*: Newton-CG
  - *'cobyla'*: Cobyla
  - *'bfgs'*: BFGS
  - *'tnc'*: Truncated Newton
  - *'trust-ncg'*: Newton-CG trust-region
  - *'trust-exact'*: nearly exact trust-region
  - *'trust-krylov'*: Newton GLTR trust-region
  - *'trust-constr'*: trust-region for constrained optimization
  - *'dogleg'*: Dog-leg trust-region
  - *'slsqp'*: Sequential Linear Squares Programming
  - *'emcee'*: Maximum likelihood via Monte-Carlo Markov Chain
  - *'shgo'*: Simplicial Homology Global Optimization
  - *'dual_annealing'*: Dual Annealing optimization

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

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

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

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

**Return type** *MinimizerResult*

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

`Minimizer.`**`leastsq`**(*params=None*, *max_nfev=None*, ***kws*)

Use Levenberg-Marquardt minimization to perform a fit.

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

This method calls scipy.optimize.leastsq and, 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 |
| *Dfun* | None | Function to call for Jacobian calculation |

**Parameters**

- **params** (`Parameters, optional`) – Parameters to use as starting point.
- **max_nfev** (`int or None, optional`) – Maximum number of function evaluations. Defaults to `2000*(nvars+1)`, where `nvars` is the number of variable parameters.
- ****kws** (`dict, optional`) – Minimizer options to pass to scipy.optimize.leastsq.

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

**Return type** *MinimizerResult*

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

`Minimizer.`**`least_squares`**(*params=None*, *max_nfev=None*, ***kws*)

Least-squares minimization using scipy.optimize.least_squares.

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

**Parameters**

- **params** (`Parameters, optional`) – Parameters to use as starting point.
- **max_nfev** (`int or None, optional`) – Maximum number of function evaluations. Defaults to `2000*(nvars+1)`, where `nvars` is the number of variable parameters.
- ****kws** (`dict, optional`) – Minimizer options to pass to scipy.optimize.least_squares.

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

**Return type** *MinimizerResult*

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

`Minimizer.`**`scalar_minimize`**(*method='Nelder-Mead'*, *params=None*, *max_nfev=None*, ***kws*)

Scalar minimization using scipy.optimize.minimize.

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

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

**Parameters**

- **method** (*str, optional*) – Name of the fitting method to use. One of:
  - *'Nelder-Mead'* (default)
  - *'L-BFGS-B'*
  - *'Powell'*
  - *'CG'*
  - *'Newton-CG'*
  - *'COBYLA'*
  - *'BFGS'*
  - *'TNC'*
  - *'trust-ncg'*
  - *'trust-exact'*
  - *'trust-krylov'*
  - *'trust-constr'*
  - *'dogleg'*
  - *'SLSQP'*
  - *'differential_evolution'*
- **params** (*Parameters, optional*) – Parameters to use as starting point.
- **max_nfev** (*int or None, optional*) – Maximum number of function evaluations. Defaults to `2000*(nvars+1)`, where `nvars` is the number of variable parameters.
- ****kws** (*dict, optional*) – Minimizer options pass to scipy.optimize.minimize.

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

**Return type** *MinimizerResult*

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

## Notes

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

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

`Minimizer.`**`prepare_fit`**(*params=None*)

Prepare parameters for fitting.

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

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

**Returns**

**Return type** *MinimizerResult*

### Notes

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

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

`Minimizer.`**`brute`**(*params=None*, *Ns=20*, *keep=50*, *workers=1*, *max_nfev=None*)

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

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

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

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

**Parameters**

- **`params`** (`Parameters`, *`optional`*) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.
- **`Ns`** (*`int`*, *`optional`*) – Number of grid points along the axes, if not otherwise specified (see Notes).
- **`keep`** (*`int`*, *`optional`*) – Number of best candidates from the brute force method that are stored in the `candidates` attribute. If *'all'*, then all grid points from scipy.optimize.brute are stored as candidates.
- **`workers`** (*`int or map-like callable`*, *`optional`*) – For parallel evaluation of the grid (see scipy.optimize.brute for more details).
- **`max_nfev`** (*`int or None`*, *`optional`*) – Maximum number of function evaluations (default is None). Defaults to `200000*(nvarys+1)`.

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

**Return type** *MinimizerResult*

New in version 0.9.6.

### Notes

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

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

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

For more information, check the examples in `examples/lmfit_brute_example.ipynb`.

`Minimizer.`**`basinhopping`**`(`*params=None*, *max_nfev=None*, ***kws*`)`

Use the *basinhopping* algorithm to find the global minimum.

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

**Parameters**

- **`params`** (`Parameters`, *`optional`*) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.
- **`max_nfev`** (*`int or None, optional`*) – Maximum number of function evaluations (default is None). Defaults to `200000*(nvarys+1)`.
- **`**kws`** (*`dict, optional`*) – Minimizer options to pass to scipy.optimize.basinhopping.

**Returns** Object containing the optimization results from the basinhopping algorithm.

**Return type** *MinimizerResult*

New in version 0.9.10.

`Minimizer.`**`ampgo`**`(`*params=None*, *max_nfev=None*, ***kws*`)`

Find the global minimum of a multivariate function using AMPGO.

AMPGO stands for 'Adaptive Memory Programming for Global Optimization' and is an efficient algorithm to find the global minimum.

**Parameters**

- **`params`** (`Parameters`, *`optional`*) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.
- **`max_nfev`** (*`int, optional`*) – Maximum number of total function evaluations. If None (default), the optimization will stop after *totaliter* number of iterations (see below)..
- **`**kws`** (*`dict, optional`*) – Minimizer options to pass to the ampgo algorithm, the options are listed below:

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

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

**Return type** *MinimizerResult*

New in version 0.9.10.

### Notes

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

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

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

`Minimizer.`**`shgo`**(*params=None*, *max_nfev=None*, ***kws*)

Use the *SHGO* algorithm to find the global minimum.

SHGO stands for "simplicial homology global optimization" and calls scipy.optimize.shgo using its default arguments.

**Parameters**

- **params** (`Parameters`, *optional*) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.
- **max_nfev** (*int or None, optional*) – Maximum number of function evaluations. Defaults to `200000*(nvars+1)`, where `nvars` is the number of variable parameters.
- ****kws** (*dict, optional*) – Minimizer options to pass to the SHGO algorithm.

**Returns** Object containing the parameters from the SHGO method. The return values specific to scipy.optimize.shgo (`x`, `xl`, `fun`, `funl`, `nfev`, `nit`, `nlfev`, `nlhev`, and `nljev`) are stored as `shgo_<parname>` attributes.

**Return type** *MinimizerResult*

New in version 0.9.14.

`Minimizer.`**`dual_annealing`**(*params=None*, *max_nfev=None*, ***kws*)

Use the *dual_annealing* algorithm to find the global minimum.

This method calls scipy.optimize.dual_annealing using its default arguments.

**Parameters**

- **params** (`Parameters`, *optional*) – Contains the Parameters for the model. If None, then the Parameters used to initialize the Minimizer object are used.
- **max_nfev** (*int or None, optional*) – Maximum number of function evaluations. Defaults to `200000*(nvars+1)`, where `nvars` is the number of variables.
- ****kws** (*dict, optional*) – Minimizer options to pass to the dual_annealing algorithm.

**Returns** Object containing the parameters from the dual_annealing method. The return values specific to scipy.optimize.dual_annealing (`x`, `fun`, `nfev`, `nhev`, `njev`, and `nit`) are stored as `da_<parname>` attributes.

**Return type** *MinimizerResult*

New in version 0.9.14.

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

Bayesian sampling of the posterior distribution.

The method uses the `emcee` Markov Chain Monte Carlo package and assumes that the prior is Uniform. You need to have `emcee` version 3 or newer installed to use this method.

**Parameters**

- **params** (`Parameters`, *optional*) – Parameters to use as starting point. If this is not specified then the Parameters used to initialize the Minimizer object are used.
- **steps** (*int, optional*) – How many samples you would like to draw from the posterior distribution for each of the walkers?
- **nwalkers** (*int, optional*) – Should be set so $nwalkers >> nvarys$, where `nvarys` are the number of parameters being varied during the fit. 'Walkers are the members of the ensemble. They are almost like separate Metropolis-Hastings chains but, of course, the proposal distribution for a given walker depends on the positions of all the other walkers in the ensemble.' - from the *emcee* webpage.
- **burn** (*int, optional*) – Discard this many samples from the start of the sampling regime.
- **thin** (*int, optional*) – Only accept 1 in every *thin* samples.
- **ntemps** (*int, deprecated*) – ntemps has no effect.
- **pos** (*numpy.ndarray, optional*) – Specify the initial positions for the sampler, an ndarray of shape `(nwalkers, nvarys)`. You can also initialise using a previous chain of the same *nwalkers* and `nvarys`. Note that `nvarys` may be one larger than you expect it to be if your `userfcn` returns an array and `is_weighted=False`.
- **reuse_sampler** (*bool, optional*) – Set to True if you have already run *emcee* with the *Minimizer* instance and want to continue to draw from its `sampler` (and so retain the chain history). If False, a new sampler is created. The keywords *nwalkers*, *pos*, and *params* will be ignored when this is set, as they will be set by the existing sampler. **Important**: the Parameters used to create the sampler must not change in-between calls to *emcee*. Alteration of Parameters would include changed `min`, `max`, `vary` and `expr` attributes. This may happen, for example, if you use an altered Parameters object and call the *minimize* method in-between calls to *emcee*.
- **workers** (*Pool-like or int, optional*) – For parallelization of sampling. It can be any Pool-like object with a map method that follows the same calling sequence as the built-in *map* function. If int is given as the argument, then a multiprocessing-based pool is spawned internally with the corresponding number of parallel processes. 'mpi4py'-based parallelization and 'joblib'-based parallelization pools can also be used here. **Note**: because of multiprocessing overhead it may only be worth parallelising if the objective function is expensive to calculate, or if there are a large number of objective evaluations per step (`nwalkers * nvarys`).
- **float_behavior** (*str, optional*) – Meaning of float (scalar) output of objective function. Use *'posterior'* if it returns a log-posterior probability or *'chi2'* if it returns $\chi^2$. See Notes for further details.
- **is_weighted** (*bool, optional*) – Has your objective function been weighted by measurement uncertainties? If `is_weighted=True` then your objective function is assumed to return residuals that have been divided by the true measurement uncertainty `(data - model) / sigma`. If `is_weighted=False` then the objective function is assumed to return unweighted residuals, `data - model`. In this case *emcee* will employ a positive measurement uncertainty during the sampling. This measurement uncertainty will be present in the output params and output chain with the name `__lnsigma`. A side effect of this is that you cannot use this parameter name yourself. **Important**: this parameter only has any effect if your objective function returns an array. If your objective function returns a float, then this parameter is ignored. See Notes for more details.

- **seed** (*int or numpy.random.RandomState, optional*) – If *seed* is an `int`, a new *numpy.random.RandomState* instance is used, seeded with *seed*. If *seed* is already a *numpy.random.RandomState* instance, then that *numpy.random.RandomState* instance is used. Specify *seed* for repeatable minimizations.
- **progress** (*bool, optional*) – Print a progress bar to the console while running.
- **run_mcmc_kwargs** (*dict, optional*) – Additional (optional) keyword arguments that are passed to `emcee.EnsembleSampler.run_mcmc`.

**Returns** MinimizerResult object containing updated params, statistics, etc. The updated params represent the median of the samples, while the uncertainties are half the difference of the 15.87 and 84.13 percentiles. The *MinimizerResult* contains a few additional attributes: *chain* contain the samples and has shape `((steps - burn) // thin, nwalkers, nvarys)`. *flatchain* is a *pandas.DataFrame* of the flattened chain, that can be accessed with *result.flatchain[parname]*. *lnprob* contains the log probability for each sample in *chain*. The sample with the highest probability corresponds to the maximum likelihood estimate. *acor* is an array containing the autocorrelation time for each parameter if the autocorrelation time can be computed from the chain. Finally, *acceptance_fraction* (an array of the fraction of steps accepted for each walker).

**Return type** *MinimizerResult*

## Notes

This method samples the posterior distribution of the parameters using Markov Chain Monte Carlo. It calculates the log-posterior probability of the model parameters, *F*, given the data, *D*, $\ln p(F_{true}|D)$. This 'posterior probability' is given by:

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

where $\ln p(D|F_{true})$ is the 'log-likelihood' and $\ln p(F_{true})$ is the 'log-prior'. The default log-prior encodes prior information known about the model that the log-prior probability is `-numpy.inf` (impossible) if any of the parameters is outside its limits, and is zero if all the parameters are inside their bounds (uniform prior). The log-likelihood function is[1]:

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

The first term represents the residual ($g$ being the generative model, $D_n$ the data and $s_n$ the measurement uncertainty). This gives $\chi^2$ when summed over all data points. The objective function may also return the log-posterior probability, $\ln p(F_{true}|D)$. Since the default log-prior term is zero, the objective function can also just return the log-likelihood, unless you wish to create a non-uniform prior.

If the objective function returns a float value, this is assumed by default to be the log-posterior probability, (*float_behavior* default is 'posterior'). If your objective function returns $\chi^2$, then you should use `float_behavior='chi2'` instead.

By default objective functions may return an ndarray of (possibly weighted) residuals. In this case, use *is_weighted* to select whether these are correctly weighted by measurement uncertainty. Note that this ignores the second term above, so that to calculate a correct log-posterior probability value your objective function should return a float value. With `is_weighted=False` the data uncertainty, *s_n*, will be treated as a nuisance parameter to be marginalized out. This uses strictly positive uncertainty (homoscedasticity) for each data point, $s_n = \exp(\_\_\mathrm{lnsigma})$. `__lnsigma` will be present in *MinimizerResult.params*, as well as *Minimizer.chain* and `nvarys` will be increased by one.

[1] https://emcee.readthedocs.io

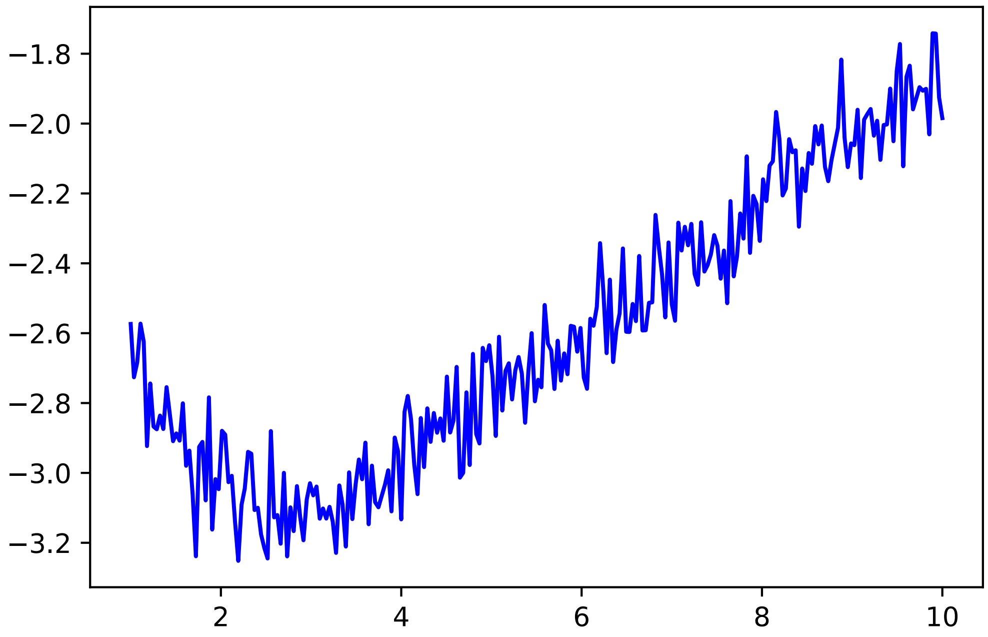
### References

## 6.8 `Minimizer.emcee()` - calculating the posterior probability distribution of parameters

*`Minimizer.emcee()`* can be used to obtain the posterior probability distribution of parameters, given a set of experimental data. Note that this method does *not* actually perform a fit at all. Instead, it explores parameter space to determine the probability distributions for the parameters, but without an explicit goal of attempting to refine the solution. It should not be used for fitting, but it is a useful method to to more thoroughly explore the parameter space around the solution after a fit has been done and thereby get an improved understanding of the probability distribution for the parameters. It may be able to refine your estimate of the most likely values for a set of parameters, but it will not iteratively find a good solution to the minimization problem. To use this method effectively, you should first use another minimization method and then use this method to explore the parameter space around thosee best-fit values.

To illustrate this, we'll use an example problem of fitting data to function of a double exponential decay, including a modest amount of Gaussian noise to the data. Note that this example is the same problem used in *An advanced example for evaluating confidence intervals* for evaluating confidence intervals in the parameters, which is a similar goal to the one here.

```
import matplotlib.pyplot as plt
import numpy as np

import lmfit

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(x.
↪size)
plt.plot(x, y, 'b')
plt.show()
```



Create a Parameter set for the initial guesses:

```
p = lmfit.Parameters()
p.add_many(('a1', 4.), ('a2', 4.), ('t1', 3.), ('t2', 3., True))

def residual(p):
    v = p.valuesdict()
    return v['a1'] * np.exp(-x / v['t1']) + v['a2'] * np.exp(-(x - 0.1) / v['t2']) - y
```

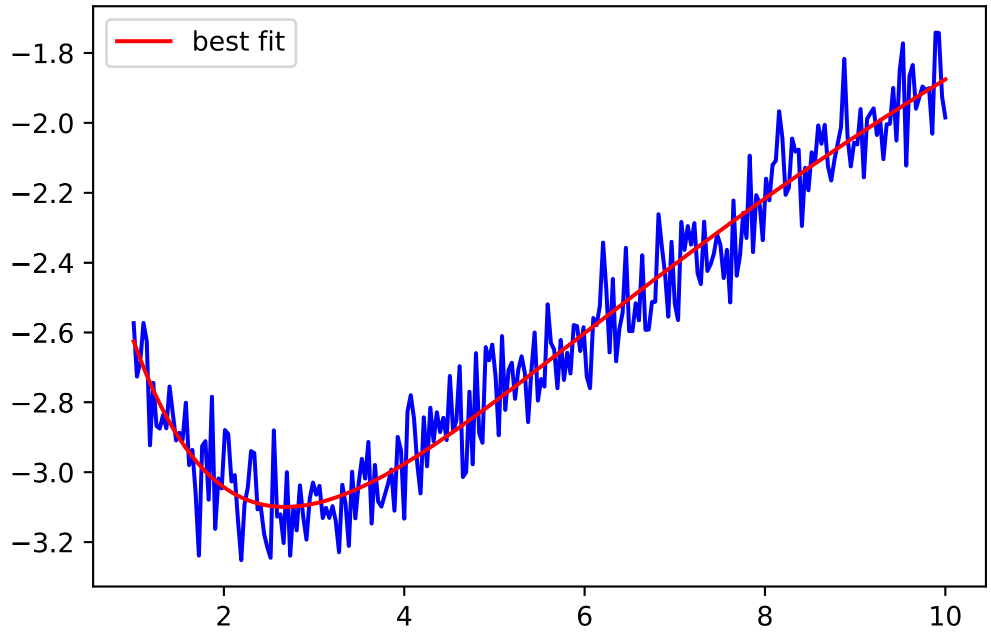
Solving with *minimize()* gives the Maximum Likelihood solution. Note that we use the robust Nelder-Mead method here. The default Levenberg-Marquardt method seems to have difficulty with exponential decays, though it can refine the solution if starting near the solution:

```
mi = lmfit.minimize(residual, p, method='nelder', nan_policy='omit')
lmfit.printfuncs.report_fit(mi.params, min_correl=0.5)
```

```
[[Variables]]
    a1:  2.98623689 +/- 0.15010519 (5.03%) (init = 4)
    a2: -4.33525597 +/- 0.11765824 (2.71%) (init = 4)
    t1:  1.30993186 +/- 0.13449656 (10.27%) (init = 3)
    t2:  11.8240752 +/- 0.47172610 (3.99%) (init = 3)
[[Correlations]] (unreported correlations are < 0.500)
    C(a2, t2) =  0.988
    C(a2, t1) = -0.928
    C(t1, t2) = -0.885
    C(a1, t1) = -0.609
```

and plotting the fit using the Maximum Likelihood solution gives the graph below:

```
plt.plot(x, y, 'b')
plt.plot(x, residual(mi.params) + y, 'r', label='best fit')
plt.legend(loc='best')
plt.show()
```



Note that the fit here (for which the `numdifftools` package is installed) does estimate and report uncertainties in the parameters and correlations for the parameters, and reports the correlation of parameters `a2` and `t2` to be very high. As we'll see, these estimates are pretty good, but when faced with such high correlation, it can be helpful to get the full probability distribution for the parameters. MCMC methods are very good for this.

Furthermore, we wish to deal with the data uncertainty. This is called marginalisation of a nuisance parameter. `emcee` requires a function that returns the log-posterior probability. The log-posterior probability is a sum of the log-prior probability and log-likelihood functions. The log-prior probability is assumed to be zero if all the parameters are within their bounds and `-np.inf` if any of the parameters are outside their bounds.

If the objective function returns an array of unweighted residuals (i.e., `data-model`) as is the case here, you can use `is_weighted=False` as an argument for `emcee`. In that case, `emcee` will automatically add/use the `__lnsigma` parameter to estimate the true uncertainty in the data. To place boundaries on this parameter one can do:

```
mi.params.add('__lnsigma', value=np.log(0.1), min=np.log(0.001), max=np.log(2))
```

Now we have to set up the minimizer and do the sampling (again, just to be clear, this is *not* doing a fit):

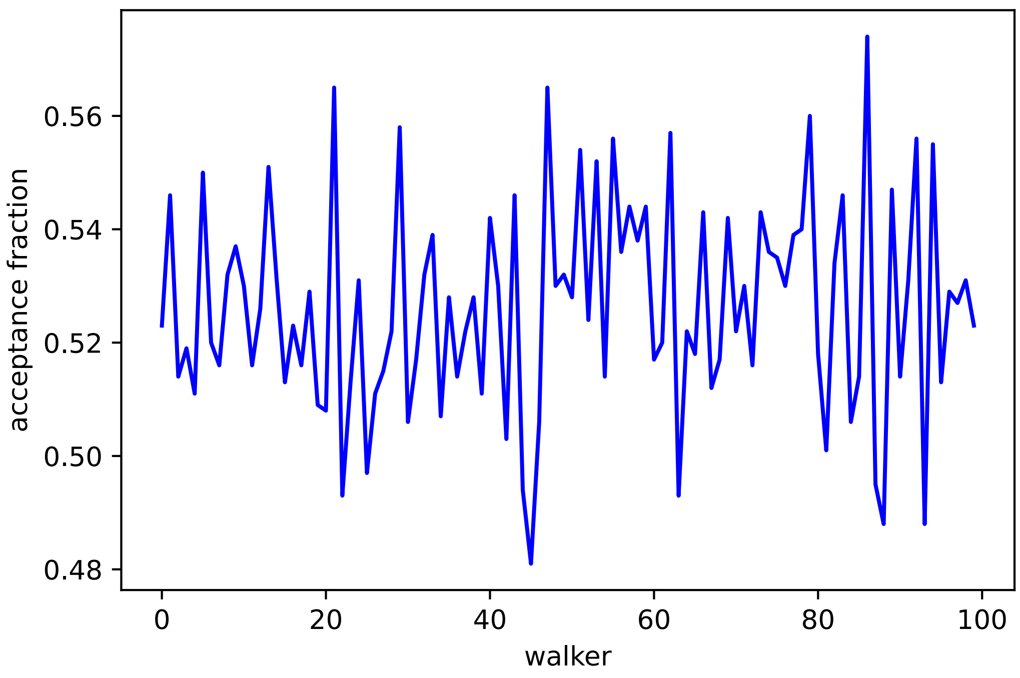
```
res = lmfit.minimize(residual, method='emcee', nan_policy='omit', burn=300,␣
↪steps=1000, thin=20,
                     params=mi.params, is_weighted=False, progress=False)
```

As mentioned in the Notes for *`Minimizer.emcee()`*, the `is_weighted` argument will be ignored if your objective function returns a float instead of an array. For the documentation we set `progress=False`; the default is to print a progress bar to the Terminal if the `tqdm` package is installed.

The success of the method (i.e., whether or not the sampling went well) can be assessed by checking the integrated autocorrelation time and/or the acceptance fraction of the walkers. For this specific example the autocorrelation time could not be estimated because the "chain is too short". Instead, we plot the acceptance fraction per walker and its mean value suggests that the sampling worked as intended (as a rule of thumb the value should be between 0.2 and 0.5).

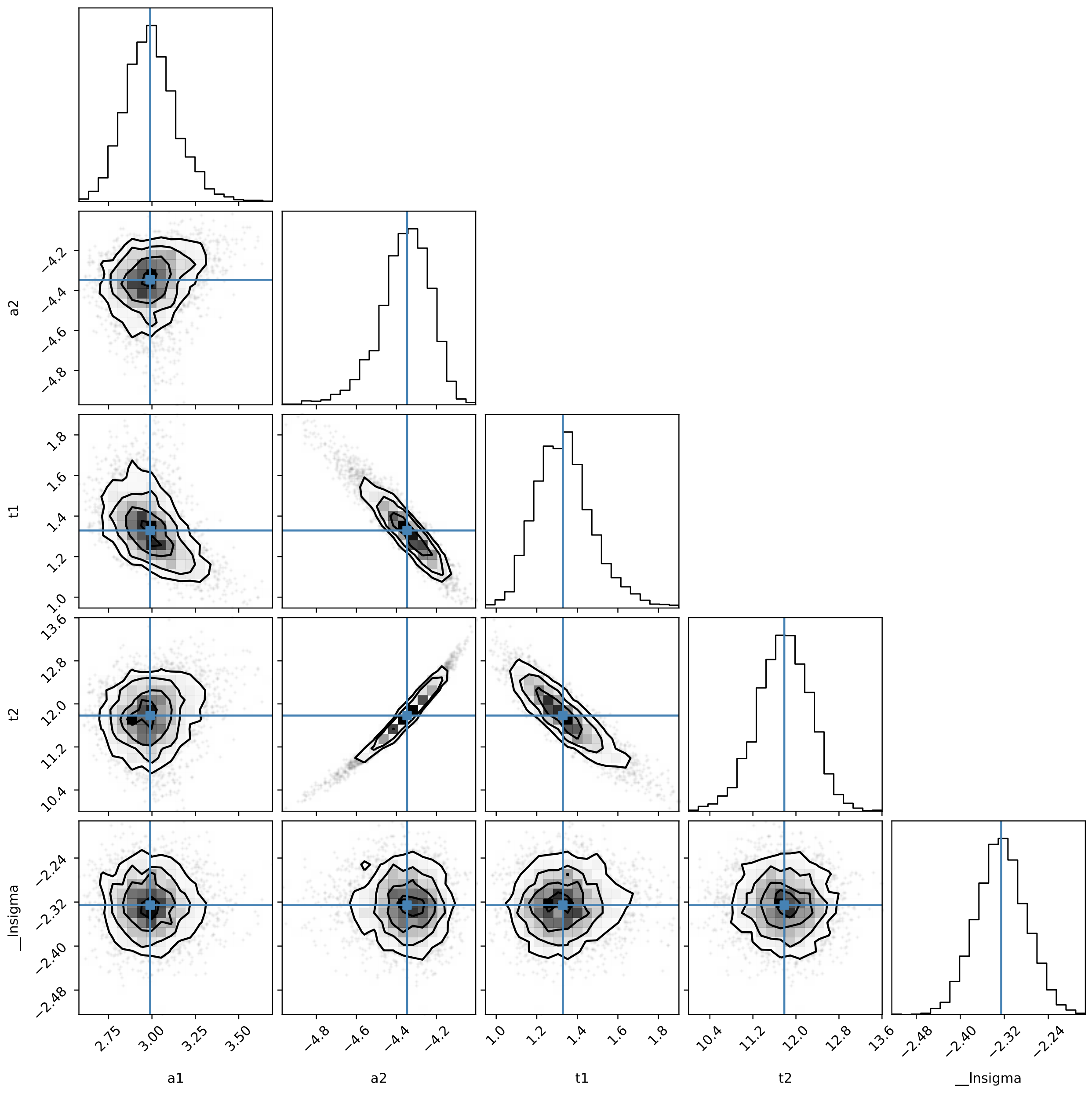
```
plt.plot(res.acceptance_fraction, 'b')
plt.xlabel('walker')
plt.ylabel('acceptance fraction')
```

```
Text(0, 0.5, 'acceptance fraction')
```



With the results from `emcee`, we can visualize the posterior distributions for the parameters using the `corner` package:

```
import corner

emcee_plot = corner.corner(res.flatchain, labels=res.var_names,
                           truths=list(res.params.valuesdict().values()))
```



The values reported in the *MinimizerResult* are the medians of the probability distributions and a 1 $\sigma$ quantile, estimated as half the difference between the 15.8 and 84.2 percentiles. Printing these values:

```
print('median of posterior probability distribution')
print('--------------------------------------------')
lmfit.report_fit(res.params)
```

```
median of posterior probability distribution
--------------------------------------------
[[Variables]]
    a1:         2.98945718 +/- 0.14033921 (4.69%) (init = 2.986237)
    a2:        -4.34687243 +/- 0.12131092 (2.79%) (init = -4.335256)
    t1:         1.32883916 +/- 0.13766047 (10.36%) (init = 1.309932)
    t2:         11.7836194 +/- 0.47719763 (4.05%) (init = 11.82408)
```

(continues on next page)

(continued from previous page)

```
    __lnsigma: -2.32559226 +/- 0.04542650 (1.95%) (init = -2.302585)
[[Correlations]] (unreported correlations are < 0.100)
    C(a2, t2) =  0.981
    C(a2, t1) = -0.938
    C(t1, t2) = -0.894
    C(a1, t1) = -0.508
    C(a1, a2) =  0.214
    C(a1, t2) =  0.178
```

You can see that this recovered the right uncertainty level on the data. Note that these values agree pretty well with the results, uncertainties and correlations found by the fit and using `numdifftools` to estimate the covariance matrix. That is, even though the parameters `a2`, `t1`, and `t2` are all highly correlated and do not display perfectly Gaussian probability distributions, the probability distributions found by explicitly sampling the parameter space are not so far from elliptical as to make the simple (and much faster) estimates from inverting the covariance matrix completely invalid.

As mentioned above, the result from `emcee` reports the median values, which are not necessarily the same as the Maximum Likelihood Estimate. To obtain the values for the Maximum Likelihood Estimation (MLE) we find the location in the chain with the highest probability:

```
highest_prob = np.argmax(res.lnprob)
hp_loc = np.unravel_index(highest_prob, res.lnprob.shape)
mle_soln = res.chain[hp_loc]
for i, par in enumerate(p):
    p[par].value = mle_soln[i]


print('\nMaximum Likelihood Estimation from emcee       ')
print('-------------------------------------------------')
print('Parameter  MLE Value   Median Value   Uncertainty')
fmt = '  {:5s}  {:11.5f} {:11.5f}   {:11.5f}'.format
for name, param in p.items():
    print(fmt(name, param.value, res.params[name].value,
              res.params[name].stderr))
```

```

Maximum Likelihood Estimation from emcee
-------------------------------------------------
Parameter  MLE Value   Median Value   Uncertainty
  a1         2.93839     2.98946       0.14034
  a2        -4.35274    -4.34687       0.12131
  t1         1.34310     1.32884       0.13766
  t2        11.78782    11.78362       0.47720
```

Here the difference between MLE and median value are seen to be below 0.5%, and well within the estimated 1-$\sigma$ uncertainty.

Finally, we can use the samples from `emcee` to work out the 1- and 2-$\sigma$ error estimates.

```
print('\nError estimates from emcee:')
print('------------------------------------------------------')
print('Parameter  -2sigma  -1sigma   median  +1sigma  +2sigma')

for name in p.keys():
    quantiles = np.percentile(res.flatchain[name],
                              [2.275, 15.865, 50, 84.135, 97.275])
```

(continues on next page)

(continued from previous page)

```
    median = quantiles[2]
    err_m2 = quantiles[0] - median
    err_m1 = quantiles[1] - median
    err_p1 = quantiles[3] - median
    err_p2 = quantiles[4] - median
    fmt = '  {:5s}   {:8.4f} {:8.4f} {:8.4f} {:8.4f} {:8.4f}'.format
    print(fmt(name, err_m2, err_m1, median, err_p1, err_p2))
```

```

Error estimates from emcee:
------------------------------------------------------
Parameter  -2sigma  -1sigma   median  +1sigma  +2sigma
  a1       -0.2656  -0.1362   2.9895   0.1445   0.3141
  a2       -0.3209  -0.1309  -4.3469   0.1118   0.1985
  t1       -0.2377  -0.1305   1.3288   0.1448   0.3278
  t2       -1.0677  -0.4807  11.7836   0.4739   0.8990
```

And we see that the initial estimates for the 1-$\sigma$ standard error using `numdifftools` was not too bad. We'll return to this example problem in *An advanced example for evaluating confidence intervals* and use a different method to calculate the 1- and 2-$\sigma$ error bars.

CHAPTER

SEVEN

# 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 so that it most closely matches some data. With `scipy`, such problems are typically solved with scipy.optimize.curve_fit, which is a wrapper around scipy.optimize.leastsq. Since lmfit's *`minimize()`* is also a high-level wrapper around scipy.optimize.leastsq it can be used for curve-fitting problems. While it offers many benefits over scipy.optimize.leastsq, using *`minimize()`* for many curve-fitting problems still requires more effort than using scipy.optimize.curve_fit.

The *`Model`* class in lmfit provides a simple and flexible approach to curve-fitting problems. Like scipy.optimize.curve_fit, a *`Model`* uses a *model function* – a function that is meant to calculate a model for some phenomenon – and then uses that to best match an array of supplied data. Beyond that similarity, its interface is rather different from scipy.optimize.curve_fit, for example in that it uses *`Parameters`*, but also offers several other important advantages.

In addition to allowing you to turn any model function into a curve-fitting method, lmfit also provides canonical definitions for many known line shapes 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 functions into high-level fitting models with the *`Model`* class, and using these to fit data.

## 7.1 Motivation and simple 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 built-in `GaussianModel` class that can help do this, but here we'll build our own. We start with a simple definition of the model function:

```
from numpy import exp, linspace, random

def gaussian(x, amp, cen, wid):
    return amp * exp(-(x-cen)**2 / wid)
```

We want to use this function to fit to data $y(x)$ represented by the arrays `y` and `x`. With scipy.optimize.curve_fit, this would be:

```
from scipy.optimize import curve_fit

x = linspace(-10, 10, 101)
y = gaussian(x, 2.33, 0.21, 1.51) + random.normal(0, 0.2, x.size)

init_vals = [1, 0, 1]  # for [amp, cen, wid]
best_vals, covar = curve_fit(gaussian, x, y, p0=init_vals)
```

That is, we create data, 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 useful, but it misses the benefits of lmfit.

With lmfit, we create a *Model* that wraps the `gaussian` model function, which automatically generates the appropriate residual function, and determines the corresponding parameter names from the function signature itself:

```
from lmfit import Model

gmodel = Model(gaussian)
print('parameter names: {}'.format(gmodel.param_names))
print('independent variables: {}'.format(gmodel.independent_vars))
```

```
parameter names: ['amp', 'cen', 'wid']
independent variables: ['x']
```

As you can see, the Model `gmodel` determined 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 independent variable is `x`, and the parameters are named `amp`, `cen`, and `wid`, and – all taken directly from the signature of the model function. As we will see below, you can modify the default assignment of independent variable / arguments and specify yourself what the independent variable is and which function arguments should be identified as parameter names.

The Parameters are *not* created when the model is created. The model knows what the parameters should be named, but nothing about the scale and range of your data. You will normally have to make these parameters and assign initial 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 = gmodel.make_params()
```

This creates the *Parameters* but does not automatically give them initial values since it has no idea what the scale should be. You can set initial values for parameters with keyword arguments to `make_params()`:

```
params = gmodel.make_params(cen=0.3, amp=3, wid=1.25)
```

or assign them (and other parameter properties) after the *Parameters* class 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_eval = linspace(0, 10, 201)
y_eval = gmodel.eval(params, x=x_eval)
```

or with:

```
y_eval = gmodel.eval(x=x_eval, cen=6.5, amp=100, wid=2.0)
```

Admittedly, this a slightly long-winded way to calculate a Gaussian function, given that you could have called your `gaussian` function directly. But now that the model is set up, we can use its `fit()` method to fit this model to data, as with:

```
result = gmodel.fit(y, params, x=x)
```

or with:

```
result = gmodel.fit(y, x=x, cen=0.5, amp=10, wid=2.0)
```

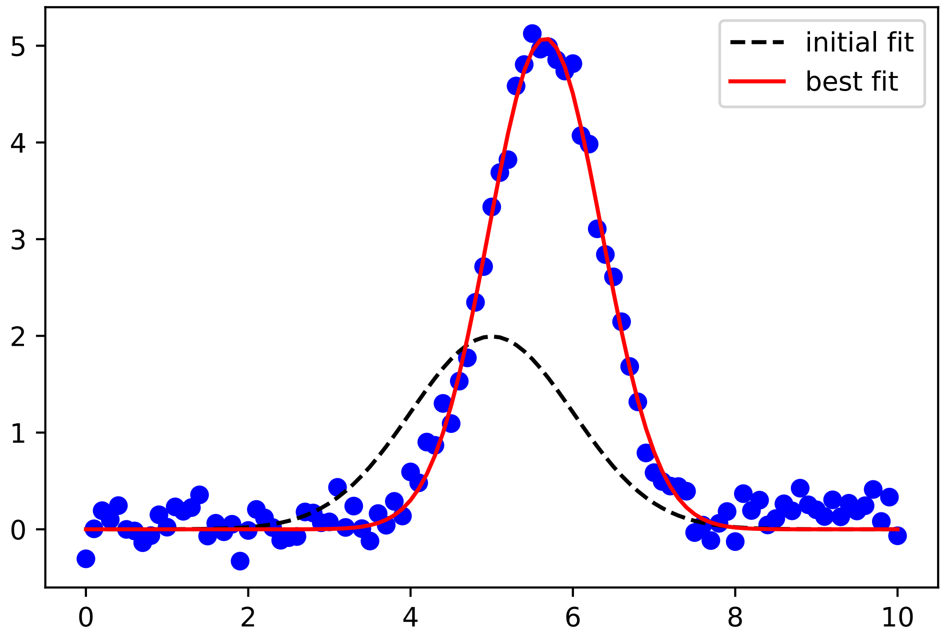
Putting everything together, included in the `examples` folder with the source code, is:

```
# <examples/doc_model_gaussian.py>
import matplotlib.pyplot as plt
from numpy import exp, loadtxt, pi, sqrt

from lmfit import Model

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))


gmodel = Model(gaussian)
result = gmodel.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--', label='initial fit')
plt.plot(x, result.best_fit, 'r-', label='best fit')
plt.legend(loc='best')
plt.show()
# <end examples/doc_model_gaussian.py>
```

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

```
[[Model]]
    Model(gaussian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 33
    # data points      = 101
    # variables        = 3
    chi-square         = 3.40883599
    reduced chi-square = 0.03478404
    Akaike info crit   = -336.263713
    Bayesian info crit = -328.418352
[[Variables]]
    amp:  8.88021830 +/- 0.11359492 (1.28%) (init = 5)
    cen:  5.65866102 +/- 0.01030495 (0.18%) (init = 5)
    wid:  0.69765468 +/- 0.01030495 (1.48%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(amp, wid) =  0.577
```

As the script shows, 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.

Note that the model fitting was really performed with:

```
gmodel = Model(gaussian)
result = gmodel.fit(y, params, 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`. In addition, all the other features of lmfit are included: *`Parameters`* can have bounds and constraints and the result is a rich object that can be reused to explore the model fit in detail.

## 7.2 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*, *nan_policy='raise'*, *prefix=''*, *name=None*, ***kws*)

Create a model from a user-supplied model function.

The model function will normally take an independent variable (generally, the first argument) and a series of arguments that are meant to be parameters for the model. It will return an array of data to model some data as for a curve-fitting problem.

**Parameters**

- **`func`** (*`callable`*) – Function to be wrapped.
- **`independent_vars`** (`list` of `str`, optional) – Arguments to *func* that are independent variables (default is None).
- **`param_names`** (`list` of `str`, optional) – Names of arguments to *func* that are to be made into parameters (default is None).

- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- **prefix** (*str, optional*) – Prefix used for the model.
- **name** (*str, optional*) – Name for the model. When None (default) the name is the same as the model function (*func*).
- ****kws** (*dict, optional*) – Additional keyword arguments to pass to model function.

### Notes

1. Parameter names are inferred from the function arguments, and a residual function is automatically constructed.

2. The model function must return an array that will be the same size as the data being modeled.

3. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:

   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### Examples

The model function will normally take an independent variable (generally, the first argument) and a series of arguments that are meant to be parameters for the model. Thus, a simple peak using a Gaussian defined as:

```
>>> import numpy as np
>>> def gaussian(x, amp, cen, wid):
...     return amp * np.exp(-(x-cen)**2 / wid)
```

can be turned into a Model with:

```
>>> gmodel = Model(gaussian)
```

this will automatically discover the names of the independent variables and parameters:

```
>>> print(gmodel.param_names, gmodel.independent_vars)
['amp', 'cen', 'wid'], ['x']
```

## 7.2.1 Model class Methods

Model.**eval**(*params=None*, ***kwargs*)

Evaluate the model with supplied parameters and keyword arguments.

**Parameters**

- **params** (*Parameters, optional*) – Parameters to use in Model.
- ****kwargs** (*optional*) – Additional keyword arguments to pass to model function.

**Returns** Value of model given the parameters and other arguments.

**Return type** numpy.ndarray

### Notes

1. 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.

2. all non-parameter arguments for the model function, **including all the independent variables** will need to be passed in using keyword arguments.

`Model.`**`fit`**(*data*, *params=None*, *weights=None*, *method='leastsq'*, *iter_cb=None*, *scale_covar=True*, *verbose=False*, *fit_kws=None*, *nan_policy=None*, *calc_covar=True*, *max_nfev=None*, ***kwargs*)

Fit the model to the data using the supplied Parameters.

**Parameters**

- **data** (*array_like*) – Array of data to be fit.
- **params** (*Parameters, optional*) – Parameters to use in fit (default is None).
- **weights** (*array_like, optional*) – Weights to use for the calculation of the fit residual (default is None). Must have the same size as *data*.
- **method** (*str, optional*) – Name of fitting method to use (default is *'leastsq'*).
- **iter_cb** (*callable, optional*) – Callback function to call at each iteration (default is None).
- **scale_covar** (*bool, optional*) – Whether to automatically scale the covariance matrix when calculating uncertainties (default is True).
- **verbose** (*bool, optional*) – Whether to print a message when a new parameter is added because of a hint (default is True).
- **fit_kws** (*dict, optional*) – Options to pass to the minimizer being used.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – What to do when encountering NaNs when fitting Model.
- **calc_covar** (*bool, optional*) – Whether to calculate the covariance matrix (default is True) for solvers other than *'leastsq'* and *'least_squares'*. Requires the `numdifftools` package to be installed.
- **max_nfev** (*int or None, optional*) – Maximum number of function evaluations (default is None). The default value depends on the fitting method.
- ****kwargs** (*optional*) – Arguments to pass to the model function, possibly overriding parameters.

**Returns**

**Return type** *ModelResult*

### Notes

1. 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.

2. all non-parameter arguments for the model function, **including all the independent variables** will need to be passed in using keyword arguments.

3. Parameters (however passed in), are copied on input, so the original Parameter objects are unchanged, and the updated values are in the returned *ModelResult*.

### Examples

Take `t` to be the independent variable and data to be the curve we will fit. Use keyword arguments to set initial guesses:

```
>>> result = my_model.fit(data, tau=5, N=3, t=t)
```

Or, for more control, pass a Parameters object.

```
>>> result = my_model.fit(data, params, t=t)
```

Keyword arguments override Parameters.

```
>>> result = my_model.fit(data, params, tau=5, t=t)
```

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

Guess starting values for the parameters of a Model.

This is not implemented for all models, but is available for many of the built-in models.

**Parameters**

- **`data`** (*`array_like`*) – Array of data to use to guess parameter values.
- **`**kws`** (*`optional`*) – Additional keyword arguments, passed to model function.

**Returns** Initial, guessed values for the parameters of a Model.

**Return type** *Parameters*

**Raises** **`NotImplementedError`** – If the *guess* method is not implemented for a Model.

### Notes

Should be implemented for each model subclass to run *self.make_params()*, update starting values and return a Parameters object.

`Model.`**`make_params`**(*verbose=False*, ***kwargs*)

Create a Parameters object for a Model.

**Parameters**

- **`verbose`** (*`bool, optional`*) – Whether to print out messages (default is False).
- **`**kwargs`** (*`optional`*) – Parameter names and initial values.

**Returns** **params** – Parameters object for the Model.

**Return type** *Parameters*

### Notes

1. The parameters may or may not have decent initial values for each parameter.
2. This applies any default values or parameter hints that may have been set.

`Model.`**`set_param_hint`**(*name*, ***kwargs*)

Set *hints* to use when creating parameters with *make_params()*.

This is especially convenient for setting initial values. The *name* can include the models *prefix* or not. The hint given can also include optional bounds and constraints `(value, vary, min, max, expr)`, which will be used by *make_params()* when building default parameters.

**Parameters**

- **name** (*str*) – Parameter name.
- ****kwargs** (*optional*) – Arbitrary keyword arguments, needs to be a Parameter attribute. Can be any of the following:
  - **value** [float, optional] Numerical Parameter value.
  - **vary** [bool, optional] Whether the Parameter is varied during a fit (default is True).
  - **min** [float, optional] Lower bound for value (default is `-numpy.inf`, no lower bound).
  - **max** [float, optional] Upper bound for value (default is `numpy.inf`, no upper bound).
  - **expr** [str, optional] Mathematical expression used to constrain the value during the fit.

### Example

```
>>> model = GaussianModel()
>>> model.set_param_hint('sigma', min=0)
```

See *Using parameter hints*.

`Model.`**`print_param_hints`**(*colwidth=8*)

Print a nicely aligned text-table of parameter hints.

**Parameters colwidth** (*int, optional*) – Width of each column, except for first and last columns.

## 7.2.2 `Model` class Attributes

**`func`**

The model function used to calculate the model.

**`independent_vars`**

List of strings for names of the independent variables.

**`nan_policy`**

Describes what to do for NaNs that indicate missing values in the data. The choices are:

- `'raise'`: Raise a `ValueError` (default)
- `'propagate'`: Do not check for NaNs or missing values. The fit will try to ignore them.
- `'omit'`: Remove NaNs or missing observations in data. If pandas is installed, `pandas.isnull()` is used, otherwise `numpy.isnan()` is used.

**`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.

### 7.2.3 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 taken as the first argument to the function. You can, of course, explicitly set this, and will need to do so if the independent variable is not first in the list, or if there is actually more than one independent variable.

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 will have to do this because none of the parameters have valid initial values. In addition, one can place bounds and constraints on Parameters, or fix their values.

### 7.2.4 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 one:

```
import numpy as np
from lmfit import Model


def decay(t, tau, N):
   return N*np.exp(-t/tau)


decay_model = Model(decay)
print('independent variables: {}'.format(decay_model.independent_vars))

params = decay_model.make_params()
print('\nParameters:')
for pname, par in params.items():
    print(pname, par)
```

```
independent variables: ['t']

Parameters:
tau <Parameter 'tau', value=-inf, bounds=[-inf:inf]>
N <Parameter 'N', value=-inf, bounds=[-inf:inf]>
```

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('independent variables: {}'.format(decay_model.independent_vars))

params = decay_model.make_params()
```

(continues on next page)

(continued from previous page)

```
print('\nParameters:')
for pname, par in params.items():
    print(pname, par)
```

```
independent variables: ['tau']

Parameters:
t <Parameter 't', value=-inf, bounds=[-inf:inf]>
N <Parameter 'N', value=-inf, bounds=[-inf:inf]>
```

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 `Model.fit()` or evaluation with `Model.eval()`.

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

### 7.2.5 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)
params = mod.make_params()
print('Parameters:')
for pname, par in params.items():
    print(pname, par)
```

```
Parameters:
tau <Parameter 'tau', value=-inf, bounds=[-inf:inf]>
N <Parameter 'N', value=10, bounds=[-inf:inf]>
```

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.

### 7.2.6 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 *Composite Models : adding (or multiplying) 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):
    # function definition, for now just ``pass``
    pass


mod = Model(myfunc, prefix='f1_')
params = mod.make_params()
print('Parameters:')
for pname, par in params.items():
    print(pname, par)
```

```
Parameters:
f1_amplitude <Parameter 'f1_amplitude', value=1, bounds=[-inf:inf]>
f1_center <Parameter 'f1_center', value=0, bounds=[-inf:inf]>
f1_sigma <Parameter 'f1_sigma', value=1, bounds=[-inf:inf]>
```

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`.

### 7.2.7 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 `Model.make_params()`.
3. You can give parameter hints with `Model.set_param_hint()`.
4. You can supply initial values for the parameters when you use the `Model.eval()` or `Model.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):
    return a*x + 10*a - b
```

instead of using:

```
def myfunc(x, a, b):
    return a*x + 10*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 `Model.make_params()`

When creating parameters with *Model.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 *Model.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 *Model.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 *Model.make_params()*, you can include values as keyword arguments to either the *Model.eval()* or *Model.fit()* methods:

```
x = linspace(0, 10, 100)
y_eval = mod.eval(x=x, a=7.0, b=-2.0)
y_sim = y_eval + random.normal(0, 0.2, x.size)
out = mod.fit(y_sim, pars, x=x, a=3.0, b=0.0)
```

These approaches 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 *Model.fit()*.

### 7.2.8 Using parameter hints

After a model has been created, you can give it hints for how to create parameters with *Model.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 *Model.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('Parameter hints:')
for pname, par in mod.param_hints.items():
    print(pname, par)
```

```
Parameter hints:
a OrderedDict([('value', 1.0)])
b OrderedDict([('value', 0.3), ('min', 0), ('max', 1.0)])
```

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

### 7.2.9 Saving and Loading Models

New in version 0.9.8.

It is sometimes desirable to save a `Model` for later use outside of the code used to define the model. Lmfit provides a `save_model()` function that will save a `Model` to a file. There is also a companion `load_model()` function that can read this file and reconstruct a `Model` from it.

Saving a model turns out to be somewhat challenging. The main issue is that Python is not normally able to *serialize* a function (such as the model function making up the heart of the Model) in a way that can be reconstructed into a callable Python object. The `dill` package can sometimes serialize functions, but with the limitation that it can be used only in the same version of Python. In addition, class methods used as model functions will not retain the rest of the class attributes and methods, and so may not be usable. With all those warnings, it should be emphasized that if you are willing to save or reuse the definition of the model function as Python code, then saving the Parameters and rest of the components that make up a model presents no problem.

If the `dill` package is installed, the model function will be saved using it. But because saving the model function is not always reliable, saving a model will always save the *name* of the model function. The `load_model()` takes an optional `funcdefs` argument that can contain a dictionary of function definitions with the function names as keys and function objects as values. If one of the dictionary keys matches the saved name, the corresponding function object will be used as the model function. With this approach, if you save a model and can provide the code used for the model function, the model can be saved and reliably reloaded and used.

**`save_model`**(*model*, *fname*)
Save a Model to a file.

**Parameters**

- **`model`** (`Model`) – Model to be saved.
- **`fname`** (*`str`*) – Name of file for saved Model.

**`load_model`**(*fname*, *funcdefs=None*)
Load a saved Model from a file.

**Parameters**

- **`fname`** (*`str`*) – Name of file containing saved Model.
- **`funcdefs`** (*`dict, optional`*) – Dictionary of custom function names and definitions.

**Returns** Model object loaded from file.

**Return type** *Model*

As a simple example, one can save a model as:

```
# <examples/doc_model_savemodel.py>
import numpy as np

from lmfit.model import Model, save_model


def mysine(x, amp, freq, shift):
    return amp * np.sin(x*freq + shift)


sinemodel = Model(mysine)
pars = sinemodel.make_params(amp=1, freq=0.25, shift=0)

save_model(sinemodel, 'sinemodel.sav')
# <end examples/doc_model_savemodel.py>
```

To load that later, one might do:

```
# <examples/doc_model_loadmodel.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.model import load_model


def mysine(x, amp, freq, shift):
    return amp * np.sin(x*freq + shift)


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

model = load_model('sinemodel.sav', funcdefs={'mysine': mysine})
params = model.make_params(amp=3, freq=0.52, shift=0)
params['shift'].max = 1
params['shift'].min = -1
params['amp'].min = 0.0

result = model.fit(y, params, x=x)
print(result.fit_report())

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

See also *Saving and Loading ModelResults*.

## 7.3 The `ModelResult` class

A *`ModelResult`* (which had been called `ModelFit` prior to version 0.9) is the object returned by *`Model.fit()`*. It is a subclass 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 *`ModelResult`* *does* contain parameters and data as well as methods to alter and re-do fits. Thus the *`Model`* is the idealized model while the *`ModelResult`* 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 *`ModelResult`* has several attributes holding values for fit results, and several methods for working with fits. These include statistics inherited from *`Minimizer`* useful for comparing different models, including `chisqr`, `redchi`, `aic`, and `bic`.

**class ModelResult**(*model*, *params*, *data=None*, *weights=None*, *method='leastsq'*, *fcn_args=None*, *fcn_kws=None*, *iter_cb=None*, *scale_covar=True*, *nan_policy='raise'*, *calc_covar=True*, *max_nfev=None*, ***fit_kws*)

Result from the Model fit.

This has many attributes and methods for viewing and working with the results of a fit using Model. It inherits from Minimizer, so that it can be used to modify and re-run the fit for the Model.

**Parameters**

- **model** (`Model`) – Model to use.
- **params** (`Parameters`) – Parameters with initial values for model.
- **data** (*array_like, optional*) – Data to be modeled.
- **weights** (*array_like, optional*) – Weights to multiply `(data-model)` for fit residual.
- **method** (*str, optional*) – Name of minimization method to use (default is *'leastsq'*).
- **fcn_args** (*sequence, optional*) – Positional arguments to send to model function.
- **fcn_dict** (*dict, optional*) – Keyword arguments to send to model function.
- **iter_cb** (*callable, optional*) – Function to call on each iteration of fit.
- **scale_covar** (*bool, optional*) – Whether to scale covariance matrix for uncertainty evaluation.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – What to do when encountering NaNs when fitting Model.
- **calc_covar** (*bool, optional*) – Whether to calculate the covariance matrix (default is True) for solvers other than *'leastsq'* and *'least_squares'*. Requires the `numdifftools` package to be installed.
- **max_nfev** (*int or None, optional*) – Maximum number of function evaluations (default is None). The default value depends on the fitting method.
- ****fit_kws** (*optional*) – Keyword arguments to send to minimization routine.

### 7.3.1 `ModelResult` methods

`ModelResult.`**`eval`**`(`*params=None*, ***kwargs*`)`

Evaluate model function.

**Parameters**

- **`params`** (`Parameters`, *`optional`*) – Parameters to use.
- **`**kwargs`** (*`optional`*) – Options to send to Model.eval().

**Returns** Array for evaluated model.

**Return type** numpy.ndarray

`ModelResult.`**`eval_components`**`(`*params=None*, ***kwargs*`)`

Evaluate each component of a composite model function.

**Parameters**

- **`params`** (`Parameters`, *`optional`*) – Parameters, defaults to ModelResult.params.
- **`**kwargs`** (*`optional`*) – Keyword arguments to pass to model function.

**Returns** Keys are prefixes of component models, and values are the estimated model value for each component of the model.

**Return type** OrderedDict

`ModelResult.`**`fit`**`(`*data=None*, *params=None*, *weights=None*, *method=None*, *nan_policy=None*, ***kwargs*`)`

Re-perform fit for a Model, given data and params.

**Parameters**

- **`data`** (*`array_like, optional`*) – Data to be modeled.
- **`params`** (`Parameters`, *`optional`*) – Parameters with initial values for model.
- **`weights`** (*`array_like, optional`*) – Weights to multiply `(data-model)` for fit residual.
- **`method`** (*`str, optional`*) – Name of minimization method to use (default is *'leastsq'*).
- **`nan_policy`** (*`{'raise', 'propagate', 'omit'}, optional`*) – What to do when encountering NaNs when fitting Model.
- **`**kwargs`** (*`optional`*) – Keyword arguments to send to minimization routine.

`ModelResult.`**`fit_report`**`(`*modelpars=None*, *show_correl=True*, *min_correl=0.1*, *sort_pars=False*`)`

Return a printable fit report.

The report contains fit statistics and best-fit values with uncertainties and correlations.

**Parameters**

- **`modelpars`** (`Parameters`, *`optional`*) – Known Model Parameters.
- **`show_correl`** (*`bool, optional`*) – Whether to show list of sorted correlations (default is True).
- **`min_correl`** (*`float, optional`*) – Smallest correlation in absolute value to show (default is 0.1).
- **`sort_pars`** (*`callable, optional`*) – Whether to show parameter names sorted in alphanumerical order (default is False). If False, then the parameters will be listed in the

order as they were added to the Parameters dictionary. If callable, then this (one argument) function is used to extract a comparison key from each list element.

**Returns** Multi-line text of fit report.

**Return type** str

`ModelResult.`**`conf_interval`**`(`***kwargs*`)`

Calculate the confidence intervals for the variable parameters.

Confidence intervals are calculated using the `confidence.conf_interval()` function and keyword arguments (***kwargs*) are passed to that function. The result is stored in the `ci_out` attribute so that it can be accessed without recalculating them.

`ModelResult.`**`ci_report`**`(`*with_offset=True*, *ndigits=5*, ***kwargs*`)`

Return a formatted text report of the confidence intervals.

**Parameters**

- **`with_offset`** (`bool, optional`) – Whether to subtract best value from all other values (default is True).
- **`ndigits`** (`int, optional`) – Number of significant digits to show (default is 5).
- **`**kwargs`** (`optional`) – Keyword arguments that are passed to the *conf_interval* function.

**Returns** Text of formatted report on confidence intervals.

**Return type** str

`ModelResult.`**`eval_uncertainty`**`(`*params=None*, *sigma=1*, ***kwargs*`)`

Evaluate the uncertainty of the *model function*.

This can be used to give confidence bands for the model from the uncertainties in the best-fit parameters.

**Parameters**

- **`params`** (`Parameters, optional`) – Parameters, defaults to ModelResult.params.
- **`sigma`** (`float, optional`) – Confidence level, i.e. how many sigma (default is 1).
- **`**kwargs`** (`optional`) – Values of options, independent variables, etcetera.

**Returns** Uncertainty at each value of the model.

**Return type** numpy.ndarray

### Notes

1. This is based on the excellent and clear example from https://www.astro.rug.nl/software/kapteyn/kmpfittutorial.html#confidence-and-prediction-intervals, which references the original work of: J. Wolberg, Data Analysis Using the Method of Least Squares, 2006, Springer
2. The value of sigma is number of *sigma* values, and is converted to a probability. Values of 1, 2, or 3 give probabilities of 0.6827, 0.9545, and 0.9973, respectively. If the sigma value is < 1, it is interpreted as the probability itself. That is, `sigma=1` and `sigma=0.6827` will give the same results, within precision errors.

## Examples

```
>>> out = model.fit(data, params, x=x)
>>> dely = out.eval_uncertainty(x=x)
>>> plt.plot(x, data)
>>> plt.plot(x, out.best_fit)
>>> plt.fill_between(x, out.best_fit-dely,
...                  out.best_fit+dely, color='#888888')
```

`ModelResult.`**`plot`**(*datafmt='o', fitfmt='-', initfmt='--', xlabel=None, ylabel=None, yerr=None, numpoints=None, fig=None, data_kws=None, fit_kws=None, init_kws=None, ax_res_kws=None, ax_fit_kws=None, fig_kws=None, show_init=False, parse_complex='abs'*)

Plot the fit results and residuals using matplotlib.

The method will produce a matplotlib figure (if package available) with both results of the fit and the residuals plotted. If the fit model included weights, errorbars will also be plotted. To show the initial conditions for the fit, pass the argument `show_init=True`.

**Parameters**

- **datafmt** (*str, optional*) – Matplotlib format string for data points.
- **fitfmt** (*str, optional*) – Matplotlib format string for fitted curve.
- **initfmt** (*str, optional*) – Matplotlib format string for initial conditions for the fit.
- **xlabel** (*str, optional*) – Matplotlib format string for labeling the x-axis.
- **ylabel** (*str, optional*) – Matplotlib format string for labeling the y-axis.
- **yerr** (*numpy.ndarray, optional*) – Array of uncertainties for data array.
- **numpoints** (*int, optional*) – If provided, the final and initial fit curves are evaluated not only at data points, but refined to contain *numpoints* points in total.
- **fig** (*matplotlib.figure.Figure, optional*) – The figure to plot on. The default is None, which means use the current pyplot figure or create one if there is none.
- **data_kws** (*dict, optional*) – Keyword arguments passed to the plot function for data points.
- **fit_kws** (*dict, optional*) – Keyword arguments passed to the plot function for fitted curve.
- **init_kws** (*dict, optional*) – Keyword arguments passed to the plot function for the initial conditions of the fit.
- **ax_res_kws** (*dict, optional*) – Keyword arguments for the axes for the residuals plot.
- **ax_fit_kws** (*dict, optional*) – Keyword arguments for the axes for the fit plot.
- **fig_kws** (*dict, optional*) – Keyword arguments for a new figure, if a new one is created.
- **show_init** (*bool, optional*) – Whether to show the initial conditions for the fit (default is False).
- **parse_complex** (*{'abs', 'real', 'imag', 'angle'}, optional*) – How to reduce complex data for plotting. Options are one of: *'abs'* (default), *'real'*, *'imag'*, or *'angle'*, which correspond to the NumPy functions with the same name.

**Returns** A tuple with matplotlib's Figure and GridSpec objects.

**Return type** tuple

**See also:**

***ModelResult.plot_fit*** Plot the fit results using matplotlib.

***ModelResult.plot_residuals*** Plot the fit residuals using matplotlib.

## Notes

The method combines *ModelResult.plot_fit* and *ModelResult.plot_residuals*.

If *yerr* is specified or if the fit model included weights, then *matplotlib.axes.Axes.errorbar* is used to plot the data. If *yerr* is not specified and the fit includes weights, *yerr* set to `1/self.weights`.

If model returns complex data, *yerr* is treated the same way that weights are in this case.

If *fig* is None then *matplotlib.pyplot.figure(**fig_kws)* is called, otherwise *fig_kws* is ignored.

`ModelResult`.**plot_fit**(*ax=None*, *datafmt='o'*, *fitfmt='-'*, *initfmt='--'*, *xlabel=None*, *ylabel=None*, *yerr=None*, *numpoints=None*, *data_kws=None*, *fit_kws=None*, *init_kws=None*, *ax_kws=None*, *show_init=False*, *parse_complex='abs'*)

Plot the fit results using matplotlib, if available.

The plot will include the data points, the initial fit curve (optional, with `show_init=True`), and the best-fit curve. If the fit model included weights or if *yerr* is specified, errorbars will also be plotted.

**Parameters**

- **ax** (*matplotlib.axes.Axes, optional*) – The axes to plot on. The default in None, which means use the current pyplot axis or create one if there is none.
- **datafmt** (*str, optional*) – Matplotlib format string for data points.
- **fitfmt** (*str, optional*) – Matplotlib format string for fitted curve.
- **initfmt** (*str, optional*) – Matplotlib format string for initial conditions for the fit.
- **xlabel** (*str, optional*) – Matplotlib format string for labeling the x-axis.
- **ylabel** (*str, optional*) – Matplotlib format string for labeling the y-axis.
- **yerr** (*numpy.ndarray, optional*) – Array of uncertainties for data array.
- **numpoints** (*int, optional*) – If provided, the final and initial fit curves are evaluated not only at data points, but refined to contain *numpoints* points in total.
- **data_kws** (*dict, optional*) – Keyword arguments passed to the plot function for data points.
- **fit_kws** (*dict, optional*) – Keyword arguments passed to the plot function for fitted curve.
- **init_kws** (*dict, optional*) – Keyword arguments passed to the plot function for the initial conditions of the fit.
- **ax_kws** (*dict, optional*) – Keyword arguments for a new axis, if a new one is created.
- **show_init** (*bool, optional*) – Whether to show the initial conditions for the fit (default is False).

- **parse_complex** (*{'abs', 'real', 'imag', 'angle'}, optional*) – How to reduce complex data for plotting. Options are one of: *'abs'* (default), *'real'*, *'imag'*, or *'angle'*, which correspond to the NumPy functions with the same name.

**Returns**

**Return type** matplotlib.axes.Axes

**See also:**

***ModelResult.plot_residuals*** Plot the fit residuals using matplotlib.

***ModelResult.plot*** Plot the fit results and residuals using matplotlib.

## Notes

For details about plot format strings and keyword arguments see documentation of *matplotlib.axes.Axes.plot*.

If *yerr* is specified or if the fit model included weights, then *matplotlib.axes.Axes.errorbar* is used to plot the data. If *yerr* is not specified and the fit includes weights, *yerr* set to `1/self.weights`.

If model returns complex data, *yerr* is treated the same way that weights are in this case.

If *ax* is None then *matplotlib.pyplot.gca(**ax_kws)* is called.

ModelResult.**plot_residuals**(*ax=None*, *datafmt='o'*, *yerr=None*, *data_kws=None*, *fit_kws=None*, *ax_kws=None*, *parse_complex='abs'*)

Plot the fit residuals using matplotlib, if available.

If *yerr* is supplied or if the model included weights, errorbars will also be plotted.

**Parameters**

- **ax** (*matplotlib.axes.Axes, optional*) – The axes to plot on. The default in None, which means use the current pyplot axis or create one if there is none.
- **datafmt** (*str, optional*) – Matplotlib format string for data points.
- **yerr** (*numpy.ndarray, optional*) – Array of uncertainties for data array.
- **data_kws** (*dict, optional*) – Keyword arguments passed to the plot function for data points.
- **fit_kws** (*dict, optional*) – Keyword arguments passed to the plot function for fitted curve.
- **ax_kws** (*dict, optional*) – Keyword arguments for a new axis, if a new one is created.
- **parse_complex** (*{'abs', 'real', 'imag', 'angle'}, optional*) – How to reduce complex data for plotting. Options are one of: *'abs'* (default), *'real'*, *'imag'*, or *'angle'*, which correspond to the NumPy functions with the same name.

**Returns**

**Return type** matplotlib.axes.Axes

**See also:**

***ModelResult.plot_fit*** Plot the fit results using matplotlib.

***ModelResult.plot*** Plot the fit results and residuals using matplotlib.

### Notes

For details about plot format strings and keyword arguments see documentation of *matplotlib.axes.Axes.plot*.

If *yerr* is specified or if the fit model included weights, then *matplotlib.axes.Axes.errorbar* is used to plot the data. If *yerr* is not specified and the fit includes weights, *yerr* set to `1/self.weights`.

If model returns complex data, *yerr* is treated the same way that weights are in this case.

If *ax* is None then *matplotlib.pyplot.gca(**ax_kws)* is called.

## 7.3.2 `ModelResult` attributes

**aic**
: Floating point best-fit Akaike Information Criterion statistic (see *MinimizerResult – the optimization result*).

**best_fit**
: numpy.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.

**bic**
: Floating point best-fit Bayesian Information Criterion statistic (see *MinimizerResult – the optimization result*).

**chisqr**
: Floating point best-fit chi-square statistic (see *MinimizerResult – the optimization result*).

**ci_out**
: Confidence interval data (see *Calculation of confidence intervals*) or `None` if the confidence intervals have not been calculated.

**covar**
: numpy.ndarray (square) covariance matrix returned from fit.

**data**
: numpy.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**
: numpy.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. See *Using a Iteration Callback 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()*.

**call_kws**
Dict of keyword arguments actually send to underlying solver with *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 parameters in fit.

**nvarys**
Integer number of independent, freely varying variables in fit.

**params**
Parameters used in fit; will contain the best-fit values.

**redchi**
Floating point reduced chi-square statistic (see *MinimizerResult – the optimization result*).

**residual**
numpy.ndarray for residual.

**scale_covar**
Boolean flag for whether to automatically scale covariance matrix.

**success**
Boolean value of whether fit succeeded.

**weights**
numpy.ndarray (or `None`) of weighting values to be used in fit. If not `None`, it will be used as a multiplicative factor of the residual array, so that `weights*(data - fit)` is minimized in the least-squares sense.

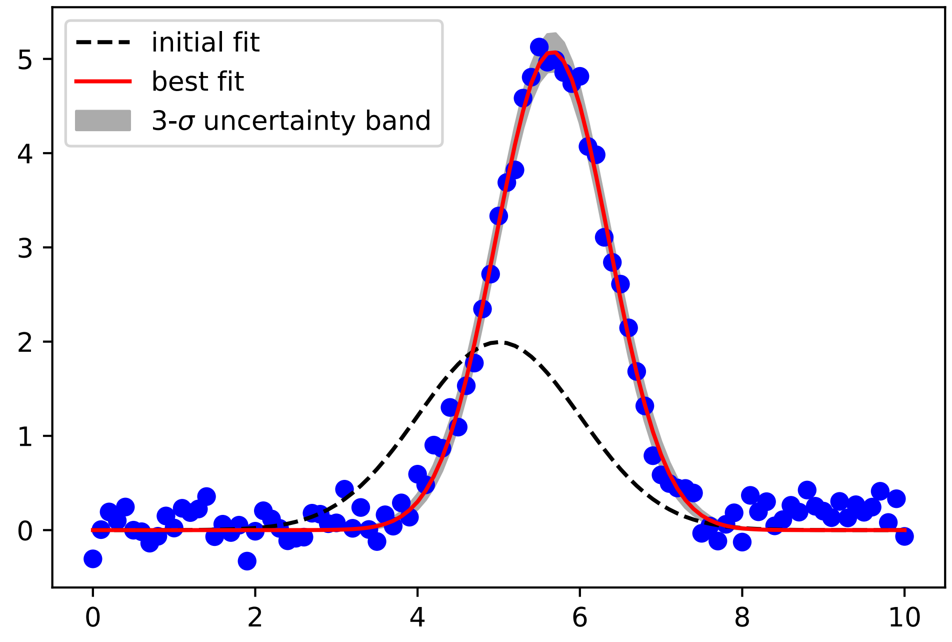
## 7.3.3 Calculating uncertainties in the model function

We return to the first example above and ask not only for the uncertainties in the fitted parameters but for the range of values that those uncertainties mean for the model function itself. We can use the *ModelResult.eval_uncertainty()* method of the model result object to evaluate the uncertainty in the model with a specified level for $\sigma$.

That is, adding:

```
dely = result.eval_uncertainty(sigma=3)
plt.fill_between(x, result.best_fit-dely, result.best_fit+dely, color="#ABABAB",
                 label='3-$\sigma$ uncertainty band')
```

to the example fit to the Gaussian at the beginning of this chapter will give 3-$\sigma$ bands for the best-fit Gaussian, and produce the figure below.



### 7.3.4 Saving and Loading ModelResults

New in version 0.9.8.

As with saving models (see section *Saving and Loading Models*), it is sometimes desirable to save a `ModelResult`, either for later use or to organize and compare different fit results. Lmfit provides a `save_modelresult()` function that will save a `ModelResult` to a file. There is also a companion `load_modelresult()` function that can read this file and reconstruct a `ModelResult` from it.

As discussed in section *Saving and Loading Models*, there are challenges to saving model functions that may make it difficult to restore a saved a `ModelResult` in a way that can be used to perform a fit. Use of the optional `funcdefs` argument is generally the most reliable way to ensure that a loaded `ModelResult` can be used to evaluate the model function or redo the fit.

**`save_modelresult`** (*modelresult*, *fname*)
Save a ModelResult to a file.

**Parameters**

- **`modelresult`** (`ModelResult`) – ModelResult to be saved.
- **`fname`** (`str`) – Name of file for saved ModelResult.

**`load_modelresult`** (*fname*, *funcdefs=None*)
Load a saved ModelResult from a file.

**Parameters**

- **`fname`** (`str`) – Name of file containing saved ModelResult.
- **`funcdefs`** (`dict, optional`) – Dictionary of custom function names and definitions.

**Returns** ModelResult object loaded from file.

**Return type** *ModelResult*

An example of saving a `ModelResult` is:

```
# <examples/doc_model_savemodelresult.py>
import numpy as np

from lmfit.model import save_modelresult
from lmfit.models import GaussianModel

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

gmodel = GaussianModel()
result = gmodel.fit(y, x=x, amplitude=5, center=5, sigma=1)

save_modelresult(result, 'gauss_modelresult.sav')

print(result.fit_report())
# <end examples/doc_model_savemodelresult.py>
```

To load that later, one might do:

```
# <examples/doc_model_loadmodelresult.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.model import load_modelresult

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

result = load_modelresult('gauss_modelresult.sav')
print(result.fit_report())

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

## 7.4 Composite Models : adding (or multiplying) Models

One of the more interesting features of the *Model* class is that Models can be added together or combined with basic algebraic operations (add, subtract, multiply, and divide) to give a composite model. The composite model will have parameters from each of the component models, with all parameters being available to influence the whole model. This ability to combine models 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, and build a model of a Gaussian plus a line, as to model a peak with a background. For such a simple problem, we could just build a model that included both components:

```
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:

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

But we already had a function for a gaussian function, and maybe we'll discover that a linear background isn't sufficient which would mean the model function would have to be changed.

Instead, lmfit allows models to be combined into a *CompositeModel*. As an alternative to including a linear background in our model function, we could define a linear function:

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

and build a composite model with just:

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

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

```
# <examples/doc_model_two_components.py>
import matplotlib.pyplot as plt
from numpy import exp, loadtxt, pi, sqrt

from lmfit import Model

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):
    """a line"""
    return slope*x + intercept


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

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

print(result.fit_report())

plt.plot(x, y, 'bo')
plt.plot(x, result.init_fit, 'k--', label='initial fit')
plt.plot(x, result.best_fit, 'r-', label='best fit')
plt.legend(loc='best')
plt.show()
# <end examples/doc_model_two_components.py>
```

which prints out the results:

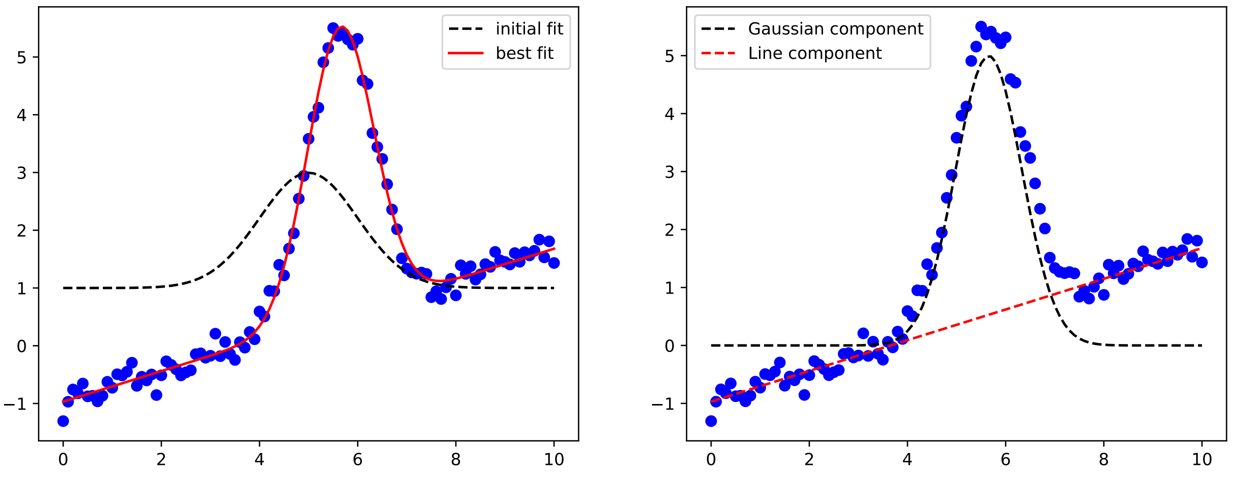
```
[[Model]]
    (Model(gaussian) + Model(line))
```

(continues on next page)

(continued from previous page)

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 44
    # data points      = 101
    # variables        = 5
    chi-square         = 2.57855517
    reduced chi-square = 0.02685995
    Akaike info crit   = -360.457020
    Bayesian info crit = -347.381417
[[Variables]]
    amp:        8.45931062 +/- 0.12414515 (1.47%) (init = 5)
    cen:        5.65547873 +/- 0.00917678 (0.16%) (init = 5)
    wid:        0.67545524 +/- 0.00991686 (1.47%) (init = 1)
    slope:      0.26484404 +/- 0.00574892 (2.17%) (init = 0)
    intercept: -0.96860202 +/- 0.03352202 (3.46%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(slope, intercept) = -0.795
    C(amp, wid)         =  0.666
    C(amp, intercept)   = -0.222
    C(amp, slope)       = -0.169
    C(cen, slope)       = -0.162
    C(wid, intercept)   = -0.148
    C(cen, intercept)   =  0.129
    C(wid, slope)       = -0.113
```

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. The figure on the right shows again the data in blue dots, the Gaussian component as a black dashed line and the linear component as a red dashed line. It is created using the following code:

```
comps = result.eval_components()
plt.plot(x, y, 'bo')
plt.plot(x, comps['gaussian'], 'k--', label='Gaussian component')
plt.plot(x, comps['line'], 'r--', label='Line component')
```

The components were generated after the fit using the *ModelResult.eval_components()* method of the `result`, which returns a dictionary of the components, using keys of the model name (or `prefix` if that is set). This will use the parameter values in `result.params` and the independent variables (`x`) used during the fit. Note that while the *ModelResult* held in `result` does store the best parameters and the best estimate of the model in `result.best_fit`, the original model and parameters in `pars` are left unaltered.

You can apply this composite 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 = linspace(-5, 25, 3001)
predicted = mod.eval(result.params, x=xwide)
```

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.

**`class CompositeModel`**(*left*, *right*, *op*[, ***kws*])

Combine two models (*left* and *right*) with binary operator (*op*).

Normally, one does not have to explicitly create a *CompositeModel*, but can use normal Python operators `+`, `-`, `*`, and `/` to combine components as in:

```
>>> mod = Model(fcn1) + Model(fcn2) * Model(fcn3)
```

**Parameters**

- **`left`** (`Model`) – Left-hand model.
- **`right`** (`Model`) – Right-hand model.
- **`op`** (*`callable binary operator`*) – Operator to combine *left* and *right* models.
- **`**kws`** (*`optional`*) – Additional keywords are passed to *Model* when creating this new model.

### Notes

The two models must use the same independent variable.

Note that when using built-in Python binary operators, a *`CompositeModel`* will automatically be constructed for you. That is, doing:

```
mod = Model(fcn1) + Model(fcn2) * Model(fcn3)
```

will create a *`CompositeModel`*. Here, `left` will be `Model(fcn1)`, `op` will be `operator.add()`, and `right` will be another CompositeModel that has a `left` attribute of `Model(fcn2)`, an `op` of `operator.mul()`, and a `right` of `Model(fcn3)`.

To use a binary operator other than `+`, `-`, `*`, or `/` you can explicitly create a *`CompositeModel`* with the appropriate binary operator. For example, to convolve two models, you could define a simple convolution function, perhaps as:

```
import numpy as np

def convolve(dat, kernel):
    """simple convolution of two arrays"""
    npts = min(len(dat), len(kernel))
    pad = np.ones(npts)
    tmp = np.concatenate((pad*dat[0], dat, pad*dat[-1]))
    out = np.convolve(tmp, kernel, mode='valid')
    noff = int((len(out) - npts) / 2)
    return (out[noff:])[:npts]
```

which extends the data in both directions so that the convolving kernel function gives a valid result over the data range. Because this function takes two array arguments and returns an array, it can be used as the binary operator. A full script using this technique is here:

```
# <examples/doc_model_composite.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit import CompositeModel, Model
from lmfit.lineshapes import gaussian, step

# create data from broadened step
x = np.linspace(0, 10, 201)
y = step(x, amplitude=12.5, center=4.5, sigma=0.88, form='erf')
np.random.seed(0)
y = y + np.random.normal(scale=0.35, size=x.size)


def jump(x, mid):
    """Heaviside step function."""
    o = np.zeros(x.size)
    imid = max(np.where(x <= mid)[0])
    o[imid:] = 1.0
    return o


def convolve(arr, kernel):
    """Simple convolution of two arrays."""
    npts = min(arr.size, kernel.size)
    pad = np.ones(npts)
    tmp = np.concatenate((pad*arr[0], arr, pad*arr[-1]))
    out = np.convolve(tmp, kernel, mode='valid')
    noff = int((len(out) - npts) / 2)
    return out[noff:noff+npts]


# create Composite Model using the custom convolution operator
mod = CompositeModel(Model(jump), Model(gaussian), convolve)
pars = mod.make_params(amplitude=1, center=3.5, sigma=1.5, mid=5.0)

# 'mid' and 'center' should be completely correlated, and 'mid' is
# used as an integer index, so a very poor fit variable:
pars['mid'].vary = False

# fit this model to data array y
result = mod.fit(y, params=pars, x=x)

print(result.fit_report())

# generate components
comps = result.eval_components(x=x)

# plot results
fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))

axes[0].plot(x, y, 'bo')
axes[0].plot(x, result.init_fit, 'k--', label='initial fit')
axes[0].plot(x, result.best_fit, 'r-', label='best fit')
```

(continues on next page)

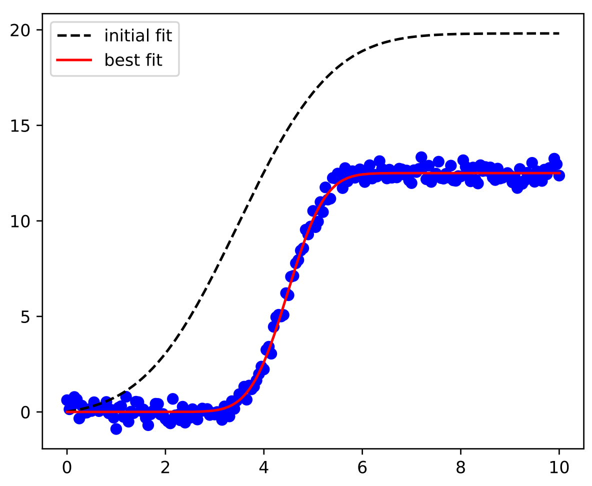
(continued from previous page)

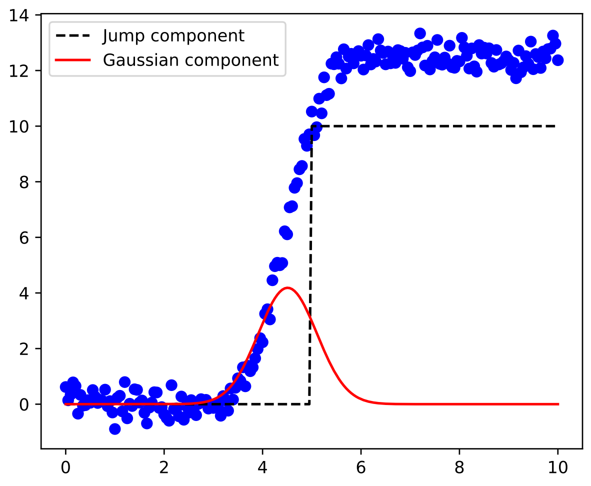
```
axes[0].legend(loc='best')

axes[1].plot(x, y, 'bo')
axes[1].plot(x, 10*comps['jump'], 'k--', label='Jump component')
axes[1].plot(x, 10*comps['gaussian'], 'r-', label='Gaussian component')
axes[1].legend(loc='best')

plt.show()
# <end examples/doc_model_composite.py>
```

which prints out the results:

```
[[Model]]
    (Model(jump) <function convolve at 0x12afb1f70> Model(gaussian))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 25
    # data points      = 201
    # variables        = 3
    chi-square         = 24.7562335
    reduced chi-square = 0.12503148
    Akaike info crit   = -414.939746
    Bayesian info crit = -405.029832
[[Variables]]
    mid:         5 (fixed)
    amplitude:  0.62508459 +/- 0.00189732 (0.30%) (init = 1)
    center:     4.50853671 +/- 0.00973231 (0.22%) (init = 3.5)
    sigma:      0.59576118 +/- 0.01348582 (2.26%) (init = 1.5)
[[Correlations]] (unreported correlations are < 0.100)
    C(amplitude, center) =  0.329
    C(amplitude, sigma)  =  0.268
```

and shows the plots:





Using composite models with built-in or custom operators allows you to build complex models from testable sub-components.

CHAPTER

EIGHT

# BUILT-IN FITTING MODELS IN THE `MODELS` MODULE

Lmfit provides several built-in 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 Gaussian, Lorentzian, and Exponential that are used in a wide range of scientific domains. In fact, all the models are 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 *`CompositeModel`*. 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 use them 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
```

Almost 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 intensity (or area of) a peak or function and a `sigma` parameter that gives a characteristic width.

After a list of built-in models, a few examples of their use are given.

## 8.1 Peak-like models

There are many peak-like models available. These include *`GaussianModel`*, *`LorentzianModel`*, *`VoigtModel`*, *`PseudoVoigtModel`*, and some less commonly used variations. Most of these models are *unit-normalized* and share the same parameter names so that you can easily switch between models and interpret the results. The `amplitude` parameter is the multiplicative factor for the unit-normalized peak lineshape, and so will represent the strength of that peak or the area under that curve. The `center` parameter will be the centroid `x` value. The `sigma` parameter is the characteristic width of the peak, with many functions using $(x-\mu)/\sigma$ where $\mu$ is the centroid value. Most of these peak functions will have two additional parameters derived from and constrained by the other parameters. The first of these is `fwhm` which will hold the estimated "Full Width at Half Max" for the peak, which is often easier to compare between different models than `sigma`. The second of these is `height` which will contain the maximum value of the peak, typically the value at $x = \mu$. Finally, each of these models has a `guess()` method that uses data to make a fairly crude but usually sufficient guess for the value of `amplitude`, `center`, and `sigma`, and sets a lower bound of 0 on the value of `sigma`.

### 8.1.1 `GaussianModel`

**`class GaussianModel`**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a Gaussian or normal distribution lineshape.

The model has three Parameters: *amplitude*, *center*, and *sigma*. In addition, parameters *fwhm* and *height* are included as constraints to report full width at half maximum and maximum peak height, respectively.

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

where the parameter *amplitude* corresponds to $A$, *center* to $\mu$, and *sigma* to $\sigma$. The full width at half maximum is $2\sigma\sqrt{2\ln 2}$, approximately $2.3548\sigma$.

For more information, see: https://en.wikipedia.org/wiki/Normal_distribution

**Parameters**

- **`independent_vars`** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **`prefix`** (*`str, optional`*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **`nan_policy`** (*`{'raise', 'propagate', 'omit'}, optional`*) – How to handle NaN and missing values in data. See Notes below.
- **`**kwargs`** (*`optional`*) – Keyword arguments to pass to `Model`.

#### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:

- *'raise'* : raise a *ValueError* (default)
- *'propagate'* : do nothing
- *'omit'* : drop missing data

### 8.1.2 `LorentzianModel`

**`class LorentzianModel`**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a Lorentzian or Cauchy-Lorentz distribution function.

The model has three Parameters: *amplitude*, *center*, and *sigma*. In addition, parameters *fwhm* and *height* are included as constraints to report full width at half maximum and maximum peak height, respectively.

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

where the parameter *amplitude* corresponds to $A$, *center* to $\mu$, and *sigma* to $\sigma$. The full width at half maximum is $2\sigma$.

For more information, see: https://en.wikipedia.org/wiki/Cauchy_distribution

**Parameters**

- **`independent_vars`** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).

- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.3 `SplitLorentzianModel`

**class SplitLorentzianModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a Lorentzian or Cauchy-Lorentz distribution function.

The model has four parameters: *amplitude*, *center*, *sigma*, and *sigma_r*. In addition, parameters *fwhm* and *height* are included as constraints to report full width at half maximum and maximum peak height, respectively.

'Split' means that the width of the distribution is different between left and right slopes.

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

where the parameter *amplitude* corresponds to $A$, *center* to $\mu$, *sigma* to $\sigma$, *sigma_l* to $\sigma_l$, and $H(x)$ is a Heaviside step function:

$$H(x) = 0 | x < 0, 1 | x \ge 0$$

The full width at half maximum is $\sigma_l + \sigma_r$. Just as with the Lorentzian model, integral of this function from *-.inf* to *+.inf* equals to *amplitude*.

For more information, see: https://en.wikipedia.org/wiki/Cauchy_distribution

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.1.4 `VoigtModel`

**`class VoigtModel`**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a Voigt distribution function.

The model has four Parameters: *amplitude*, *center*, *sigma*, and *gamma*. By default, *gamma* is constrained to have a value equal to *sigma*, though it can be varied independently. In addition, parameters *fwhm* and *height* are included as constraints to report full width at half maximum and maximum peak height, respectively. 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 complementary error function. As above, *amplitude* corresponds to $A$, *center* to $\mu$, and *sigma* to $\sigma$. The parameter *gamma* corresponds to $\gamma$. If *gamma* is kept at the default value (constrained to *sigma*), the full width at half maximum is approximately $3.6013\sigma$.

For more information, see: https://en.wikipedia.org/wiki/Voigt_profile

**Parameters**

- **`independent_vars`** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **`prefix`** (*`str, optional`*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **`nan_policy`** (*`{'raise', 'propagate', 'omit'}, optional`*) – How to handle NaN and missing values in data. See Notes below.
- **`**kwargs`** (*`optional`*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.5 `PseudoVoigtModel`

**class PseudoVoigtModel**(*independent_vars=['x']*, *prefix=''*, *nan_policy='raise'*, ***kwargs*)

A model based on a pseudo-Voigt distribution function.

This is a weighted sum of a Gaussian and Lorentzian distribution function that share values for *amplitude* ($A$), *center* ($\mu$), and full width at half maximum *fwhm* (and so has constrained values of *sigma* ($\sigma$) and *height* (maximum peak height). The parameter *fraction* ($\alpha$) controls the relative weight of the Gaussian and Lorentzian components, giving the full definition of:

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

where $\sigma_g = \sigma/\sqrt{2\ln 2}$ so that the full width at half maximum of each component and of the sum is $2\sigma$. The `guess()` function always sets the starting value for *fraction* at 0.5.

For more information, see: https://en.wikipedia.org/wiki/Voigt_profile#Pseudo-Voigt_Approximation

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str`, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}`, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

#### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.6 `MoffatModel`

**class MoffatModel**(*independent_vars=['x']*, *prefix=''*, *nan_policy='raise'*, ***kwargs*)

A model based on the Moffat distribution function.

The model has four Parameters: *amplitude* ($A$), *center* ($\mu$), a width parameter *sigma* ($\sigma$), and an exponent *beta* ($\beta$). In addition, parameters *fwhm* and *height* are included as constraints to report full width at half maximum and maximum peak height, respectively.

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

the full width at half maximum is $2\sigma\sqrt{2^{1/\beta}-1}$. The `guess()` function always sets the starting value for *beta* to 1.

Note that for ($\beta = 1$) the Moffat has a Lorentzian shape. For more information, see: https://en.wikipedia.org/wiki/Moffat_distribution

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str`, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}`, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.7 `Pearson7Model`

**class Pearson7Model**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a Pearson VII distribution.

The model has four parameters: *amplitude* ($A$), *center* ($\mu$), *sigma* ($\sigma$), and *exponent* ($m$). In addition, parameters *fwhm* and *height* are included as constraints to report estimates for the full width at half maximum and maximum peak height, respectively.

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

where $\beta$ is the beta function (see scipy.special.beta). The `guess()` function always gives a starting value for *exponent* of 1.5. In addition, parameters *fwhm* and *height* are included as constraints to report full width at half maximum and maximum peak height, respectively.

For more information, see: https://en.wikipedia.org/wiki/Pearson_distribution#The_Pearson_type_VII_distribution

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str`, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}`, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.1.8 `StudentsTModel`

**class StudentsTModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a Student's t-distribution function.

The model has three Parameters: *amplitude* ($A$), *center* ($\mu$), and *sigma* ($\sigma$). In addition, parameters *fwhm* and *height* are included as constraints to report full width at half maximum and maximum peak height, respectively.

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

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

For more information, see: https://en.wikipedia.org/wiki/Student%27s_t-distribution

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str, optional`*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}, optional`*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*`optional`*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.1.9 `BreitWignerModel`

**class BreitWignerModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a Breit-Wigner-Fano function.

The model has four Parameters: *amplitude* ($A$), *center* ($\mu$), *sigma* ($\sigma$), and *q* ($q$).

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

For more information, see: https://en.wikipedia.org/wiki/Fano_resonance

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.10 `LognormalModel`

**class LognormalModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on the Log-normal distribution function.

The modal has three Parameters *amplitude* ($A$), *center* ($\mu$), and *sigma* ($\sigma$). In addition, parameters *fwhm* and *height* are included as constraints to report estimates of full width at half maximum and maximum peak height, respectively.

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

For more information, see: https://en.wikipedia.org/wiki/Lognormal

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.11 `DampedOscillatorModel`

**class DampedOscillatorModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on the Damped Harmonic Oscillator Amplitude.

The model has three Parameters: *amplitude* ($A$), *center* ($\mu$), and *sigma* ($\sigma$). In addition, the parameter *height* is included as a constraint to report the maximum peak height.

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

For more information, see: https://en.wikipedia.org/wiki/Harmonic_oscillator#Amplitude_part

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str, optional`*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}, optional`*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*`optional`*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.12 `DampedHarmonicOscillatorModel`

**class DampedHarmonicOscillatorModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a variation of the Damped Harmonic Oscillator.

The model follows the definition given in DAVE/PAN (see: https://www.ncnr.nist.gov/dave) and has four Parameters: *amplitude* ($A$), *center* ($\mu$), *sigma* ($\sigma$), and *gamma* ($\gamma$). In addition, parameters *fwhm* and *height* are included as constraints to report estimates for full width at half maximum and maximum peak height, respectively.

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

where $\gamma = kT$, $k$ is the Boltzmann constant in $evK^-1$, and $T$ is the temperature in $K$.

For more information, see: https://en.wikipedia.org/wiki/Harmonic_oscillator

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is [‘x’]).
- **prefix** (*`str, optional`*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}, optional`*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*`optional`*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *‘raise’* : raise a *ValueError* (default)
   - *‘propagate’* : do nothing
   - *‘omit’* : drop missing data

### 8.1.13 `ExponentialGaussianModel`

**class ExponentialGaussianModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model of an Exponentially modified Gaussian distribution.

The model has four Parameters: *amplitude* ($A$), *center* ($\mu$), *sigma* ($\sigma$), and *gamma* ($\gamma$).

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

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

For more information, see: https://en.wikipedia.org/wiki/Exponentially_modified_Gaussian_distribution

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is [‘x’]).
- **prefix** (*`str, optional`*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}, optional`*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*`optional`*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.14 `SkewedGaussianModel`

**class SkewedGaussianModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A skewed Gaussian model, using a skewed normal distribution.

The model has four Parameters: *amplitude* ($A$), *center* ($\mu$), *sigma* ($\sigma$), and *gamma* ($\gamma$).

$$f(x; A, \mu, \sigma, \gamma) = \frac{A}{\sigma\sqrt{2\pi}} e^{[-(x-\mu)^2/2\sigma^2]} \Big\{ 1 + \operatorname{erf}\big[\frac{\gamma(x-\mu)}{\sigma\sqrt{2}}\big] \Big\}$$

where `erf()` is the error function.

For more information, see: https://en.wikipedia.org/wiki/Skew_normal_distribution

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.15 `SkewedVoigtModel`

**class SkewedVoigtModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A skewed Voigt model, modified using a skewed normal distribution.

The model has five Parameters *amplitude* ($A$), *center* ($\mu$), *sigma* ($\sigma$), and *gamma* ($\gamma$), as usual for a Voigt distribution, and adds a new Parameter *skew*.

$$f(x; A, \mu, \sigma, \gamma, \mathrm{skew}) = \mathrm{Voigt}(\mathrm{x}; \mathrm{A}, \mu, \sigma, \gamma) \Big\{ 1 + \operatorname{erf}\big[\frac{\mathrm{skew}(\mathrm{x}-\mu)}{\sigma\sqrt{2}}\big] \Big\}$$

where `erf()` is the error function.

For more information, see: https://en.wikipedia.org/wiki/Skew_normal_distribution

**Parameters**

- **independent_vars** (list of str, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.16 `ThermalDistributionModel`

**class ThermalDistributionModel**(*independent_vars=['x'], prefix='', nan_policy='raise', form='bose', **kwargs*)

Return a thermal distribution function.

Variable *form* defines the kind of distribution as below with three Parameters: *amplitude* ($A$), *center* ($x_0$), and *kt* ($kt$). The following distributions are available:

- *'bose'* : Bose-Einstein distribution (default)
- *'maxwell'* : Maxwell-Boltzmann distribution
- *'fermi'* : Fermi-Dirac distribution

The functional forms are defined as:

$$\begin{aligned} f(x; A, x_0, kt, \mathrm{form} = '\mathrm{bose}') &= \frac{1}{A\exp(\frac{x-x_0}{kt}) - 1} \\ f(x; A, x_0, kt, \mathrm{form} = '\mathrm{maxwell}') &= \frac{1}{A\exp(\frac{x-x_0}{kt})} \\ f(x; A, x_0, kt, \mathrm{form} = '\mathrm{fermi}') &= \frac{1}{A\exp(\frac{x-x_0}{kt}) + 1}] \end{aligned}$$

### Notes

- *kt* should be defined in the same units as *x* ($k_B = 8.617 \times 10^{-5}$ eV/K).
- set $kt < 0$ to implement the energy loss convention common in scattering research.

For more information, see: http://hyperphysics.phy-astr.gsu.edu/hbase/quantum/disfcn.html

**Parameters**

- **independent_vars** (list of str, optional) – Arguments to the model function that are independent variables default is ['x']).

- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.1.17 `DoniachModel`

**class DoniachModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model of an Doniach Sunjic asymmetric lineshape.

This model is used in photo-emission and has four Parameters: *amplitude* ($A$), *center* ($\mu$), *sigma* ($\sigma$), and *gamma* ($\gamma$). In addition, parameter *height* is included as a constraint to report maximum peak height.

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

For more information, see: https://www.casaxps.com/help_manual/line_shapes.htm

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.2 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 a composite model.

### 8.2.1 `ConstantModel`

**class ConstantModel**(*independent_vars=['x']*, *prefix=''*, *nan_policy='raise'*, ***kwargs*)

Constant model, with a single Parameter: *c*.

Note that 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 Parameter that will be varied in the fit (by default, of course).

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str`, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

#### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.2.2 `LinearModel`

**class LinearModel**(*independent_vars=['x']*, *prefix=''*, *nan_policy='raise'*, ***kwargs*)

Linear model, with two Parameters: *intercept* and *slope*.

Defined as:

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

with *slope* for $m$ and *intercept* for $b$.

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str`, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.

- `**kwargs` (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.2.3 `QuadraticModel`

**class QuadraticModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A quadratic model, with three Parameters: $a$, $b$, and $c$.

Defined as:

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

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- `**kwargs` (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.2.4 `PolynomialModel`

**class PolynomialModel**(*degree=7, independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A polynomial model with up to 7 Parameters, specified by *degree*.

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

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.

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str`, `optional`*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}`, `optional`*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*`optional`*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.3 Periodic Models

These models correspond to periodic functions.

### 8.3.1 `SineModel`

**class SineModel**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a sinusoidal lineshape.

The model has three Parameters: *amplitude*, *frequency*, and *shift*.

$$f(x; A, \phi, f) = A \sin(fx + \phi)$$

where the parameter *amplitude* corresponds to $A$, *frequency* to $f$, and *shift* to $\phi$. All are constrained to be non-negative, and *shift* additionally to be smaller than $2\pi$.

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str`, `optional`*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}`, `optional`*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*`optional`*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.4 Step-like models

Two models represent step-like functions, and share many characteristics.

### 8.4.1 `StepModel`

**class StepModel**(*independent_vars=['x'], prefix='', nan_policy='raise', form='linear', **kwargs*)

A model based on a Step function.

The model has three Parameters: *amplitude* ($A$), *center* ($\mu$), and *sigma* ($\sigma$).

There are four choices for *form*:

- *'linear'* (default)
- *'atan'* or *'arctan'* for an arc-tangent function
- *'erf'* for an error function
- *'logistic'* for a logistic function (for more information, see: https://en.wikipedia.org/wiki/Logistic_function)

The step function starts with a value 0 and ends with a value of $A$ rising to $A/2$ at $\mu$, with $\sigma$ setting the characteristic width. The functional forms are defined as:

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

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

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (`str, optional`) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (`{'raise', 'propagate', 'omit'}, optional`) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (`optional`) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.4.2 `RectangleModel`

**class RectangleModel**(*independent_vars=['x'], prefix='', nan_policy='raise', form='linear', **kwargs*)

A model based on a Step-up and Step-down function.

The model has five Parameters: *amplitude* ($A$), *center1* ($\mu_1$), *center2* ($\mu_2$), *sigma1* ($\sigma_1$), and *sigma2* ($\sigma_2$).

There are four choices for *form*, which is used for both the Step up and the Step down:

- *'linear'* (default)
- *'atan'* or *'arctan'* for an arc-tangent function
- *'erf'* for an error function
- *'logistic'* for a logistic function (for more information, see: https://en.wikipedia.org/wiki/Logistic_function)

The function starts with a value 0 and transitions to a value of $A$, taking the value $A/2$ at $\mu_1$, with $\sigma_1$ setting the characteristic width. The function then transitions again to the value $A/2$ at $\mu_2$, with $\sigma_2$ setting the characteristic width. The functional forms are defined as:

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

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

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.5 Exponential and Power law models

### 8.5.1 `ExponentialModel`

**`class ExponentialModel`**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on an exponential decay function.

The model has two Parameters: *amplitude* ($A$) and *decay* ($\tau$) and is defined as:

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

For more information, see: https://en.wikipedia.org/wiki/Exponential_decay

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*`str, optional`*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*`{'raise', 'propagate', 'omit'}, optional`*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*`optional`*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

### 8.5.2 `PowerLawModel`

**`class PowerLawModel`**(*independent_vars=['x'], prefix='', nan_policy='raise', **kwargs*)

A model based on a Power Law.

The model has two Parameters: *amplitude* ($A$) and *exponent* ($k$) and is defined as:

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

For more information, see: https://en.wikipedia.org/wiki/Power_law

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.6 Two dimensional Peak-like models

The one example of a two-dimensional peak is a two-dimensional Gaussian.

### 8.6.1 `Gaussian2dModel`

**class Gaussian2dModel**(*independent_vars=['x', 'y'], prefix='', nan_policy='raise', **kwargs*)

A model based on a two-dimensional Gaussian function.

The model has two independent variables *x* and *y* and five Parameters: *amplitude*, *centerx*, *sigmax*, *centery*, and *sigmay*. In addition, parameters *fwhmx*, *fwhmy*, and *height* are included as constraints to report the maximum peak height and the two full width at half maxima, respectively.

$$f(x, y; A, \mu_x, \sigma_x, \mu_y, \sigma_y) = Ag(x; A = 1, \mu_x, \sigma_x)g(y; A = 1, \mu_y, \sigma_y)$$

where subfunction $g(x; A, \mu, \sigma)$ is a Gaussian lineshape:

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

**Parameters**

- **independent_vars** (`list` of `str`, optional) – Arguments to the model function that are independent variables default is ['x', 'y']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to `Model`.

### Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

## 8.7 User-defined Models

As shown in the previous chapter (*Modeling Data and Curve Fitting*), it is fairly straightforward to build fitting models from parametrized Python functions. The number of model classes listed so far in the present chapter should make it clear that this process is not too difficult. Still, it is sometimes desirable to build models from a user-supplied function. This may be especially true if model-building is built-in to some larger library or application for fitting in which the user may not be able to easily build and use a new model from Python code.

The `ExpressionModel` allows a model to be built from a user-supplied expression. This uses the asteval module also used for mathematical constraints as discussed in *Using Mathematical Constraints*.

### 8.7.1 `ExpressionModel`

**class ExpressionModel**(*expr*, *independent_vars=None*, *init_script=None*, *nan_policy='raise'*, ***kws*)
ExpressionModel class.

Generate a model from user-supplied expression.

**Parameters**

- **expr** (`str`) – Mathematical expression for model.
- **independent_vars** (`list` of `str` or None, optional) – Variable names to use as independent variables.
- **init_script** (*`str or None, optional`*) – Initial script to run in asteval interpreter.
- **nan_policy** (*`{'raise, 'propagate', 'omit'}, optional`*) – How to handle NaN and missing values in data. See Notes below.
- ****kws** (*`optional`*) – Keyword arguments to pass to `Model`.

### Notes

1. each instance of ExpressionModel will create and use its own version of an asteval interpreter.
2. *prefix* is **not supported** for ExpressionModel.

3. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

Since the point of this model is that an arbitrary expression will be supplied, the determination of what are the parameter names for the model happens when the model is created. To do this, the expression is parsed, and all symbol names are found. Names that are already known (there are over 500 function and value names in the asteval namespace, including most Python built-ins, more than 200 functions inherited from NumPy, and more than 20 common lineshapes defined in the `lineshapes` module) are not converted to parameters. Unrecognized names are expected to be names of either parameters or independent variables. If `independent_vars` is the default value of `None`, and if the expression contains a variable named `x`, that will be used as the independent variable. Otherwise, `independent_vars` must be given.

For example, if one creates an *`ExpressionModel`* as:

```
from lmfit.models import ExpressionModel

mod = ExpressionModel('off + amp * exp(-x/x0) * sin(x*phase)')
```

The name `exp` will be recognized as the exponent function, so the model will be interpreted to have parameters named `off`, `amp`, `x0` and `phase`. In addition, `x` will be assumed to be the sole independent variable. In general, there is no obvious way to set default parameter values or parameter hints for bounds, so this will have to be handled explicitly.

To evaluate this model, you might do the following:

```
from numpy import exp, linspace, sin

x = linspace(0, 10, 501)
params = mod.make_params(off=0.25, amp=1.0, x0=2.0, phase=0.04)
y = mod.eval(params, x=x)
```

While many custom models can be built with a single line expression (especially since the names of the lineshapes like `gaussian`, `lorentzian` and so on, as well as many NumPy functions, are available), more complex models will inevitably require multiple line functions. You can include such Python code with the `init_script` argument. The text of this script is evaluated when the model is initialized (and before the actual expression is parsed), so that you can define functions to be used in your expression.

As a probably unphysical example, to make a model that is the derivative of a Gaussian function times the logarithm of a Lorentzian function you may could to define this in a script:

```
script = """
def mycurve(x, amp, cen, sig):
    loren = lorentzian(x, amplitude=amp, center=cen, sigma=sig)
    gauss = gaussian(x, amplitude=amp, center=cen, sigma=sig)
    return log(loren) * gradient(gauss) / gradient(x)
"""
```

and then use this with *`ExpressionModel`* as:

```
mod = ExpressionModel('mycurve(x, height, mid, wid)', init_script=script,
                      independent_vars=['x'])
```

As above, this will interpret the parameter names to be `height`, `mid`, and `wid`, and build a model that can be used to fit data.

## 8.8 Example 1: Fit Peak data to Gaussian, Lorentzian, and Voigt profiles

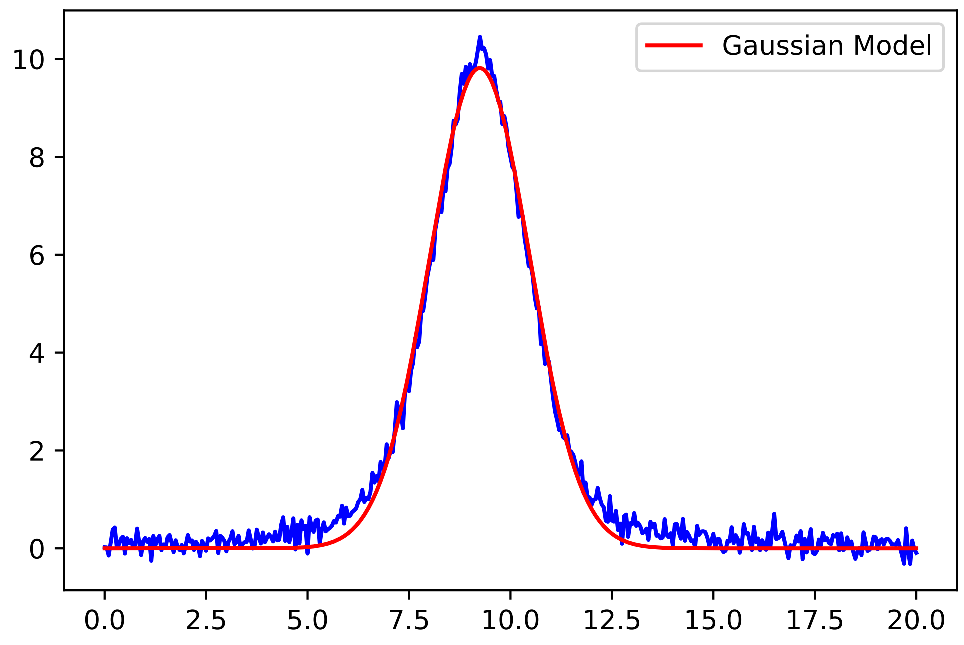
Here, we will fit data to three similar line shapes, in order to decide which might be the better model. We will start with a Gaussian profile, as in the previous chapter, but use the built-in *GaussianModel* instead of writing one ourselves. This is a slightly different version from the one in previous example in that the parameter names are different, and have built-in default values. We will 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]]
    Model(gaussian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 25
    # data points      = 401
    # variables        = 3
    chi-square         = 29.9943157
    reduced chi-square = 0.07536260
    Akaike info crit   = -1033.77437
    Bayesian info crit = -1021.79248
[[Variables]]
    amplitude:  30.3135620 +/- 0.15712686 (0.52%) (init = 43.62238)
    center:     9.24277047 +/- 0.00737496 (0.08%) (init = 9.25)
    sigma:      1.23218359 +/- 0.00737496 (0.60%) (init = 1.35)
    fwhm:       2.90157056 +/- 0.01736670 (0.60%) == '2.3548200*sigma'
    height:     9.81457817 +/- 0.05087283 (0.52%) == '0.3989423*amplitude/max(1e-15,␣
↪sigma)'
[[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 are `fwhm` and `height` parameters, to give the full-width-at-half-maximum and maximum peak height, respectively. And third, the automated initial guesses are pretty good. A plot of the fit:



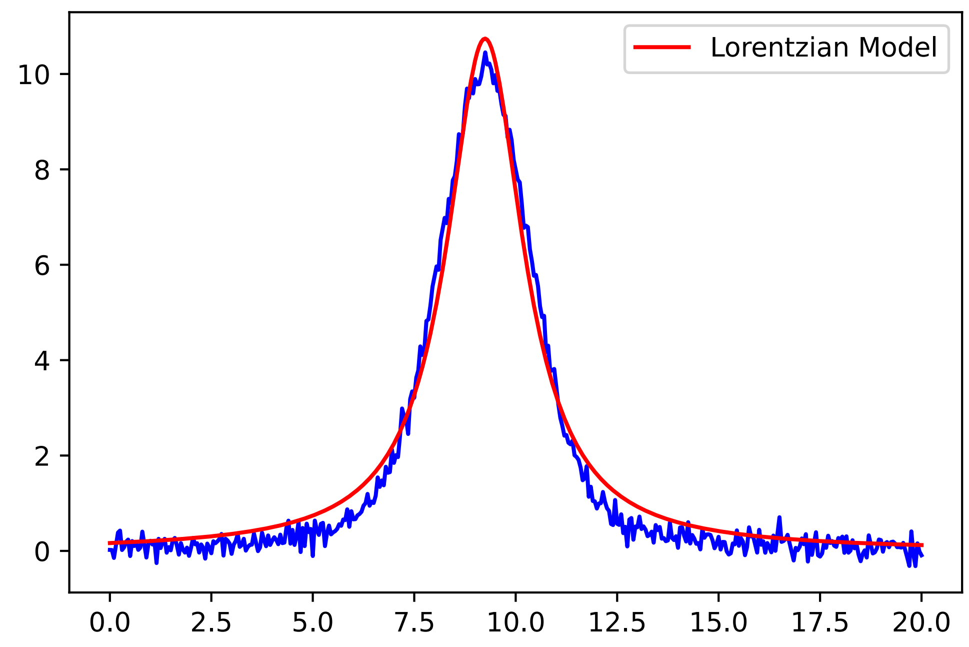
shows a decent match to the data – the fit worked with no explicit setting of initial parameter values. Looking more closely, the fit is not perfect, especially in the tails of the peak, suggesting that a different peak shape, with longer tails, should be used. Perhaps a Lorentzian would be better? To do this, we simply replace `GaussianModel` with `LorentzianModel` to get a *`LorentzianModel`*:

```
from lmfit.models import LorentzianModel
mod = LorentzianModel()
```

with the rest of the script as above. Perhaps predictably, the first thing we try gives results that are worse by comparing the fit statistics:

```
[[Model]]
    Model(lorentzian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 21
    # data points      = 401
    # variables        = 3
    chi-square         = 53.7535387
    reduced chi-square = 0.13505914
    Akaike info crit   = -799.830322
    Bayesian info crit = -787.848438
[[Variables]]
    amplitude:  38.9727645 +/- 0.31386183 (0.81%) (init = 54.52798)
    center:     9.24438944 +/- 0.00927619 (0.10%) (init = 9.25)
    sigma:      1.15483925 +/- 0.01315659 (1.14%) (init = 1.35)
    fwhm:       2.30967850 +/- 0.02631318 (1.14%) == '2.0000000*sigma'
    height:     10.7421156 +/- 0.08633945 (0.80%) == '0.3183099*amplitude/max(1e-15,␣
↪sigma)'
[[Correlations]] (unreported correlations are < 0.250)
    C(amplitude, sigma) =  0.709
```

and also by visual inspection of the fit to the data (figure below).



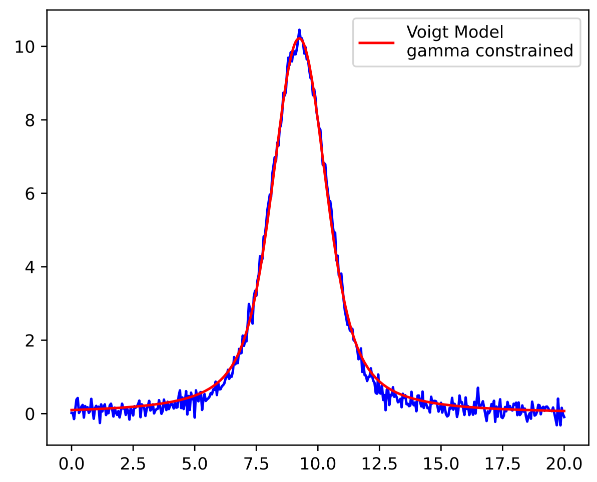
The tails are now too big, and the value for $\chi^2$ almost doubled. A Voigt model does a better job. Using *VoigtModel*, this is as simple as using:

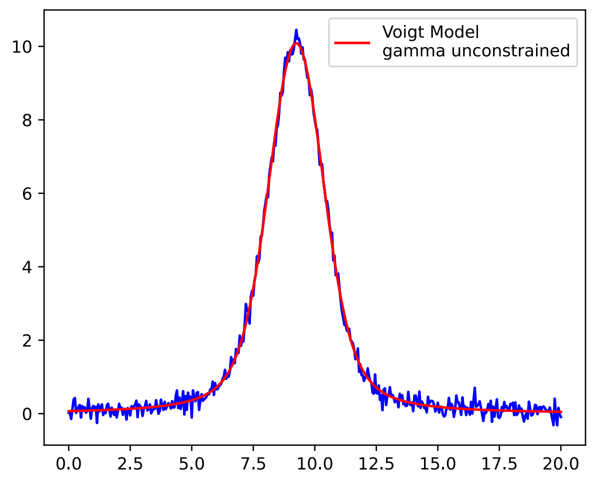
```
from lmfit.models import VoigtModel
mod = VoigtModel()
```

with all the rest of the script as above. This gives:

```
[[Model]]
    Model(voigt)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 21
    # data points      = 401
    # variables        = 3
    chi-square         = 14.5448627
    reduced chi-square = 0.03654488
    Akaike info crit   = -1324.00615
    Bayesian info crit = -1312.02427
[[Variables]]
    amplitude:  35.7554146 +/- 0.13861321 (0.39%) (init = 65.43358)
    center:     9.24411150 +/- 0.00505482 (0.05%) (init = 9.25)
    sigma:      0.73015627 +/- 0.00368460 (0.50%) (init = 0.8775)
    gamma:      0.73015627 +/- 0.00368460 (0.50%) == 'sigma'
    fwhm:       2.62950494 +/- 0.00806900 (0.31%) == '1.0692*gamma+sqrt(0.
↪8664*gamma**2+5.545083*sigma**2)'
    height:     10.2203969 +/- 0.03009415 (0.29%) == '(amplitude/(max(1e-15,␣
↪sigma*sqrt(2*pi))))*wofz((1j*gamma)/(max(1e-15, sigma*sqrt(2)))).real'
[[Correlations]] (unreported correlations are < 0.250)
    C(amplitude, sigma) =  0.651
```

which has a much better value for $\chi^2$ and the other goodness-of-fit measures, and an 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).

Can we do better? The Voigt function has a $\gamma$ 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 using something like:

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

which gives:

```
[[Model]]
    Model(voigt)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 21
    # data points      = 401
    # variables        = 4
    chi-square         = 10.9301767
    reduced chi-square = 0.02753193
    Akaike info crit   = -1436.57602
    Bayesian info crit = -1420.60017
[[Variables]]
    amplitude:  34.1914737 +/- 0.17946860 (0.52%) (init = 65.43358)
    center:     9.24374847 +/- 0.00441903 (0.05%) (init = 9.25)
    sigma:      0.89518909 +/- 0.01415450 (1.58%) (init = 0.8775)
    gamma:      0.52540199 +/- 0.01857955 (3.54%) (init = 0.7)
    fwhm:       2.72573635 +/- 0.01363980 (0.50%) == '1.0692*gamma+sqrt(0.
↪8664*gamma**2+5.545083*sigma**2)'
    height:     10.0872204 +/- 0.03482129 (0.35%) == '(amplitude/(max(1e-15,␣
↪sigma*sqrt(2*pi))))*wofz((1j*gamma)/(max(1e-15, sigma*sqrt(2)))).real'
[[Correlations]] (unreported correlations are < 0.250)
    C(sigma, gamma)     = -0.928
    C(amplitude, gamma) =  0.821
    C(amplitude, sigma) = -0.651
```

and the fit shown on the right above.

Comparing the two fits with the Voigt function, we see that $\chi^2$ 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. More compelling, reduced $\chi^2$ 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. In addition, we can use both the Akaike or Bayesian Information Criteria (see *Akaike and Bayesian Information Criteria*) to assess how likely the model with variable gamma is to explain the data than the model with gamma fixed to the value of sigma. According to theory, $\exp(-(\mathrm{AIC1} - \mathrm{AIC0})/2)$ gives the probability that a model with AIC1 is more likely than a model with AIC0. For the two models here, with AIC values of -1436 and -1324 (Note: if we had more carefully set the value for weights based on the noise in the data, these values might be positive, but there difference would be roughly the same), this says that the model with gamma fixed to sigma has a probability less than 5.e-25 of being the better model.

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

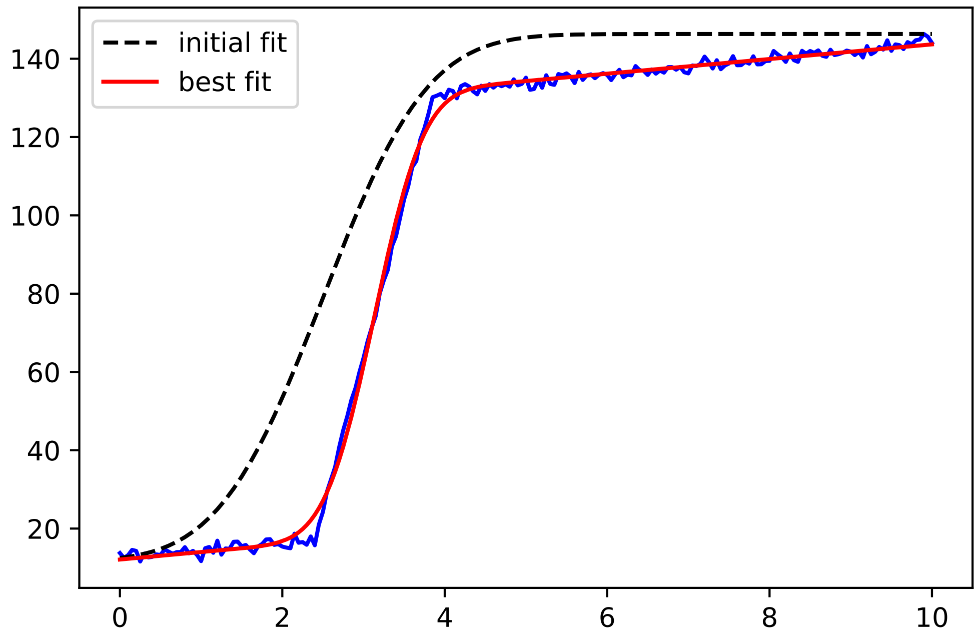
Here, we repeat the point made at the end of the last chapter that instances of *Model* class can be added together to make a *composite model*. By using the large number of built-in models available, it is therefore very simple to build models that contain multiple peaks and various backgrounds. An example of a simple fit to a noisy step function plus a constant:

```python
# <examples/doc_builtinmodels_stepmodel.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.models import LinearModel, StepModel

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)
np.random.seed(0)
y = 110.2 * (y + 9e-3*np.random.randn(x.size)) + 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, 'b')
plt.plot(x, out.init_fit, 'k--', label='initial fit')
plt.plot(x, out.best_fit, 'r-', label='best fit')
plt.legend(loc='best')
plt.show()
# <end examples/doc_builtinmodels_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 initial step function parameters, and make_params() arguments for the linear component. After making a composite model, we run fit() and report the results, which gives:

```
[[Model]]
    (Model(step, prefix='step_', form='erf') + Model(linear, prefix='line_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 49
    # data points      = 201
    # variables        = 5
    chi-square         = 593.709622
    reduced chi-square = 3.02913072
    Akaike info crit   = 227.700173
    Bayesian info crit = 244.216698
[[Variables]]
    line_slope:      1.87164656 +/- 0.09318713 (4.98%) (init = 0)
    line_intercept:  12.0964833 +/- 0.27606235 (2.28%) (init = 11.58574)
    step_amplitude:  112.858376 +/- 0.65392947 (0.58%) (init = 134.7378)
    step_center:     3.13494792 +/- 0.00516615 (0.16%) (init = 2.5)
    step_sigma:      0.67392841 +/- 0.01091168 (1.62%) (init = 1.428571)
[[Correlations]] (unreported correlations are < 0.100)
    C(line_slope, step_amplitude)     = -0.879
    C(step_amplitude, step_sigma)     =  0.564
    C(line_slope, step_sigma)         = -0.457
    C(line_intercept, step_center)    =  0.427
    C(line_slope, line_intercept)     = -0.309
    C(line_slope, step_center)        = -0.234
    C(line_intercept, step_sigma)     = -0.137
    C(line_intercept, step_amplitude) = -0.117
    C(step_amplitude, step_center)    =  0.109
```

with a plot of



## 8.10 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
# <examples/doc_builtinmodels_nistgauss.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.models import ExponentialModel, GaussianModel

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

gauss2 = GaussianModel(prefix='g2_')
pars.update(gauss2.make_params())

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

mod = gauss1 + gauss2 + exp_mod

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

print(out.fit_report(min_correl=0.5))

fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))
axes[0].plot(x, y, 'b')
axes[0].plot(x, init, 'k--', label='initial fit')
axes[0].plot(x, out.best_fit, 'r-', label='best fit')
axes[0].legend(loc='best')

comps = out.eval_components(x=x)
axes[1].plot(x, y, 'b')
axes[1].plot(x, comps['g1_'], 'g--', label='Gaussian component 1')
axes[1].plot(x, comps['g2_'], 'm--', label='Gaussian component 2')
axes[1].plot(x, comps['exp_'], 'k--', label='Exponential component')
axes[1].legend(loc='best')

plt.show()
# <end examples/doc_builtinmodels_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.

Note that the calls to `make_param()` used the bare name, without the prefix. We could have used the prefixes, 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:

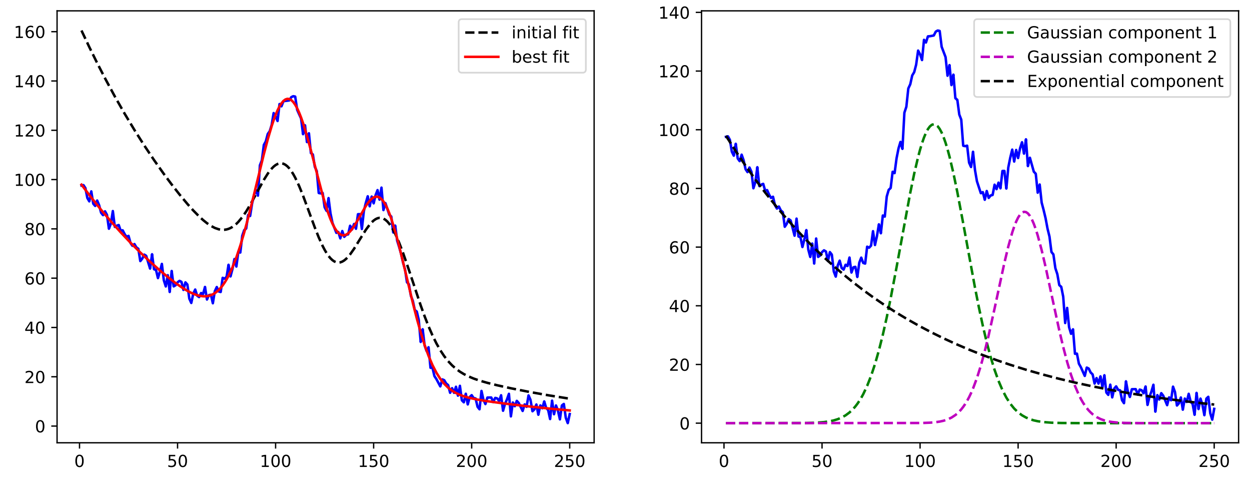
```
[[Model]]
    ((Model(gaussian, prefix='g1_') + Model(gaussian, prefix='g2_')) +␣
↪Model(exponential, prefix='exp_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 46
    # data points      = 250
    # variables        = 8
    chi-square         = 1247.52821
    reduced chi-square = 5.15507524
    Akaike info crit   = 417.864631
    Bayesian info crit = 446.036318
[[Variables]]
    exp_amplitude:  99.0183283 +/- 0.53748735 (0.54%) (init = 162.2102)
    exp_decay:      90.9508860 +/- 1.10310509 (1.21%) (init = 93.24905)
    g1_amplitude:   4257.77319 +/- 42.3833645 (1.00%) (init = 2000)
    g1_center:      107.030954 +/- 0.15006786 (0.14%) (init = 105)
    g1_sigma:       16.6725753 +/- 0.16048161 (0.96%) (init = 15)
    g1_fwhm:        39.2609138 +/- 0.37790530 (0.96%) == '2.3548200*g1_sigma'
    g1_height:      101.880231 +/- 0.59217099 (0.58%) == '0.3989423*g1_amplitude/
↪max(1e-15, g1_sigma)'
    g2_amplitude:   2493.41771 +/- 36.1694731 (1.45%) (init = 2000)
    g2_center:      153.270101 +/- 0.19466743 (0.13%) (init = 155)
    g2_sigma:       13.8069484 +/- 0.18679415 (1.35%) (init = 15)
    g2_fwhm:        32.5128783 +/- 0.43986659 (1.35%) == '2.3548200*g2_sigma'
    g2_height:      72.0455934 +/- 0.61722094 (0.86%) == '0.3989423*g2_amplitude/
↪max(1e-15, g2_sigma)'
[[Correlations]] (unreported correlations are < 0.100)
    C(g1_amplitude, g1_sigma)      =  0.824
    C(g2_amplitude, g2_sigma)      =  0.815
    C(exp_amplitude, exp_decay)    = -0.695
    C(g1_sigma, g2_center)         =  0.684
    C(g1_center, g2_amplitude)     = -0.669
    C(g1_center, g2_sigma)         = -0.652
    C(g1_amplitude, g2_center)     =  0.648
    C(g1_center, g2_center)        =  0.621
    C(g1_center, g1_sigma)         =  0.507
    C(exp_decay, g1_amplitude)     = -0.507
    C(g1_sigma, g2_amplitude)      = -0.491
    C(g2_center, g2_sigma)         = -0.489
    C(g1_sigma, g2_sigma)          = -0.483
    C(g2_amplitude, g2_center)     = -0.476
    C(exp_decay, g2_amplitude)     = -0.427
    C(g1_amplitude, g1_center)     =  0.418
    C(g1_amplitude, g2_sigma)      = -0.401
    C(g1_amplitude, g2_amplitude)  = -0.307
    C(exp_amplitude, g2_amplitude) =  0.282
    C(exp_decay, g1_sigma)         = -0.252
    C(exp_decay, g2_sigma)         = -0.233
```

(continues on next page)

(continued from previous page)

```
    C(exp_amplitude, g2_sigma)      =  0.171
    C(exp_decay, g2_center)         = -0.151
    C(exp_amplitude, g1_amplitude)  =  0.148
    C(exp_decay, g1_center)         =  0.105
```

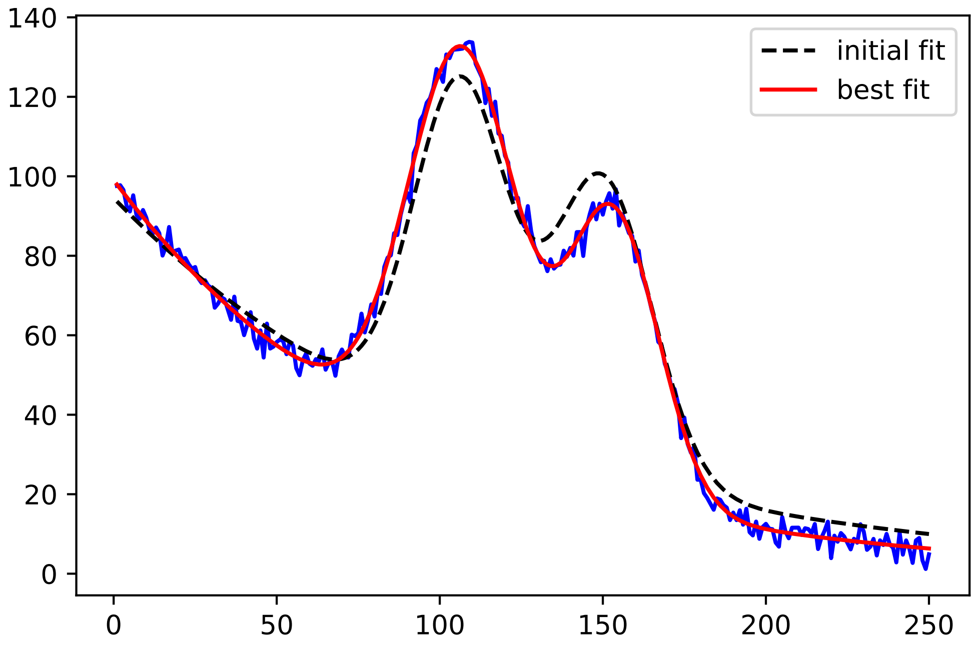
We get a very good fit to this problem (described at the NIST site as of average difficulty, but the tests there are generally deliberately challenging) by applying reasonable initial guesses and putting modest but explicit bounds on the parameter values. The overall fit is shown on the left, with its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



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 (see below).



The fit converges to the same answer, giving to identical values (to the precision printed out in the report), but in fewer steps, and without any bounds on parameters at all:

```
[[Model]]
    ((Model(gaussian, prefix='g1_') + Model(gaussian, prefix='g2_')) +␣
↪Model(exponential, prefix='exp_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 37
    # data points      = 250
    # variables        = 8
    chi-square         = 1247.52821
    reduced chi-square = 5.15507524
    Akaike info crit   = 417.864631
    Bayesian info crit = 446.036318
[[Variables]]
    exp_amplitude:  99.0183270 +/- 0.53748905 (0.54%) (init = 94.53724)
    exp_decay:      90.9508890 +/- 1.10310483 (1.21%) (init = 111.1985)
    g1_amplitude:   4257.77343 +/- 42.3836433 (1.00%) (init = 3189.648)
    g1_center:      107.030956 +/- 0.15006873 (0.14%) (init = 106.5)
    g1_sigma:       16.6725765 +/- 0.16048227 (0.96%) (init = 14.5)
    g1_fwhm:        39.2609166 +/- 0.37790686 (0.96%) == '2.3548200*g1_sigma'
    g1_height:      101.880230 +/- 0.59217233 (0.58%) == '0.3989423*g1_amplitude/
↪max(1e-15, g1_sigma)'
    g2_amplitude:   2493.41733 +/- 36.1696906 (1.45%) (init = 2818.337)
    g2_center:      153.270101 +/- 0.19466905 (0.13%) (init = 150)
    g2_sigma:       13.8069461 +/- 0.18679534 (1.35%) (init = 15)
    g2_fwhm:        32.5128728 +/- 0.43986939 (1.35%) == '2.3548200*g2_sigma'
    g2_height:      72.0455948 +/- 0.61722328 (0.86%) == '0.3989423*g2_amplitude/
↪max(1e-15, g2_sigma)'
[[Correlations]] (unreported correlations are < 0.100)
    C(g1_amplitude, g1_sigma)      =  0.824
    C(g2_amplitude, g2_sigma)      =  0.815
    C(exp_amplitude, exp_decay)    = -0.695
```

(continues on next page)

(continued from previous page)

```
    C(g1_sigma, g2_center)         =  0.684
    C(g1_center, g2_amplitude)     = -0.669
    C(g1_center, g2_sigma)         = -0.652
    C(g1_amplitude, g2_center)     =  0.648
    C(g1_center, g2_center)        =  0.621
    C(g1_center, g1_sigma)         =  0.507
    C(exp_decay, g1_amplitude)     = -0.507
    C(g1_sigma, g2_amplitude)      = -0.491
    C(g2_center, g2_sigma)         = -0.489
    C(g1_sigma, g2_sigma)          = -0.483
    C(g2_amplitude, g2_center)     = -0.476
    C(exp_decay, g2_amplitude)     = -0.427
    C(g1_amplitude, g1_center)     =  0.418
    C(g1_amplitude, g2_sigma)      = -0.401
    C(g1_amplitude, g2_amplitude)  = -0.307
    C(exp_amplitude, g2_amplitude) =  0.282
    C(exp_decay, g1_sigma)         = -0.252
    C(exp_decay, g2_sigma)         = -0.233
    C(exp_amplitude, g2_sigma)     =  0.171
    C(exp_decay, g2_center)        = -0.151
    C(exp_amplitude, g1_amplitude) =  0.148
    C(exp_decay, g1_center)        =  0.105
```

This script is in the file `doc_builtinmodels_nistgauss2.py` in the examples folder, and the figure above shows an improved initial estimate of the data.

CHAPTER

NINE

# 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 since the estimation of the standard error from the estimated covariance matrix is normally quite good.

But for some models, the sum of two exponentials for example, 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.

## 9.1 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 $\chi_0^2$ and $\chi_f^2$ 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 and `P` the number of parameters 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.

## 9.2 A basic example

First we create an example problem:

```python
import numpy as np

import lmfit

x = np.linspace(0.3, 10, 100)
np.random.seed(0)
y = 1/(0.1*x) + 2 + 0.1*np.random.randn(x.size)
pars = lmfit.Parameters()
pars.add_many(('a', 0.1), ('b', 1))


def residual(p):
    return 1/(p['a']*x) + p['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:

```
mini = lmfit.Minimizer(residual, pars)
result = mini.minimize()

print(lmfit.fit_report(result.params))
```

```
[[Variables]]
    a:  0.09943896 +/- 1.9322e-04 (0.19%) (init = 0.1)
    b:  1.98476942 +/- 0.01222678 (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(mini, result)
lmfit.printfuncs.report_ci(ci)
```

```
      99.73%    95.45%    68.27%    _BEST_     68.27%    95.45%    99.73%
 a:  -0.00059  -0.00039  -0.00019   0.09944  +0.00019  +0.00039  +0.00060
 b:  -0.03766  -0.02478  -0.01230   1.98477  +0.01230  +0.02478  +0.03761
```

This shows the best-fit values for the parameters in the `_BEST_` column, and parameter values that are at the varying confidence levels given by steps in $\sigma$. As we can see, the estimated error is almost the same, and the uncertainties are well behaved: Going from 1-$\sigma$ (68% confidence) to 3-$\sigma$ (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.

## 9.3 Working without standard error estimates

Sometimes the estimation of the standard errors from the covariance matrix fails, especially if values are near given bounds. Hence, to find the confidence intervals in these cases, it is necessary to set the errors by hand. Note that the standard error is only used to find an upper limit for each value, hence the exact value is not important.

To set the step-size to 10% of the initial value we loop through all parameters and set it manually:

```
for p in result.params:
    result.params[p].stderr = abs(result.params[p].value * 0.1)
```

## 9.4 An advanced example for evaluating confidence intervals

Now we look at a problem where calculating the error from approximated covariance can lead to misleading result – the same double exponential problem shown in *Minimizer.emcee() - calculating the posterior probability distribution of parameters*. In fact such a problem is particularly hard for the Levenberg-Marquardt method, so we first estimate the results using the slower but robust Nelder-Mead method. We can then compare the uncertainties computed (if the `numdifftools` package is installed) with those estimated using Levenberg-Marquardt around the previously found solution. We can also compare to the results of using `emcee`.

```
# <examples/doc_confidence_advanced.py>
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```python
import lmfit

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(x.size)

p = lmfit.Parameters()
p.add_many(('a1', 4.), ('a2', 4.), ('t1', 3.), ('t2', 3.))


def residual(p):
    return p['a1']*np.exp(-x/p['t1']) + p['a2']*np.exp(-(x-0.1)/p['t2']) - y


# create Minimizer
mini = lmfit.Minimizer(residual, p, nan_policy='propagate')

# first solve with Nelder-Mead algorithm
out1 = mini.minimize(method='Nelder')

# then solve with Levenberg-Marquardt using the
# Nelder-Mead solution as a starting point
out2 = mini.minimize(method='leastsq', params=out1.params)

lmfit.report_fit(out2.params, min_correl=0.5)

ci, trace = lmfit.conf_interval(mini, out2, sigmas=[1, 2], trace=True)
lmfit.printfuncs.report_ci(ci)

# plot data and best fit
plt.figure()
plt.plot(x, y, 'b')
plt.plot(x, residual(out2.params) + y, 'r-')

# plot confidence intervals (a1 vs t2 and a2 vs t2)
fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))
cx, cy, grid = lmfit.conf_interval2d(mini, out2, 'a1', 't2', 30, 30)
ctp = axes[0].contourf(cx, cy, grid, np.linspace(0, 1, 11))
fig.colorbar(ctp, ax=axes[0])
axes[0].set_xlabel('a1')
axes[0].set_ylabel('t2')

cx, cy, grid = lmfit.conf_interval2d(mini, out2, 'a2', 't2', 30, 30)
ctp = axes[1].contourf(cx, cy, grid, np.linspace(0, 1, 11))
fig.colorbar(ctp, ax=axes[1])
axes[1].set_xlabel('a2')
axes[1].set_ylabel('t2')

# plot dependence between two parameters
fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))
cx1, cy1, prob = trace['a1']['a1'], trace['a1']['t2'], trace['a1']['prob']
cx2, cy2, prob2 = trace['t2']['t2'], trace['t2']['a1'], trace['t2']['prob']

axes[0].scatter(cx1, cy1, c=prob, s=30)
axes[0].set_xlabel('a1')
axes[0].set_ylabel('t2')
```

(continues on next page)

(continued from previous page)

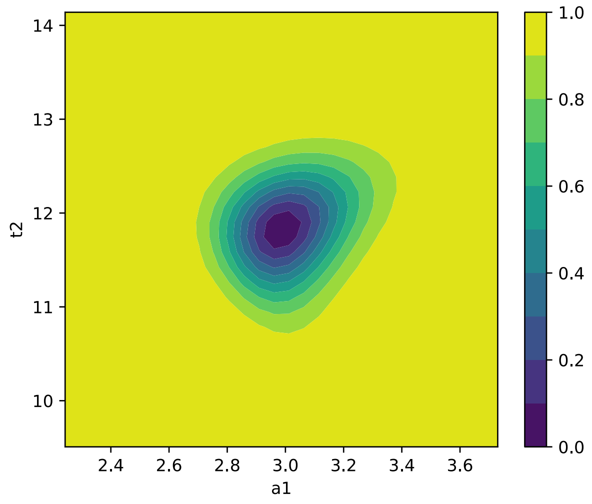
```
axes[1].scatter(cx2, cy2, c=prob2, s=30)
axes[1].set_xlabel('t2')
axes[1].set_ylabel('a1')

plt.show()
# <end examples/doc_confidence_advanced.py>
```

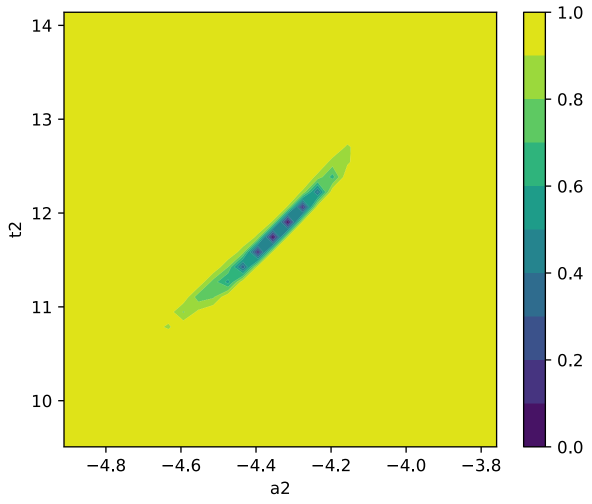
which will report:

```
[[Variables]]
    a1:  2.98622120 +/- 0.14867187 (4.98%) (init = 2.986237)
    a2: -4.33526327 +/- 0.11527506 (2.66%) (init = -4.335256)
    t1:  1.30994233 +/- 0.13121177 (10.02%) (init = 1.309932)
    t2:  11.8240351 +/- 0.46316470 (3.92%) (init = 11.82408)
[[Correlations]] (unreported correlations are < 0.500)
    C(a2, t2) =  0.987
    C(a2, t1) = -0.925
    C(t1, t2) = -0.881
    C(a1, t1) = -0.599

       95.45%    68.27%    _BEST_    68.27%    95.45%
 a1:  -0.27286  -0.14165   2.98622  +0.16353  +0.36343
 a2:  -0.30444  -0.13219  -4.33526  +0.10688  +0.19683
 t1:  -0.23392  -0.12494   1.30994  +0.14660  +0.32369
 t2:  -1.01943  -0.48820  11.82404  +0.46041  +0.90441
```

Again we called `conf_interval()`, this time with tracing and only for 1- and 2-$\sigma$. 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 $\sigma$ (68% confidence) to 2 $\sigma$ (95% confidence) is not very predictable.

Plots of the confidence region are shown in the figures below for `a1` and `t2` (left), and `a2` and `t2` (right):



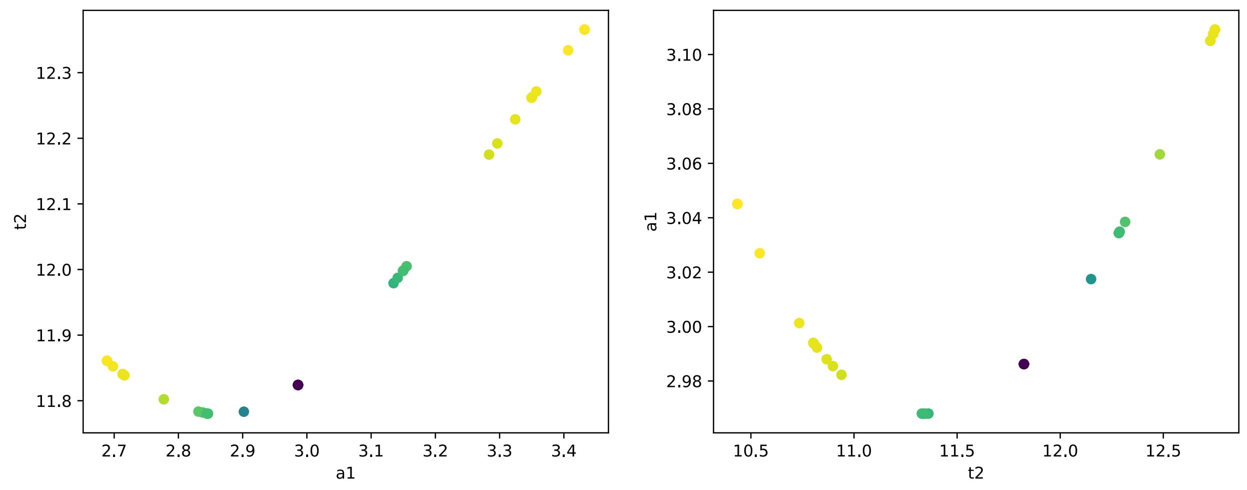


Neither of these plots is very much like an ellipse, which is implicitly assumed by the approach using the covariance matrix. The plots actually look quite a bit like those found with MCMC and shown in the "corner plot" in *Minimizer.emcee() - calculating the posterior probability distribution of parameters*. In fact, comparing the confidence interval results here with the results for the 1- and 2-$\sigma$ error estimated with `emcee`, we can see that the agreement is pretty good and that the asymmetry in the parameter distributions are reflected well in the asymmetry of the uncertainties.

The trace returned as the optional second argument from `conf_interval()` contains a dictionary for each variable parameter. The values are dictionaries with arrays of values for each variable, and an array of corresponding probabilities for the corresponding cumulative variables. This can be used to show the dependence between two parameters:

```
fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))
cx1, cy1, prob = trace['a1']['a1'], trace['a1']['t2'], trace['a1']['prob']
cx2, cy2, prob2 = trace['t2']['t2'], trace['t2']['a1'], trace['t2']['prob']

axes[0].scatter(cx1, cy1, c=prob, s=30)
axes[0].set_xlabel('a1')
axes[0].set_ylabel('t2')

axes[1].scatter(cx2, cy2, c=prob2, s=30)
axes[1].set_xlabel('t2')
axes[1].set_ylabel('a1')

plt.show()
```

which shows the trace of values:



As an alternative/complement to the confidence intervals, the `Minimizer.emcee()` method uses Markov Chain Monte Carlo to sample the posterior probability distribution. These distributions demonstrate the range of solutions that the data supports and we refer to *Minimizer.emcee() - calculating the posterior probability distribution of parameters* where this methodology was used on the same problem.

Credible intervals (the Bayesian equivalent of the frequentist confidence interval) can be obtained with this method. MCMC can be used for model selection, to determine outliers, to marginalize over nuisance parameters, etcetera. For example, you may have fractionally underestimated the uncertainties on a dataset. MCMC can be used to estimate the true level of uncertainty on each data point. A tutorial on the possibilities offered by MCMC can be found at[1].

[1] https://jakevdp.github.io/blog/2014/03/11/frequentism-and-bayesianism-a-practical-intro/

## 9.5 Confidence Interval Functions

**`conf_interval`**(*minimizer*, *result*, *p_names=None*, *sigmas=None*, *trace=False*, *maxiter=200*, *verbose=False*, *prob_func=None*)

Calculate the confidence interval (CI) for parameters.

The parameter for which the CI is calculated will be varied, while the remaining parameters are re-optimized to minimize the chi-square. The resulting chi-square is used to calculate the probability with a given statistic (e.g., F-test). 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, holding objective function.
- **`result`** (`MinimizerResult`) – The result of running minimize().
- **`p_names`** (*`list, optional`*) – Names of the parameters for which the CI is calculated. If None (default), the CI is calculated for every parameter.
- **`sigmas`** (*`list, optional`*) – The sigma-levels to find (default is [1, 2, 3]). See Notes below.
- **`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”.
- **`maxiter`** (*`int, optional`*) – Maximum of iteration to find an upper limit (default is 200).
- **`verbose`** (*`bool, optional`*) – Print extra debugging information (default is False).
- **`prob_func`** (*`None or callable, optional`*) – Function to calculate the probability from the optimized chi-square. Default is None and uses the built-in function *f_compare* (i.e., F-test).

**Returns**

- **output** (*dict*) – A dictionary containing a list of `(sigma, vals)`-tuples for each parameter.
- **trace_dict** (*dict, optional*) – Only if trace is True. Is a dictionary, 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.

**See also:**

*`conf_interval2d`*

### Notes

The values for *sigma* are taken as the number of standard deviations for a normal distribution and converted to probabilities. That is, the default `sigma=[1, 2, 3]` will use probabilities of 0.6827, 0.9545, and 0.9973. If any of the sigma values is less than 1, that will be interpreted as a probability. That is, a value of 1 and 0.6827 will give the same results, within precision.

### 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=[0.5, 1, 2, 3], 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 parameters.

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

Calculate confidence regions for two fixed parameters.

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

**Parameters**

- **minimizer** (`Minimizer`) – The minimizer to use, holding objective function.
- **result** (`MinimizerResult`) – The result of running minimize().
- **x_name** (*str*) – The name of the parameter which will be the x direction.
- **y_name** (*str*) – The name of the parameter which will be the y direction.
- **nx** (*int, optional*) – Number of points in the x direction.
- **ny** (*int, optional*) – Number of points in the y direction.
- **limits** (*tuple, optional*) – Should have the form `((x_upper, x_lower), (y_upper, y_lower))`. If not given, the default is 5.0*std-errors in each direction.
- **prob_func** (*None or callable, optional*) – Function to calculate the probability from the optimized chi-square. Default is None and uses the built-in function *f_compare* (i.e., F-test).

**Returns**

- **x** (*numpy.ndarray*) – X-coordinates (same shape as *nx*).
- **y** (*numpy.ndarray*) – Y-coordinates (same shape as *ny*).
- **grid** (*numpy.ndarray*) – Grid containing the calculated probabilities (with shape `(nx, ny)`).

**See also:**

*conf_interval*

### Examples

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

**`ci_report`**(*ci*, *with_offset=True*, *ndigits=5*)

Return text of a report for confidence intervals.

**Parameters**

- **ci** (*dict*) – The result of `conf_interval()`: a dictionary containing a list of `(sigma, vals)`-tuples for each parameter.
- **with_offset** (*bool, optional*) – Whether to subtract best value from all other values (default is True).
- **ndigits** (*int, optional*) – Number of significant digits to show (default is 5).

**Returns** Text of formatted report on confidence intervals.

**Return type** str

CHAPTER

TEN

# 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

ELEVEN

# 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.

## 11.1 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:

```
from lmfit import Parameters

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.

## 11.2 Supported Operators, Functions, and Constants

The mathematical expressions used to define constrained Parameters need to be valid Python expressions. As you would expect, the operators `+`, `-`, `*`, `/`, and `**`, 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('wid_2', expr='sqrt(pi)*wid_1')
pars.add('cen_2', expr='cen_1 * wid_2 / max(wid_1, 0.001)')
```

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

```
pars.add('param_a', value=1)
pars.add('param_b', value=2)
pars.add('test_val', value=100)

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

which is equivalent to the more familiar:

```
if pars['test_val'].value/2. > 100:
    bounded = pars['param_a'].value
else:
    bounded = pars['param_b'].value
```

## 11.3 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.

## 11.4 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 `Parameters` 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, for example this Lorentzian function:

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

You can add this user-defined function to the asteval interpreter of the `Parameters` class:

```
from lmfit import Parameters

pars = Parameters()
pars._asteval.symtable['lorentzian'] = mylorentzian
```

and then initialize the `Minimizer` class with this parameter set:

```
from lmfit import Minimizer

def userfcn(x, params):
    pass

fitter = Minimizer(userfcn, pars)
```

Alternatively, one can first initialize the `Minimizer` class and add the function to the asteval interpreter of `Minimizer.params` afterwards:

```
pars = Parameters()
fitter = Minimizer(userfcn, pars)
fitter.params._asteval.symtable['lorentzian'] = mylorentzian
```

In both cases the user-defined `lorentzian()` function can now be used in constraint expressions.

CHAPTER

TWELVE

# RELEASE NOTES

This section discusses changes between versions, especially changes significant to the use and behavior of the library. This is not meant to be a comprehensive list of changes. For such a complete record, consult the lmfit GitHub repository.

## 12.1 Version 1.0.2 Release Notes

Version 1.0.2 officially supports Python 3.9 and has dropped support for Python 3.5. The minimum version of the following dependencies were updated: asteval>=0.9.21, numpy>=1.18, and scipy>=1.3.

New features:

- added two-dimensional Gaussian lineshape and model (PR #642; @mpmdean)
- all built-in models are now registered in `lmfit.models.lmfit_models`; new Model class attribute `valid_forms` (PR #663; @rayosborn)
- added a SineModel (PR #676; @lneuhaus)
- add the `run_mcmc_kwargs argument` to `Minimizer.emcee` to pass to the `emcee.EnsembleSampler.run_mcmc` function (PR #694; @rbnvrw)

Bug fixes:

- `ModelResult.eval_uncertainty` should use provided Parameters (PR #646)
- center in lognormal model can be negative (Issue #644, PR #645; @YoshieraHuang)
- restore best-fit values after calculation of covariance matrix (Issue #655, PR #657)
- add helper-function `not_zero` to prevent ZeroDivisionError in lineshapes and use in exponential lineshape (Issue #631, PR #664; @s-weigand)
- save `last_internal_values` and use to restore internal values if fit is aborted (PR #667)
- dumping a fit using the `lbfgsb` method now works, convert bytes to string if needed (Issue #677, PR #678; @leonfoks)
- fix use of callable Jacobian for scalar methods (PR #681; @mstimberg)
- preserve float/int types when encoding for JSON (PR #696; @jedzill4)
- better support for saving/loading of ExpressionModels and assure that `init_params` and `init_fit` are set when loading a `ModelResult` (PR #706)

Various:

- update minimum dependencies (PRs #688, #693)

- improvements in coding style, docstrings, CI, and test coverage (PRs #647, #649, #650, #653, #654; #685, #668, #689)
- fix typo in Oscillator (PR #658; @flothesof)
- add example using SymPy (PR #662)
- allow better custom pool for emcee() (Issue #666, PR #667)
- update NIST Strd reference functions and tests (PR #670)
- make building of documentation cross-platform (PR #673; @s-weigand)
- relax module name check in `test_check_ast_errors` for Python 3.9 (Issue #674, PR #675; @mwhudson)
- fix/update layout of documentation, now uses the sphinx13 theme (PR #687)
- fixed DeprecationWarnings reported by NumPy v1.2.0 (PR #699)
- increase value of `tiny` and check for it in bounded parameters to avoid "parameter not moving from initial value" (Issue #700, PR #701)
- add `max_nfev` to `basinhopping` and `brute` (now supported everywhere in lmfit) and set to more uniform default values (PR #701)
- use Azure Pipelines for CI, drop Travis (PRs #696 and #702)

## 12.2 Version 1.0.1 Release Notes

**Version 1.0.1 is the last release that supports Python 3.5**. All newer version will require 3.6+ so that we can use formatting-strings and rely on dictionaries being ordered.

New features:

- added thermal distribution model and lineshape (PR #620; @mpmdean)
- introduced a new argument `max_nfev` to uniformly specify the maximum number of function evalutions (PR #610) **Please note: all other arguments (e.g., ``maxfev``, ``maxiter``, ...) will no longer be passed to the underlying solver. A warning will be emitted stating that one should use ``max_nfev``.**
- the attribute `call_kws` was added to the `MinimizerResult` class and contains the keyword arguments that are supplied to the solver in SciPy.

Bug fixes:

- fixes to the `load` and `__setstate__` methods of the Parameter class
- fixed failure of ModelResult.dump() due to missing attributes (Issue #611, PR #623; @mpmdean)
- `guess_from_peak` function now also works correctly with decreasing x-values or when using pandas (PRs #627 and #629; @mpmdean)
- the `Parameter.set()` method now correctly first updates the boundaries and then the value (Issue #636, PR #637; @arunpersaud)

Various:

- fixed typo for the use of expressions in the documentation (Issue #610; @jkrogager)
- removal of PY2-compatibility and unused code and improved test coverage (PRs #619, #631, and #633)
- removed deprecated `isParameter` function and automatic conversion of an `uncertainties` object (PR #626)

- inaccurate FWHM calculations were removed from built-in models, others labeled as estimates (Issue #616 and PR #630)
- corrected spelling mistake for the Doniach lineshape and model (Issue #634; @rayosborn)
- removed unsupported/untested code for IPython notebooks in lmfit/ui/*

## 12.3 Version 1.0.0 Release Notes

**Version 1.0.0 supports Python 3.5, 3.6, 3.7, and 3.8**

New features:

- no new features are introduced in 1.0.0.

Improvements:

- support for Python 2 and use of the `six` package are removed. (PR #612)

Various:

- documentation updates to clarify the use of `emcee`. (PR #614)

## 12.4 Version 0.9.15 Release Notes

**Version 0.9.15 is the last release that supports Python 2.7**; it now also fully supports Python 3.8.

New features, improvements, and bug fixes:

- move application of parameter bounds to setter instead of getter (PR #587)
- add support for non-array Jacobian types in least_squares (Issue #588, @ezwelty in PR #589)
- add more information (i.e., acor and acceptance_fraction) about emcee fit (@j-zimmermann in PR #593)
- “name” is now a required positional argument for Parameter class, update the magic methods (PR #595)
- fix nvars count and bound handling in confidence interval calculations (Issue #597, PR #598)
- support Python 3.8; requires asteval >= 0.9.16 (PR #599)
- only support emcee version 3 (i.e., no PTSampler anymore) (PR #600)
- fix and refactor prob_bunc in confidence interval calculations (PR #604)
- fix adding Parameters with custom user-defined symbols (Issue #607, PR #608; thanks to @gbouvignies for the report)

Various:

- bump requirements to LTS version of SciPy/ NumPy and code clean-up (PR #591)
- documentation updates (PR #596, and others)
- improve test coverage and Travis CI updates (PR #595, and others)
- update pre-commit hooks and configuration in setup.cfg

To-be deprecated: - function Parameter.isParameter and conversion from uncertainties.core.Variable to value in _getval (PR #595)

## 12.5 Version 0.9.14 Release Notes

New features:

- the global optimizers `shgo` and `dual_annealing` (new in SciPy v1.2) are now supported (Issue #527; PRs #545 and #556)
- `eval` method added to the Parameter class (PR #550 by @zobristnicholas)
- avoid ZeroDivisionError in `printfuncs.params_html_table` (PR #552 by @aaristov and PR #559)
- add parallelization to `brute` method (PR #564, requires SciPy v1.3)

Bug fixes:

- consider only varying parameters when reporting potential issues with calculating errorbars (PR #549) and compare `value` to both `min` and `max` (PR #571)
- guard against division by zero in lineshape functions and `FWHM` and `height` expression calculations (PR #545)
- fix issues with restoring a saved Model (Issue #553; PR #554)
- always set `result.method` for `emcee` algorithm (PR #558)
- more careful adding of parameters to handle out-of-order constraint expressions (Issue #560; PR #561)
- make sure all parameters in Model.guess() use prefixes (PRs #567 and #569)
- use `inspect.signature` for PY3 to support wrapped functions (Issue #570; PR #576)
- fix `result.nfev`` for `brute` method when using parallelization (Issue #578; PR #579)

Various:

- remove “missing” in the Model class (replaced by nan_policy) and “drop” as option to nan_policy (replaced by omit) deprecated since 0.9 (PR #565).
- deprecate ‘report_errors’ in printfuncs.py (PR #571)
- updates to the documentation to use `jupyter-sphinx` to include examples/output (PRs #573 and #575)
- include a Gallery with examples in the documentation using `sphinx-gallery` (PR #574 and #583)
- improve test-coverage (PRs #571, #572 and #585)
- add/clarify warning messages when NaN values are detected (PR #586)
- several updates to docstrings (Issue #584; PR #583, and others)
- update pre-commit hooks and several docstrings

## 12.6 Version 0.9.13 Release Notes

New features:

- Clearer warning message in fit reports when uncertainties should but cannot be estimated, including guesses of which Parameters to examine (#521, #543)
- SplitLorenztianModel and split_lorentzian function (#523)
- HTML representations for Parameter, MinimizerResult, and Model so that they can be printed better with Jupyter (#524, #548)
- support parallelization for differential evolution (#526)

Bug fixes:

- delay import of matplotlib (and so, the selection of its backend) as late as possible (#528, #529)
- fix for saving, loading, and reloading ModelResults (#534)
- fix to leastsq to report the best-fit values, not the values tried last (#535, #536)
- fix synchronization of all parameter values on Model.guess() (#539, #542)
- improve deprecation warnings for outdated nan_policy keywords (#540)
- fix for edge case in gformat() (#547)

Project management:

- using pre-commit framework to improve and enforce coding style (#533)
- added code coverage report to github main page
- updated docs, github templates, added several tests.
- dropped support and testing for Python 3.4.

## 12.7 Version 0.9.12 Release Notes

Lmfit package is now licensed under BSD-3.

New features:

- SkewedVoigtModel was added as built-in model (Issue #493)
- Parameter uncertainties and correlations are reported for least_squares
- Plotting of complex-valued models is now handled in ModelResult class (PR #503)
- A model's independent variable is allowed to be an object (Issue #492)
- Added `usersyms` to Parameters() initialization to make it easier to add custom functions and symbols (Issue #507)
- the `numdifftools` package can be used to calculate parameter uncertainties and correlations for all solvers that do not natively support this (PR #506)
- `emcee` can now be used as method keyword-argument to Minimizer.minimize and minimize function, which allows for using `emcee` in the Model class (PR #512; see `examples/example_emcee_with_Model.py`)

(Bug)fixes:

- asteval errors are now flushed after raising (Issue #486)
- max_time and evaluation time for ExpressionModel increased to 1 hour (Issue #489)
- loading a saved ModelResult now restores all attributes (Issue #491)
- development versions of scipy and emcee are now supported (Issue #497 and PR #496)
- ModelResult.eval() do no longer overwrite the userkws dictionary (Issue #499)
- running the test suite requires `pytest` only (Issue #504)
- improved FWHM calculation for VoigtModel (PR #514)

## 12.8 Version 0.9.10 Release Notes

Two new global algorithms were added: basinhopping and AMPGO. Basinhopping wraps the method present in `scipy`, and more information can be found in the documentation (*`basinhopping()`* and scipy.optimize.basinhopping). The Adaptive Memory Programming for Global Optimization (AMPGO) algorithm was adapted from Python code written by Andrea Gavana. A more detailed explanation of the algorithm is available in the AMPGO paper and specifics for lmfit can be found in the *`ampgo()`* function.

Lmfit uses the external uncertainties (https://github.com/lebigot/uncertainties) package (available on PyPI), instead of distributing its own fork.

An `AbortFitException` is now raised when the fit is aborted by the user (i.e., by using `iter_cb`).

Bugfixes:

- all exceptions are allowed when trying to import matplotlib
- simplify and fix corner-case errors when testing closeness of large integers

## 12.9 Version 0.9.9 Release Notes

Lmfit now uses the asteval (https://github.com/newville/asteval) package instead of distributing its own copy. The minimum required asteval version is 0.9.12, which is available on PyPI. If you see import errors related to asteval, please make sure that you actually have the latest version installed.

## 12.10 Version 0.9.6 Release Notes

Support for SciPy 0.14 has been dropped: SciPy 0.15 is now required. This is especially important for lmfit maintenance, as it means we can now rely on SciPy having code for differential evolution and do not need to keep a local copy.

A brute force method was added, which can be used either with `Minimizer.brute()` or using the `method='brute'` option to `Minimizer.minimize()`. This method requires finite bounds on all varying parameters, or that parameters have a finite `brute_step` attribute set to specify the step size.

Custom cost functions can now be used for the scalar minimizers using the `reduce_fcn` option.

Many improvements to documentation and docstrings in the code were made. As part of that effort, all API documentation in this main Sphinx documentation now derives from the docstrings.

Uncertainties in the resulting best-fit for a model can now be calculated from the uncertainties in the model parameters.

Parameters have two new attributes: `brute_step`, to specify the step size when using the `brute` method, and `user_data`, which is unused but can be used to hold additional information the user may desire. This will be preserved on copy and pickling.

Several bug fixes and cleanups.

Versioneer was updated to 0.18.

Tests can now be run either with nose or pytest.

## 12.11 Version 0.9.5 Release Notes

Support for Python 2.6 and SciPy 0.13 has been dropped.

## 12.12 Version 0.9.4 Release Notes

Some support for the new `least_squares` routine from SciPy 0.17 has been added.

Parameters can now be used directly in floating point or array expressions, so that the Parameter value does not need `sigma = params['sigma'].value`. The older, explicit usage still works, but the docs, samples, and tests have been updated to use the simpler usage.

Support for Python 2.6 and SciPy 0.13 is now explicitly deprecated and will be dropped in version 0.9.5.

## 12.13 Version 0.9.3 Release Notes

Models involving complex numbers have been improved.

The `emcee` module can now be used for uncertainty estimation.

Many bug fixes, and an important fix for performance slowdown on getting parameter values.

ASV benchmarking code added.

## 12.14 Version 0.9.0 Release Notes

This upgrade makes an important, non-backward-compatible change to the way many fitting scripts and programs will work. Scripts that work with version 0.8.3 will not work with version 0.9.0 and vice versa. The change was not made lightly or without ample discussion, and is really an improvement. Modifying scripts that did work with 0.8.3 to work with 0.9.0 is easy, but needs to be done.

### 12.14.1 Summary

The upgrade from 0.8.3 to 0.9.0 introduced the `MinimizerResult` class (see *MinimizerResult – the optimization result*) which is now used to hold the return value from `minimize()` and `Minimizer.minimize()`. This returned object contains many goodness of fit statistics, and holds the optimized parameters from the fit. Importantly, the parameters passed into `minimize()` and `Minimizer.minimize()` are no longer modified by the fit. Instead, a copy of the passed-in parameters is made which is changed and returns as the `params` attribute of the returned `MinimizerResult`.

### 12.14.2 Impact

This upgrade means that a script that does:

```
my_pars = Parameters()
my_pars.add('amp', value=300.0, min=0)
my_pars.add('center', value=5.0, min=0, max=10)
my_pars.add('decay', value=1.0, vary=False)

result = minimize(objfunc, my_pars)
```

will still work, but that `my_pars` will **NOT** be changed by the fit. Instead, `my_pars` is copied to an internal set of parameters that is changed in the fit, and this copy is then put in `result.params`. To look at fit results, use `result.params`, not `my_pars`.

This has the effect that `my_pars` will still hold the starting parameter values, while all of the results from the fit are held in the `result` object returned by `minimize()`.

If you want to do an initial fit, then refine that fit to, for example, do a pre-fit, then refine that result different fitting method, such as:

```
result1 = minimize(objfunc, my_pars, method='nelder')
result1.params['decay'].vary = True
result2 = minimize(objfunc, result1.params, method='leastsq')
```

and have access to all of the starting parameters `my_pars`, the result of the first fit `result1`, and the result of the final fit `result2`.

### 12.14.3 Discussion

The main goal for making this change were to

1. give a better return value to `minimize()` and `Minimizer.minimize()` that can hold all of the information about a fit. By having the return value be an instance of the `MinimizerResult` class, it can hold an arbitrary amount of information that is easily accessed by attribute name, and even be given methods. Using objects is good!
2. To limit or even eliminate the amount of "state information" a `Minimizer` holds. By state information, we mean how much of the previous fit is remembered after a fit is done. Keeping (and especially using) such information about a previous fit means that a `Minimizer` might give different results even for the same problem if run a second time. While it's desirable to be able to adjust a set of `Parameters` re-run a fit to get an improved result, doing this by changing an internal attribute (`Minimizer.params`) has the undesirable side-effect of not being able to "go back", and makes it somewhat cumbersome to keep track of changes made while adjusting parameters and re-running fits.

CHAPTER

# THIRTEEN

# EXAMPLES GALLERY

Below are examples of the different things you can do with lmfit. Click on any image to see the complete source code and output.

We encourage users (i.e., YOU) to submit user-guide-style, documented, and preferably self-contained examples of how you use lmfit for inclusion in this gallery! Please note that many of the examples below currently do *not* follow these guidelines yet.

## 13.1 Fit with Data in a pandas DataFrame

Simple example demonstrating how to read in the data using pandas and supply the elements of the DataFrame from lmfit.

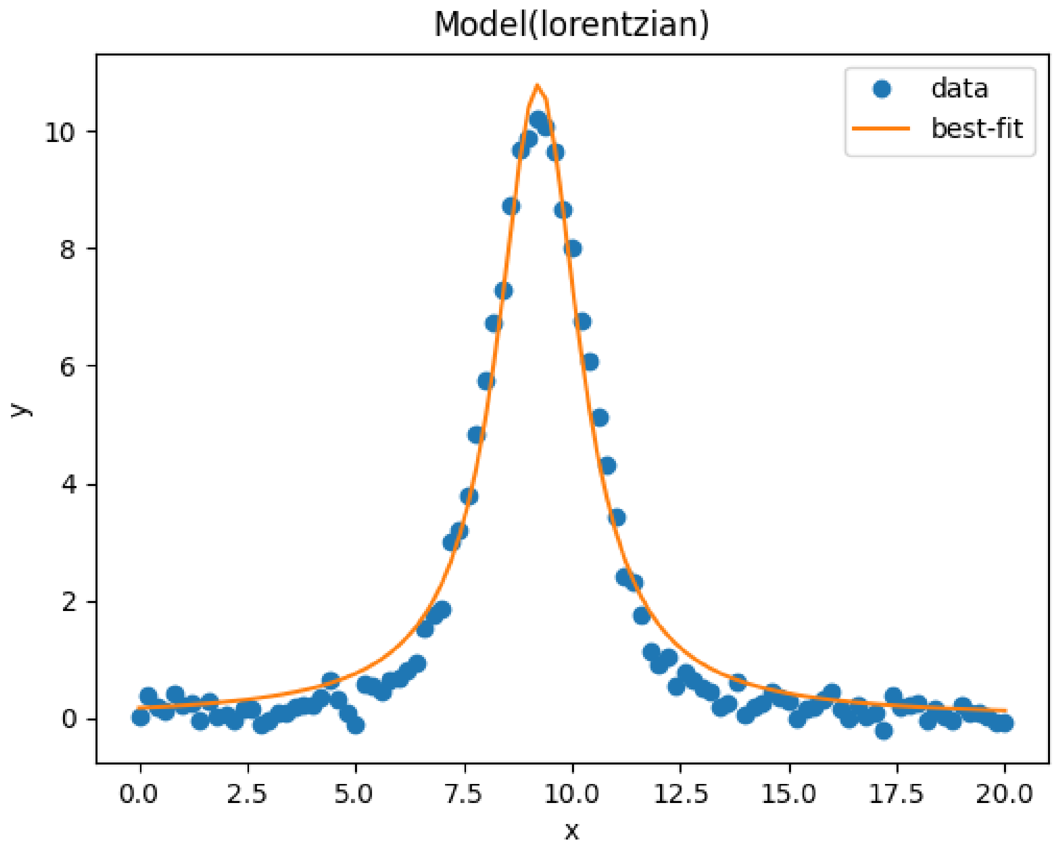
```
import matplotlib.pyplot as plt
import pandas as pd

from lmfit.models import LorentzianModel
```

read the data into a pandas DataFrame, and use the 'x' and 'y' columns:

```
dframe = pd.read_csv('peak.csv')

model = LorentzianModel()
params = model.guess(dframe['y'], x=dframe['x'])

result = model.fit(dframe['y'], params, x=dframe['x'])
```

and gives the plot and fitting results below:

```
result.plot_fit()
plt.show()

print(result.fit_report())
```



Out:

```
[[Model]]
    Model(lorentzian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 21
    # data points      = 101
    # variables        = 3
    chi-square         = 13.0737250
    reduced chi-square = 0.13340536
    Akaike info crit   = -200.496119
    Bayesian info crit = -192.650757
[[Variables]]
    amplitude:  39.1530829 +/- 0.62389698 (1.59%) (init = 50.7825)
    center:     9.22379520 +/- 0.01835869 (0.20%) (init = 9.3)
    sigma:      1.15503893 +/- 0.02603688 (2.25%) (init = 1.3)
    fwhm:       2.31007785 +/- 0.05207377 (2.25%) == '2.0000000*sigma'
    height:     10.7899514 +/- 0.17160472 (1.59%) == '0.3183099*amplitude/max(1e-15,
↪sigma)'
[[Correlations]] (unreported correlations are < 0.100)
    C(amplitude, sigma) =  0.709
```

**Total running time of the script:** ( 0 minutes 0.385 seconds)

## 13.2 Using an ExpressionModel

ExpressionModels allow a model to be built from a user-supplied expression. See: https://lmfit.github.io/lmfit-py/builtin_models.html#user-defined-models

```
import matplotlib.pyplot as plt
import numpy as np

from lmfit.models import ExpressionModel
```

Generate synthetic data for the user-supplied model:

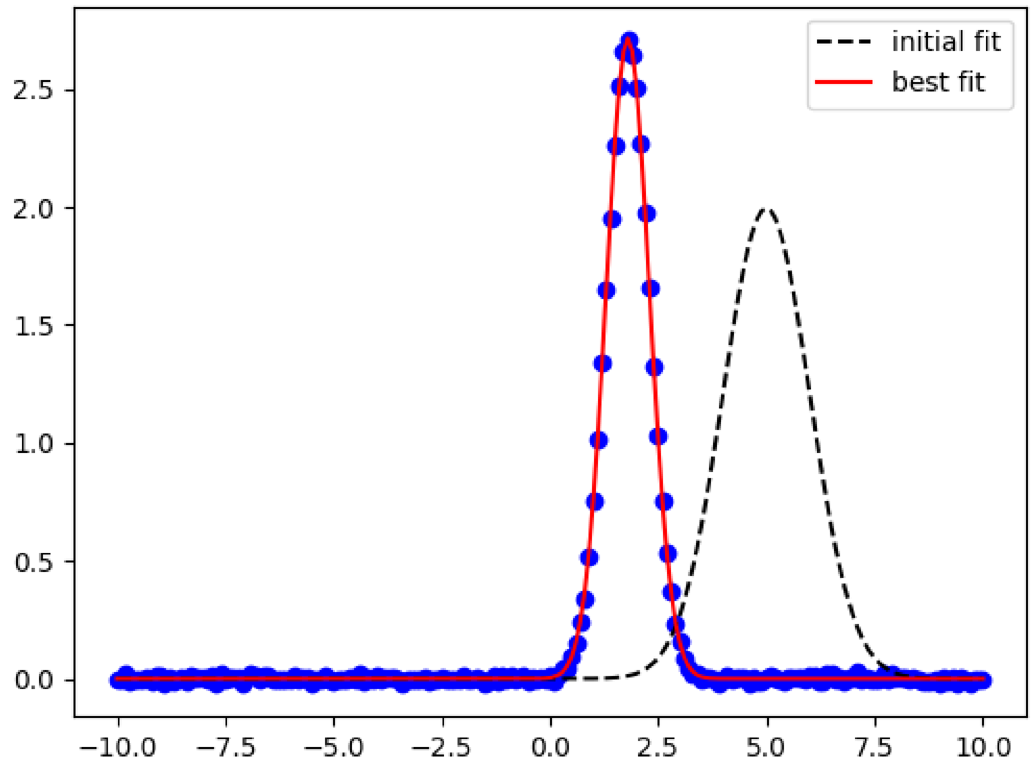
```
x = np.linspace(-10, 10, 201)
amp, cen, wid = 3.4, 1.8, 0.5

y = amp * np.exp(-(x-cen)**2 / (2*wid**2)) / (np.sqrt(2*np.pi)*wid)
y = y + np.random.normal(size=x.size, scale=0.01)
```

Define the ExpressionModel and perform the fit:

```
gmod = ExpressionModel("amp * exp(-(x-cen)**2 /(2*wid**2))/(sqrt(2*pi)*wid)")
result = gmod.fit(y, x=x, amp=5, cen=5, wid=1)
```

this results in the following output:

```
print(result.fit_report())

plt.plot(x, y, 'bo')
plt.plot(x, result.init_fit, 'k--', label='initial fit')
plt.plot(x, result.best_fit, 'r-', label='best fit')
plt.legend(loc='best')
plt.show()
```



Out:

```
[[Model]]
    Model(_eval)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 56
    # data points      = 201
    # variables        = 3
    chi-square         = 0.01825874
    reduced chi-square = 9.2216e-05
    Akaike info crit   = -1864.58964
    Bayesian info crit = -1854.67973
[[Variables]]
    amp:  3.40013669 +/- 0.00494962 (0.15%) (init = 5)
    cen:  1.80087014 +/- 8.3982e-04 (0.05%) (init = 5)
    wid:  0.49962144 +/- 8.3982e-04 (0.17%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(amp, wid) =  0.577
```

**Total running time of the script:** ( 0 minutes 0.283 seconds)

## 13.3 Fit Using Inequality Constraint

Sometimes specifying boundaries using `min` and `max` are not sufficient, and more complicated (inequality) constraints are needed. In the example below the center of the Lorentzian peak is constrained to be between 0-5 away from the center of the Gaussian peak.

See also: https://lmfit.github.io/lmfit-py/constraints.html#using-inequality-constraints

```
import matplotlib.pyplot as plt
import numpy as np

from lmfit import Minimizer, Parameters, report_fit
from lmfit.lineshapes import gaussian, lorentzian


def residual(pars, x, data):
    model = (gaussian(x, pars['amp_g'], pars['cen_g'], pars['wid_g']) +
             lorentzian(x, pars['amp_l'], pars['cen_l'], pars['wid_l']))
    return model - data
```

Generate the simulated data using a Gaussian and Lorentzian line shape:

```
np.random.seed(0)
x = np.linspace(0, 20.0, 601)

data = (gaussian(x, 21, 6.1, 1.2) + lorentzian(x, 10, 9.6, 1.3) +
        np.random.normal(scale=0.1, size=x.size))
```

Create the fitting parameters and set an inequality constraint for `cen_l`. First, we add a new fitting parameter `peak_split`, which can take values between 0 and 5. Afterwards, we constrain the value for `cen_l` using the expression to be `'peak_split+cen_g'`:

```
pfit = Parameters()
pfit.add(name='amp_g', value=10)
pfit.add(name='amp_l', value=10)
pfit.add(name='cen_g', value=5)
pfit.add(name='peak_split', value=2.5, min=0, max=5, vary=True)
pfit.add(name='cen_l', expr='peak_split+cen_g')
pfit.add(name='wid_g', value=1)
pfit.add(name='wid_l', expr='wid_g')

mini = Minimizer(residual, pfit, fcn_args=(x, data))
out = mini.leastsq()
best_fit = data + out.residual
```

Performing a fit, here using the `leastsq` algorithm, gives the following fitting results:

```
report_fit(out.params)
```

Out:

```
[[Variables]]
    amp_g:       21.2722837 +/- 0.05138760 (0.24%) (init = 10)
    amp_l:       9.46504191 +/- 0.05445416 (0.58%) (init = 10)
    cen_g:       6.10496394 +/- 0.00334614 (0.05%) (init = 5)
    peak_split:  3.52163535 +/- 0.01004618 (0.29%) (init = 2.5)
    cen_l:       9.62659929 +/- 0.01066173 (0.11%) == 'peak_split+cen_g'
```

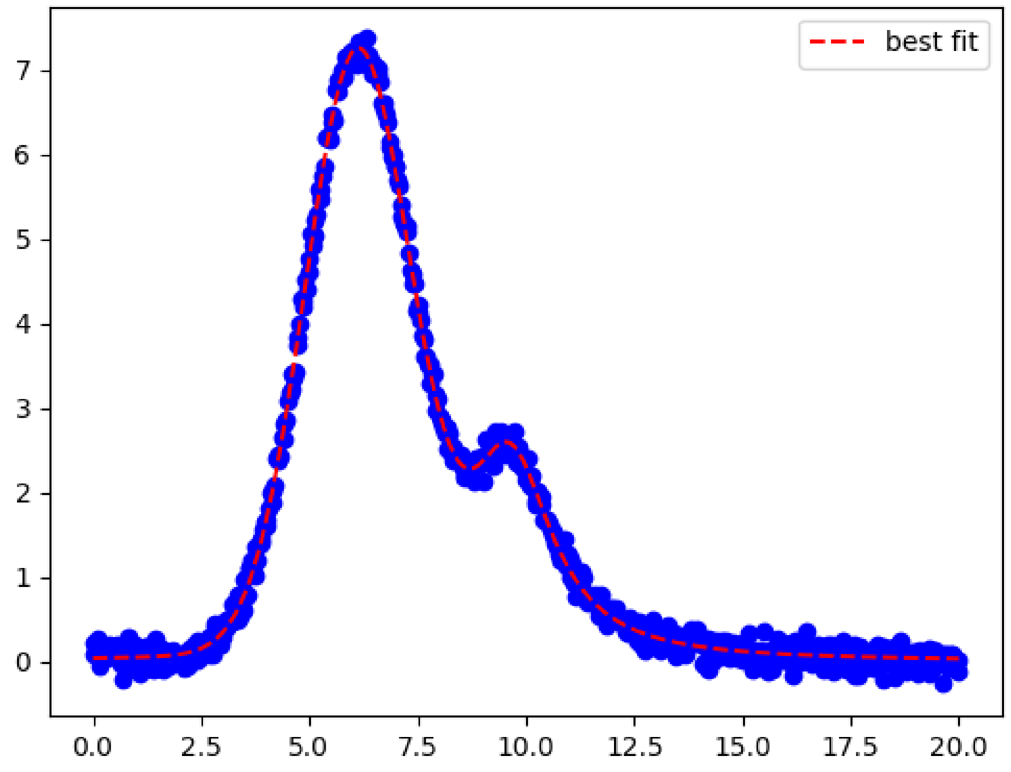
(continues on next page)

(continued from previous page)

```
    wid_g:        1.21434950 +/- 0.00327315 (0.27%) (init = 1)
    wid_l:        1.21434950 +/- 0.00327315 (0.27%) == 'wid_g'
[[Correlations]] (unreported correlations are < 0.100)
    C(amp_g, wid_g)        =  0.620
    C(amp_g, peak_split) =  0.380
    C(peak_split, wid_g) =  0.344
    C(amp_g, amp_l)        = -0.295
    C(amp_l, cen_g)        = -0.276
    C(amp_g, cen_g)        =  0.194
    C(amp_l, wid_g)        = -0.165
    C(cen_g, wid_g)        =  0.155
```

and figure:

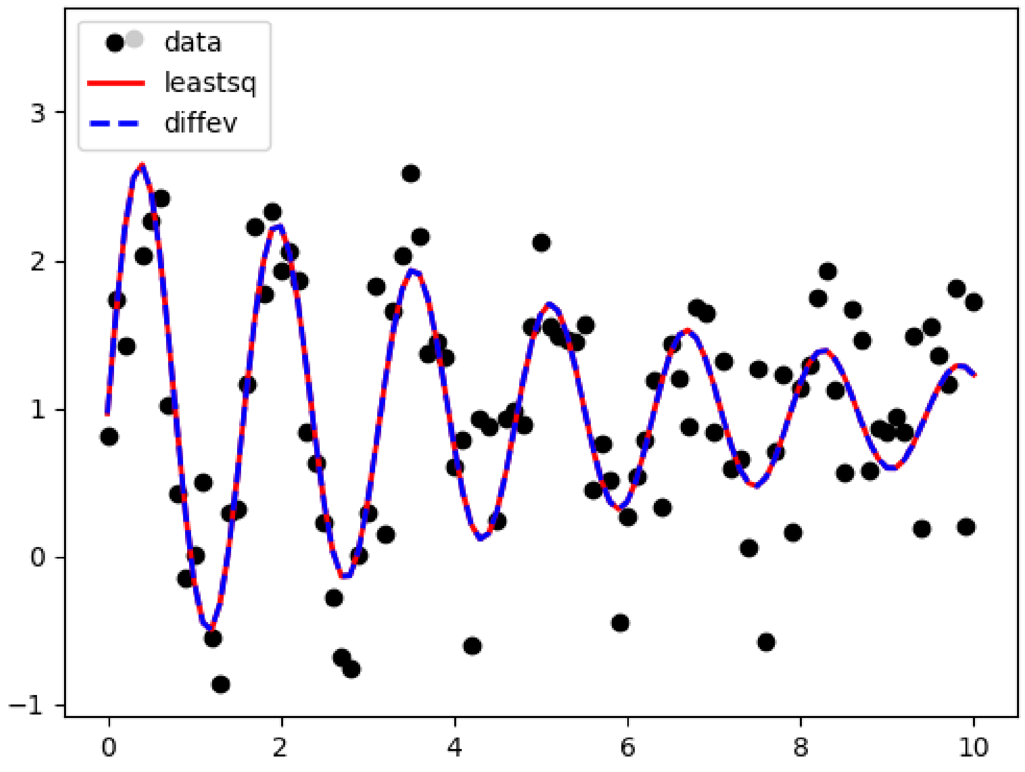
```
plt.plot(x, data, 'bo')
plt.plot(x, best_fit, 'r--', label='best fit')
plt.legend(loc='best')
plt.show()
```



**Total running time of the script:** ( 0 minutes 0.303 seconds)

## 13.4 Fit Using differential_evolution Algorithm

This example compares the "leastsq" and "differential_evolution" algorithms on a fairly simple problem.



Out:

```
# Fit using leastsq:
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 65
    # data points      = 101
    # variables        = 4
    chi-square         = 21.7961792
    reduced chi-square = 0.22470288
    Akaike info crit   = -146.871969
    Bayesian info crit = -136.411487
[[Variables]]
    offset:  0.96333090 +/- 0.04735921 (4.92%) (init = 2)
    omega:   3.98700821 +/- 0.02079710 (0.52%) (init = 3.3)
    amp:     1.80253577 +/- 0.19401989 (10.76%) (init = 2.5)
    decay:   5.76279825 +/- 1.04073303 (18.06%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(amp, decay) = -0.755
```

(continues on next page)

(continued from previous page)

```
# Fit using differential_evolution:
[[Fit Statistics]]
    # fitting method   = differential_evolution
    # function evals   = 1425
    # data points      = 101
    # variables        = 4
    chi-square         = 21.7961792
    reduced chi-square = 0.22470288
    Akaike info crit   = -146.871969
    Bayesian info crit = -136.411487
[[Variables]]
    offset:  0.96333133 +/- 0.04735903 (4.92%) (init = 2)
    omega:   3.98700854 +/- 0.02121810 (0.53%) (init = 3.3)
    amp:     1.80252618 +/- 0.19022408 (10.55%) (init = 2.5)
    decay:   5.76284502 +/- 1.00452606 (17.43%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(amp, decay) = -0.743
```

```
import matplotlib.pyplot as plt
import numpy as np

import lmfit

np.random.seed(2)
x = np.linspace(0, 10, 101)

# Setup example
decay = 5
offset = 1.0
amp = 2.0
omega = 4.0

y = offset + amp*np.sin(omega*x) * np.exp(-x/decay)
yn = y + np.random.normal(size=y.size, scale=0.450)


def resid(params, x, ydata):
    decay = params['decay'].value
    offset = params['offset'].value
    omega = params['omega'].value
    amp = params['amp'].value

    y_model = offset + amp * np.sin(x*omega) * np.exp(-x/decay)
    return y_model - ydata


params = lmfit.Parameters()
params.add('offset', 2.0, min=0, max=10.0)
params.add('omega', 3.3, min=0, max=10.0)
params.add('amp', 2.5, min=0, max=10.0)
params.add('decay', 1.0, min=0, max=10.0)
```

(continues on next page)

(continued from previous page)

```
o1 = lmfit.minimize(resid, params, args=(x, yn), method='leastsq')
print("# Fit using leastsq:")
lmfit.report_fit(o1)

o2 = lmfit.minimize(resid, params, args=(x, yn), method='differential_evolution')
print("\n\n# Fit using differential_evolution:")
lmfit.report_fit(o2)

plt.plot(x, yn, 'ko', lw=2)
plt.plot(x, yn+o1.residual, 'r-', lw=2)
plt.plot(x, yn+o2.residual, 'b--', lw=2)
plt.legend(['data', 'leastsq', 'diffev'], loc='upper left')
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.878 seconds)

## 13.5 Fit Using Bounds

A major advantage of using lmfit is that one can specify boundaries on fitting parameters, even if the underlying algorithm in SciPy does not support this. For more information on how this is implemented, please refer to: https://lmfit.github.io/lmfit-py/bounds.html

The example below shows how to set boundaries using the `min` and `max` attributes to fitting parameters.

```
import matplotlib.pyplot as plt
from numpy import exp, linspace, pi, random, sign, sin

from lmfit import Parameters, minimize
from lmfit.printfuncs import report_fit

p_true = Parameters()
p_true.add('amp', value=14.0)
p_true.add('period', value=5.4321)
p_true.add('shift', value=0.12345)
p_true.add('decay', value=0.01000)


def residual(pars, x, data=None):
    argu = (x * pars['decay'])**2
    shift = pars['shift']
    if abs(shift) > pi/2:
        shift = shift - sign(shift)*pi
    model = pars['amp'] * sin(shift + x/pars['period']) * exp(-argu)
    if data is None:
        return model
    return model - data


random.seed(0)
x = linspace(0, 250, 1500)
noise = random.normal(scale=2.80, size=x.size)
data = residual(p_true, x) + noise

fit_params = Parameters()
```

(continues on next page)

(continued from previous page)

```
fit_params.add('amp', value=13.0, max=20, min=0.0)
fit_params.add('period', value=2, max=10)
fit_params.add('shift', value=0.0, max=pi/2., min=-pi/2.)
fit_params.add('decay', value=0.02, max=0.10, min=0.00)

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

This gives the following fitting results:

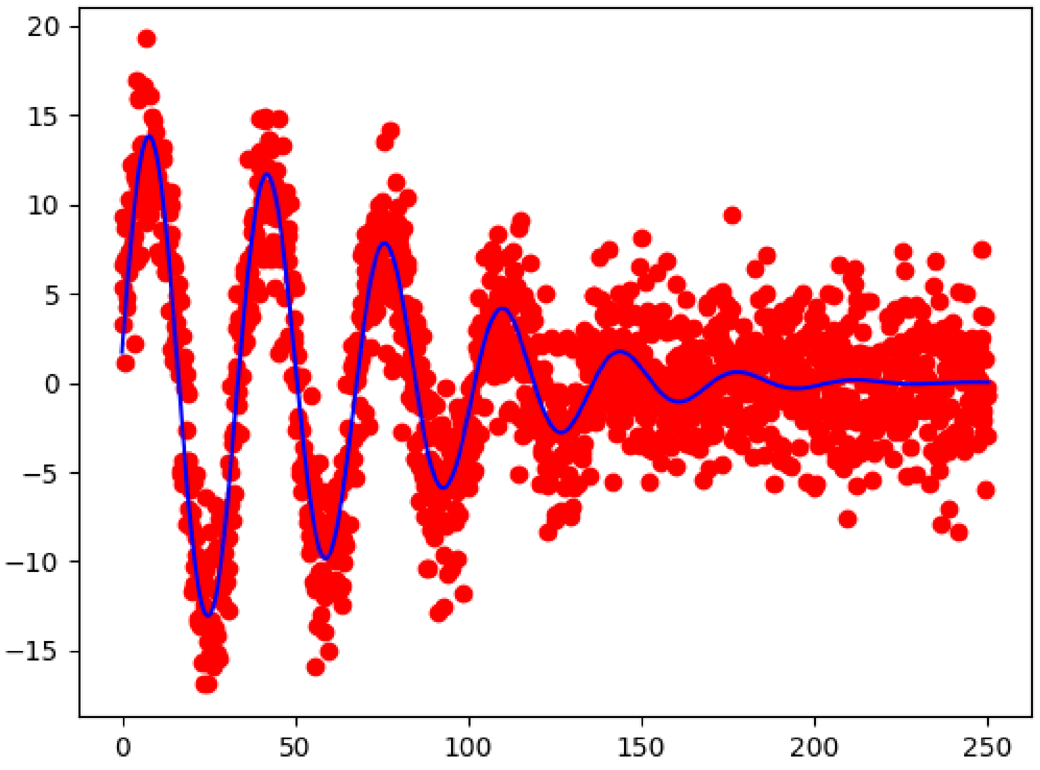
```
report_fit(out, show_correl=True, modelpars=p_true)
```

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 74
    # data points      = 1500
    # variables        = 4
    chi-square         = 11301.3646
    reduced chi-square = 7.55438813
    Akaike info crit   = 3037.18756
    Bayesian info crit = 3058.44044
[[Variables]]
    amp:     13.8903938 +/- 0.24412383 (1.76%) (init = 13), model_value = 14
    period:  5.44026442 +/- 0.01416175 (0.26%) (init = 2), model_value = 5.4321
    shift:   0.12464470 +/- 0.02414209 (19.37%) (init = 0), model_value = 0.12345
    decay:   0.00996351 +/- 2.0278e-04 (2.04%) (init = 0.02), model_value = 0.01
[[Correlations]] (unreported correlations are < 0.100)
    C(period, shift) =  0.800
    C(amp, decay)    =  0.576
```

and shows the plot below:

```
plt.plot(x, data, 'ro')
plt.plot(x, fit, 'b')
plt.show()
```

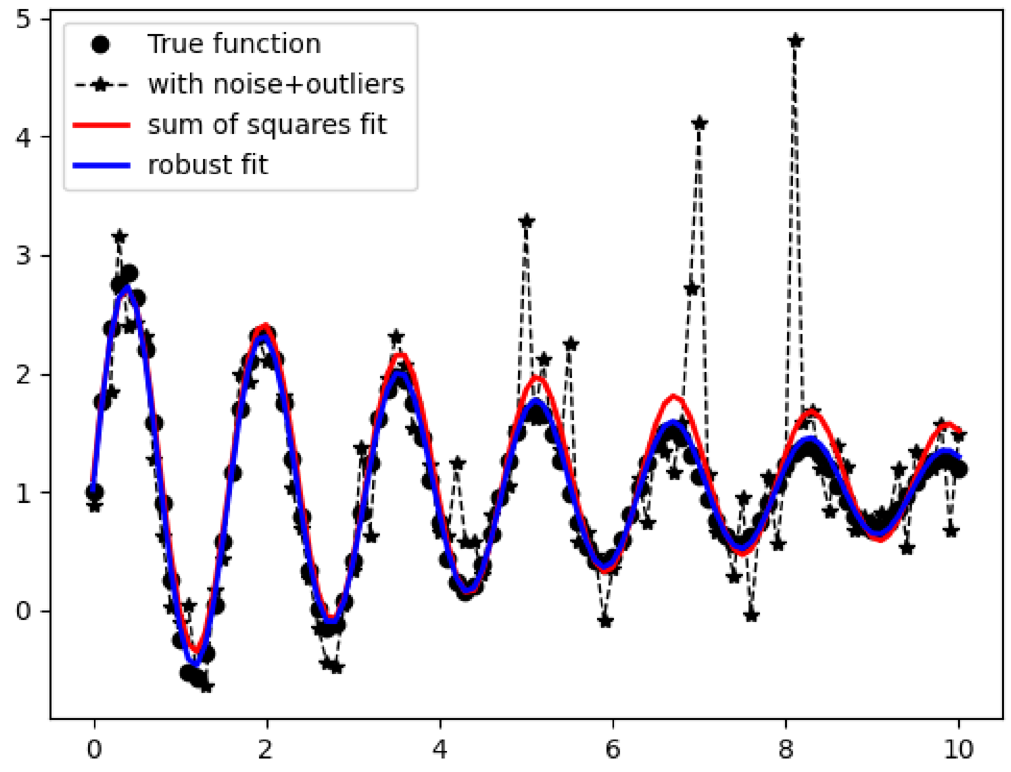


**Total running time of the script:** ( 0 minutes 0.277 seconds)

## 13.6 Fit Specifying Different Reduce Function

The reduce_fcn specifies how to convert a residual array to a scalar value for the scalar minimizers. The default value is None (i.e., "sum of squares of residual") - alternatives are: 'negentropy' and 'neglogcauchy' or a user-specified "callable". For more information please refer to: https://lmfit.github.io/lmfit-py/fitting.html#using-the-minimizer-class

Here, we use as an example the Student's t log-likelihood for robust fitting of data with outliers.



Out:

```
# Fit using sum of squares:

[[Fit Statistics]]
    # fitting method   = L-BFGS-B
    # function evals   = 130
    # data points      = 101
    # variables        = 4
    chi-square         = 32.1674767
    reduced chi-square = 0.33162347
    Akaike info crit   = -107.560626
    Bayesian info crit = -97.1001440
[[Variables]]
    offset:  1.10392445 +/- 0.05751441 (5.21%) (init = 2)
    omega:   3.97313428 +/- 0.02073920 (0.52%) (init = 3.3)
    amp:     1.69977036 +/- 0.21587450 (12.70%) (init = 2.5)
    decay:   7.65901704 +/- 1.87208973 (24.44%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(amp, decay) = -0.773


# Robust Fit, using log-likelihood with Cauchy PDF:

[[Fit Statistics]]
    # fitting method   = L-BFGS-B
```

(continues on next page)

(continued from previous page)

```
    # function evals   = 130
    # data points      = 101
    # variables        = 4
    chi-square         = 33.5081438
    reduced chi-square = 0.34544478
    Akaike info crit   = -103.436533
    Bayesian info crit = -92.9760507
[[Variables]]
    offset:  1.02005947 +/- 0.06642640 (6.51%) (init = 2)
    omega:   3.98224468 +/- 0.02898703 (0.73%) (init = 3.3)
    amp:     1.83231389 +/- 0.27241853 (14.87%) (init = 2.5)
    decay:   5.77327242 +/- 1.45140477 (25.14%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(amp, decay)   = -0.758
    C(offset, amp)  = -0.107
```

```
import matplotlib.pyplot as plt
import numpy as np

import lmfit

np.random.seed(2)
x = np.linspace(0, 10, 101)

# Setup example
decay = 5
offset = 1.0
amp = 2.0
omega = 4.0

y = offset + amp * np.sin(omega*x) * np.exp(-x/decay)
yn = y + np.random.normal(size=y.size, scale=0.250)

outliers = np.random.randint(int(len(x)/3.0), len(x), int(len(x)/12))
yn[outliers] += 5*np.random.random(len(outliers))


def resid(params, x, ydata):
    decay = params['decay'].value
    offset = params['offset'].value
    omega = params['omega'].value
    amp = params['amp'].value

    y_model = offset + amp * np.sin(x*omega) * np.exp(-x/decay)
    return y_model - ydata


params = lmfit.Parameters()

params.add('offset', 2.0)
params.add('omega', 3.3)
```

(continues on next page)

(continued from previous page)

```
params.add('amp', 2.5)
params.add('decay', 1.0, min=0)

method = 'L-BFGS-B'

o1 = lmfit.minimize(resid, params, args=(x, yn), method=method)
print("# Fit using sum of squares:\n")
lmfit.report_fit(o1)

o2 = lmfit.minimize(resid, params, args=(x, yn), method=method,
                    reduce_fcn='neglogcauchy')
print("\n\n# Robust Fit, using log-likelihood with Cauchy PDF:\n")
lmfit.report_fit(o2)

plt.plot(x, y, 'ko', lw=2)
plt.plot(x, yn, 'k--*', lw=1)
plt.plot(x, yn+o1.residual, 'r-', lw=2)
plt.plot(x, yn+o2.residual, 'b-', lw=2)
plt.legend(['True function',
            'with noise+outliers',
            'sum of squares fit',
            'robust fit'], loc='upper left')
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.478 seconds)

## 13.7 Building a lmfit model with SymPy

SymPy is a Python library for symbolic mathematics. It can be very useful to build a model with SymPy and then apply that model to the data with lmfit. This example shows how to do that. Please note that this example requires both the sympy and matplotlib packages.

```
import matplotlib.pyplot as plt
import numpy as np
import sympy
from sympy.parsing import sympy_parser

import lmfit

np.random.seed(1)
```

Instead of creating the SymPy symbols explicitly and building an expression with them, we will use the SymPy parser.

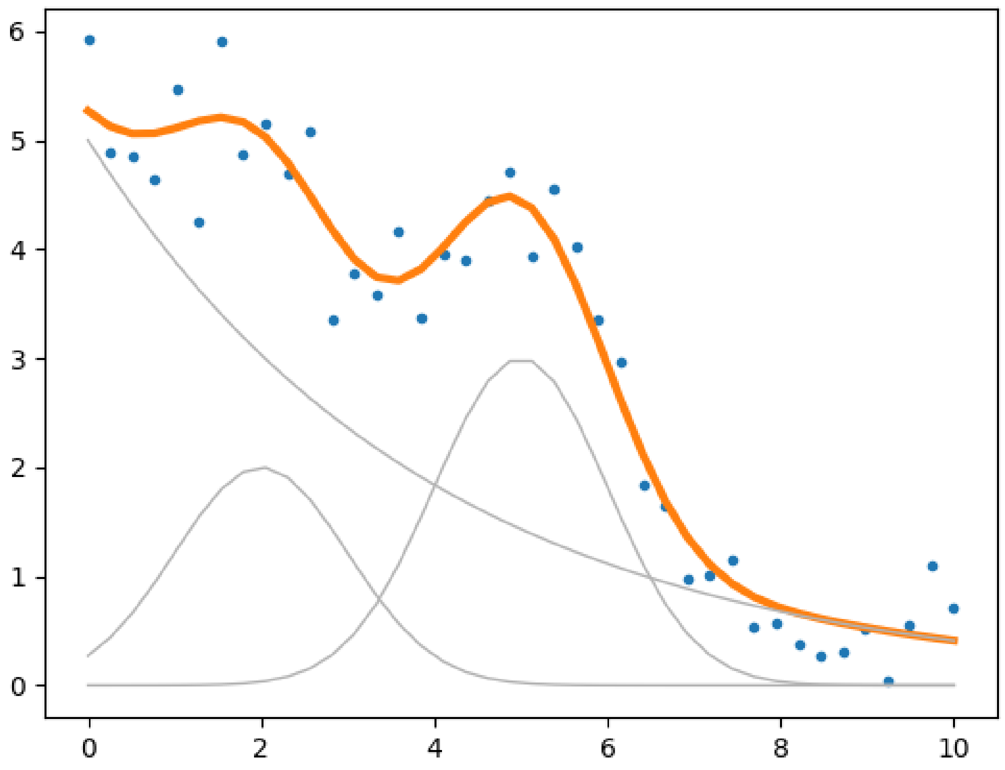
```
gauss_peak1 = sympy_parser.parse_expr('A1*exp(-(x-xc1)**2/(2*sigma1**2))')
gauss_peak2 = sympy_parser.parse_expr('A2*exp(-(x-xc2)**2/(2*sigma2**2))')
exp_back = sympy_parser.parse_expr('B*exp(-x/xw)')

model_list = sympy.Array((gauss_peak1, gauss_peak2, exp_back))
model = sum(model_list)
print(model)
```

Out:

```
A1*exp(-(x - xc1)**2/(2*sigma1**2)) + A2*exp(-(x - xc2)**2/(2*sigma2**2)) + B*exp(-x/
↪xw)
```

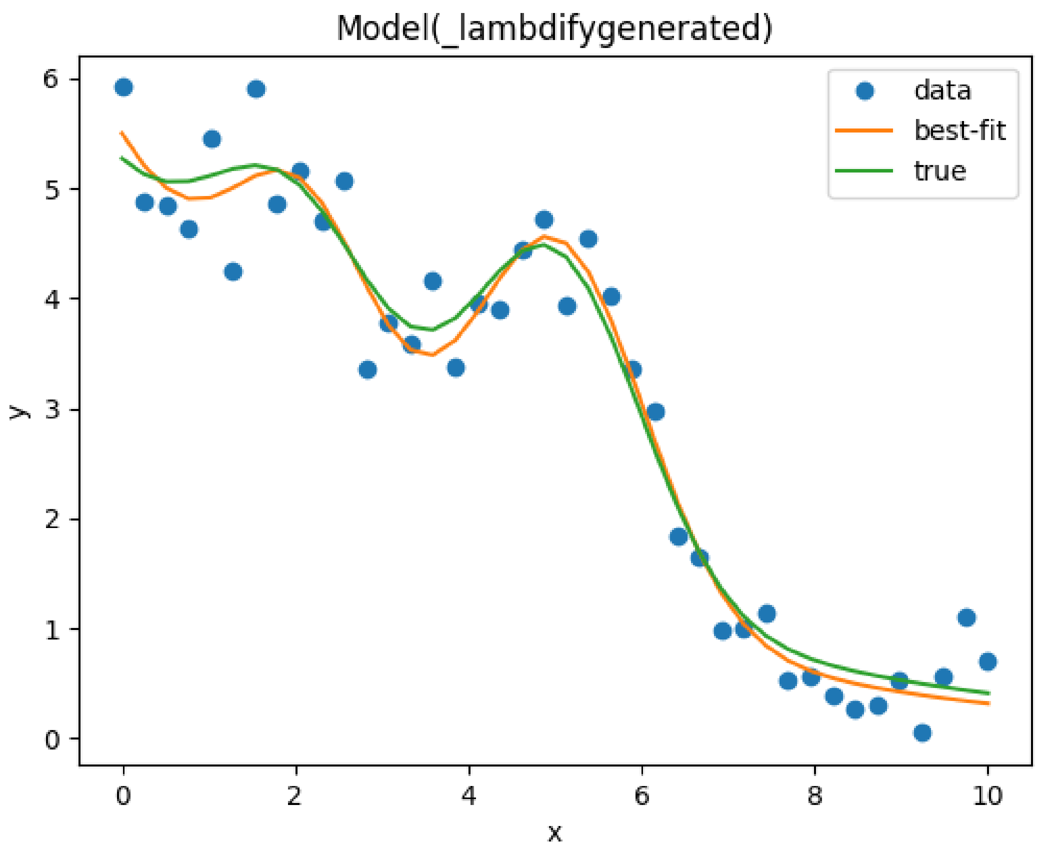
We are using SymPy's lambdify function to make a function from the model expressions. We then use these functions to generate some fake data.

```
model_list_func = sympy.lambdify(list(model_list.free_symbols), model_list)
model_func = sympy.lambdify(list(model.free_symbols), model)

x = np.linspace(0, 10, 40)
param_values = dict(x=x, A1=2, sigma1=1, sigma2=1, A2=3,
                    xc1=2, xc2=5, xw=4, B=5)
y = model_func(**param_values)
yi = model_list_func(**param_values)
yn = y + np.random.randn(y.size)*0.4

plt.plot(x, yn, 'o', zorder=1.9, ms=3)
plt.plot(x, y, lw=3)
for c in yi:
    plt.plot(x, c, lw=1, c='0.7')
```



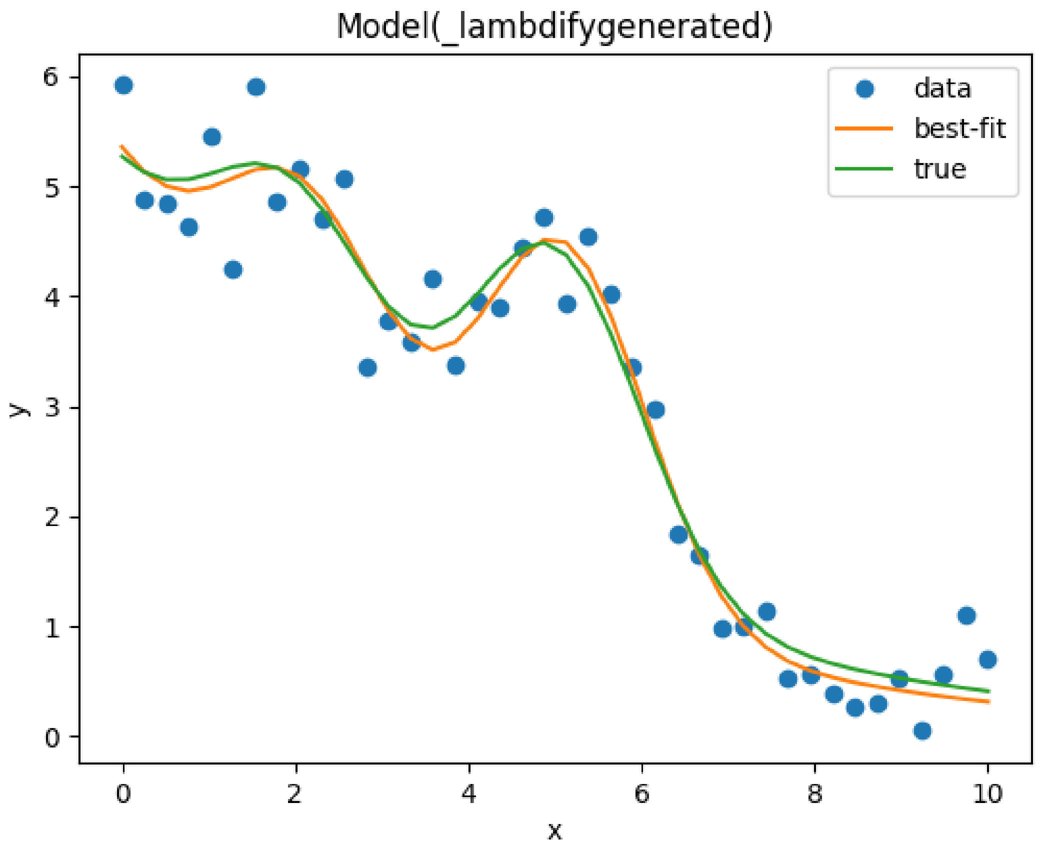
Next, we will just create a lmfit model from the function and fit the data.

```
lm_mod = lmfit.Model(model_func, independent_vars=('x'))
res = lm_mod.fit(data=yn, **param_values)
res.plot_fit()
plt.plot(x, y, label='true')
plt.legend()
plt.show()
```



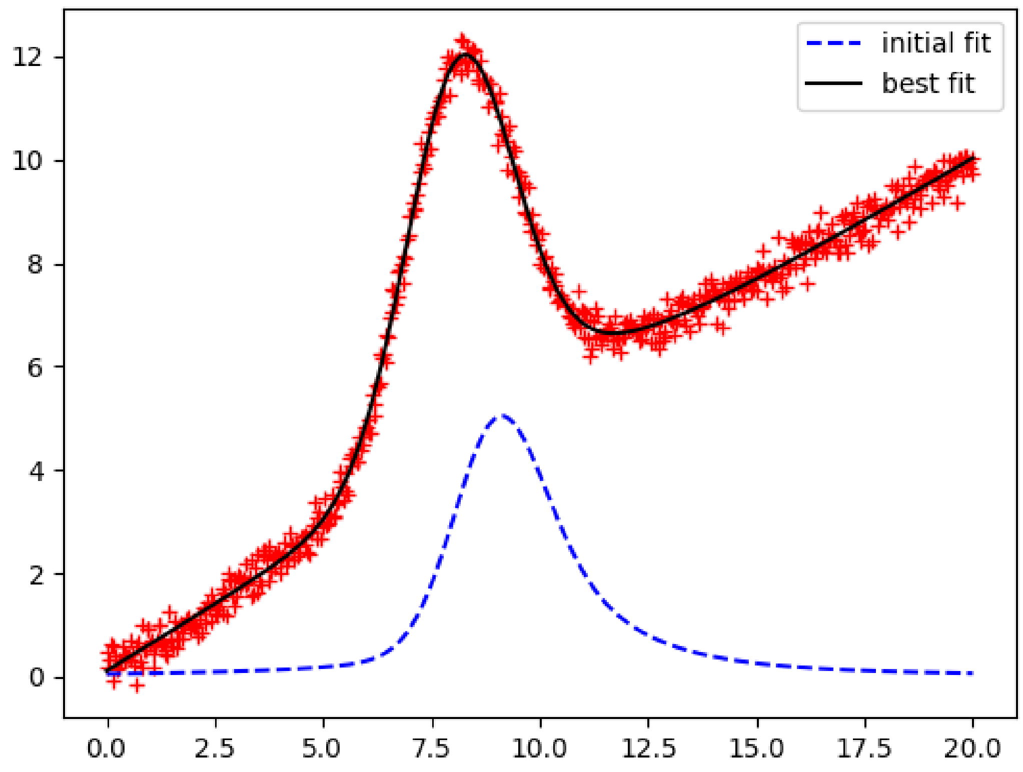
The nice thing of using SymPy is that we can easily modify our fit function. Let's assume we know that the width of both Gaussians are identical. Similarly, we assume that the ratio between both Gaussians is fixed to 3:2 for some reason. Both can be expressed by just substituting the variables.

```
model2 = model.subs('sigma2', 'sigma1').subs('A2', '3/2*A1')
model2_func = sympy.lambdify(list(model2.free_symbols), model2)
lm_mod = lmfit.Model(model2_func, independent_vars=('x'))
param2_values = dict(x=x, A1=2, sigma1=1, xc1=2, xc2=5, xw=4, B=5)
res2 = lm_mod.fit(data=yn, **param2_values)
res2.plot_fit()
plt.plot(x, y, label='true')
plt.legend()
plt.show()
```



**Total running time of the script:** ( 0 minutes 1.919 seconds)

## 13.8 Fit with Algebraic Constraint



Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 53
    # data points      = 601
    # variables        = 6
    chi-square         = 71878.3055
    reduced chi-square = 120.803875
    Akaike info crit   = 2887.26503
    Bayesian info crit = 2913.65660
[[Variables]]
    amp_g:       21.1877704 +/- 0.32191824 (1.52%) (init = 10)
    cen_g:       8.11125923 +/- 0.01162984 (0.14%) (init = 9)
    wid_g:       1.20925846 +/- 0.01170853 (0.97%) (init = 1)
    amp_tot:     30.6003738 +/- 0.36481391 (1.19%) (init = 20)
    amp_l:       9.41260340 +/- 0.61672681 (6.55%) == 'amp_tot - amp_g'
    cen_l:       9.61125923 +/- 0.01162984 (0.12%) == '1.5+cen_g'
    wid_l:       2.41851692 +/- 0.02341706 (0.97%) == '2*wid_g'
    line_slope:  0.49615727 +/- 0.00170178 (0.34%) (init = 0)
    line_off:    0.04128591 +/- 0.02448064 (59.30%) (init = 0)
[[Correlations]] (unreported correlations are < 0.100)
    C(amp_g, wid_g)          =  0.866
```

(continues on next page)

(continued from previous page)

```
    C(amp_g, cen_g)          =  0.750
    C(line_slope, line_off) = -0.714
    C(cen_g, amp_tot)       = -0.695
    C(cen_g, wid_g)         =  0.623
    C(amp_g, amp_tot)       = -0.612
    C(amp_tot, line_off)    = -0.588
    C(wid_g, amp_tot)       = -0.412
    C(cen_g, line_off)      =  0.387
    C(amp_g, line_off)      =  0.183
    C(amp_g, line_slope)    =  0.183
    C(wid_g, line_slope)    =  0.174
```

```
import matplotlib.pyplot as plt
from numpy import linspace, random

from lmfit import Minimizer, Parameters
from lmfit.lineshapes import gaussian, lorentzian
from lmfit.printfuncs import report_fit


def residual(pars, x, sigma=None, data=None):
    yg = gaussian(x, pars['amp_g'], pars['cen_g'], pars['wid_g'])
    yl = lorentzian(x, pars['amp_l'], pars['cen_l'], pars['wid_l'])

    slope = pars['line_slope']
    offset = pars['line_off']
    model = yg + yl + offset + x*slope

    if data is None:
        return model
    if sigma is None:
        return model - data
    return (model - data) / sigma


random.seed(0)
x = linspace(0.0, 20.0, 601)

data = (gaussian(x, 21, 8.1, 1.2) +
        lorentzian(x, 10, 9.6, 2.4) +
        random.normal(scale=0.23, size=x.size) +
        x*0.5)


pfit = Parameters()
pfit.add(name='amp_g', value=10)
pfit.add(name='cen_g', value=9)
pfit.add(name='wid_g', value=1)
pfit.add(name='amp_tot', value=20)
pfit.add(name='amp_l', expr='amp_tot - amp_g')
pfit.add(name='cen_l', expr='1.5+cen_g')
```

(continues on next page)

(continued from previous page)

```
pfit.add(name='wid_l', expr='2*wid_g')
pfit.add(name='line_slope', value=0.0)
pfit.add(name='line_off', value=0.0)

sigma = 0.021  # estimate of data error (for all data points)

myfit = Minimizer(residual, pfit,
                  fcn_args=(x,), fcn_kws={'sigma': sigma, 'data': data},
                  scale_covar=True)

result = myfit.leastsq()
init = residual(pfit, x)
fit = residual(result.params, x)

report_fit(result)

plt.plot(x, data, 'r+')
plt.plot(x, init, 'b--', label='initial fit')
plt.plot(x, fit, 'k-', label='best fit')
plt.legend(loc='best')
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.327 seconds)

## 13.9 Fit Multiple Data Sets

Fitting multiple (simulated) Gaussian data sets simultaneously.

All minimizers require the residual array to be one-dimensional. Therefore, in the `objective` we need to ``flatten`` the array before returning it.

TODO: this should be using the Model interface / built-in models!

```
import matplotlib.pyplot as plt
import numpy as np

from lmfit import Parameters, minimize, report_fit


def gauss(x, amp, cen, sigma):
    """Gaussian lineshape."""
    return amp * np.exp(-(x-cen)**2 / (2.*sigma**2))


def gauss_dataset(params, i, x):
    """Calculate Gaussian lineshape from parameters for data set."""
    amp = params['amp_%i' % (i+1)]
    cen = params['cen_%i' % (i+1)]
    sig = params['sig_%i' % (i+1)]
    return gauss(x, amp, cen, sig)


def objective(params, x, data):
    """Calculate total residual for fits of Gaussians to several data sets."""
    ndata, _ = data.shape
```

(continues on next page)

(continued from previous page)

```
    resid = 0.0*data[:]

    # make residual per data set
    for i in range(ndata):
        resid[i, :] = data[i, :] - gauss_dataset(params, i, x)

    # now flatten this to a 1D array, as minimize() needs
    return resid.flatten()
```

Create five simulated Gaussian data sets

```
x = np.linspace(-1, 2, 151)
data = []
for _ in np.arange(5):
    params = Parameters()
    amp = 0.60 + 9.50*np.random.rand()
    cen = -0.20 + 1.20*np.random.rand()
    sig = 0.25 + 0.03*np.random.rand()
    dat = gauss(x, amp, cen, sig) + np.random.normal(size=x.size, scale=0.1)
    data.append(dat)
data = np.array(data)
```

Create five sets of fitting parameters, one per data set

```
fit_params = Parameters()
for iy, y in enumerate(data):
    fit_params.add('amp_%i' % (iy+1), value=0.5, min=0.0, max=200)
    fit_params.add('cen_%i' % (iy+1), value=0.4, min=-2.0, max=2.0)
    fit_params.add('sig_%i' % (iy+1), value=0.3, min=0.01, max=3.0)
```

Constrain the values of sigma to be the same for all peaks by assigning sig_2, . . . , sig_5 to be equal to sig_1.

```
for iy in (2, 3, 4, 5):
    fit_params['sig_%i' % iy].expr = 'sig_1'
```

Run the global fit and show the fitting result

```
out = minimize(objective, fit_params, args=(x, data))
report_fit(out.params)
```

Out:

```
[[Variables]]
    amp_1:  6.35769745 +/- 0.02471391 (0.39%) (init = 0.5)
    cen_1: -0.06087255 +/- 0.00141148 (2.32%) (init = 0.4)
    sig_1:  0.27096733 +/- 6.7335e-04 (0.25%) (init = 0.3)
    amp_2:  6.23505243 +/- 0.02466565 (0.40%) (init = 0.5)
    cen_2:  0.90331537 +/- 0.00143924 (0.16%) (init = 0.4)
    sig_2:  0.27096733 +/- 6.7335e-04 (0.25%) == 'sig_1'
    amp_3:  6.74510330 +/- 0.02487197 (0.37%) (init = 0.5)
    cen_3:  0.30698606 +/- 0.00133040 (0.43%) (init = 0.4)
    sig_3:  0.27096733 +/- 6.7335e-04 (0.25%) == 'sig_1'
    amp_4:  3.62962270 +/- 0.02384778 (0.66%) (init = 0.5)
    cen_4:  0.00271542 +/- 0.00247235 (91.05%) (init = 0.4)
    sig_4:  0.27096733 +/- 6.7335e-04 (0.25%) == 'sig_1'
    amp_5:  6.29975266 +/- 0.02469084 (0.39%) (init = 0.5)
```

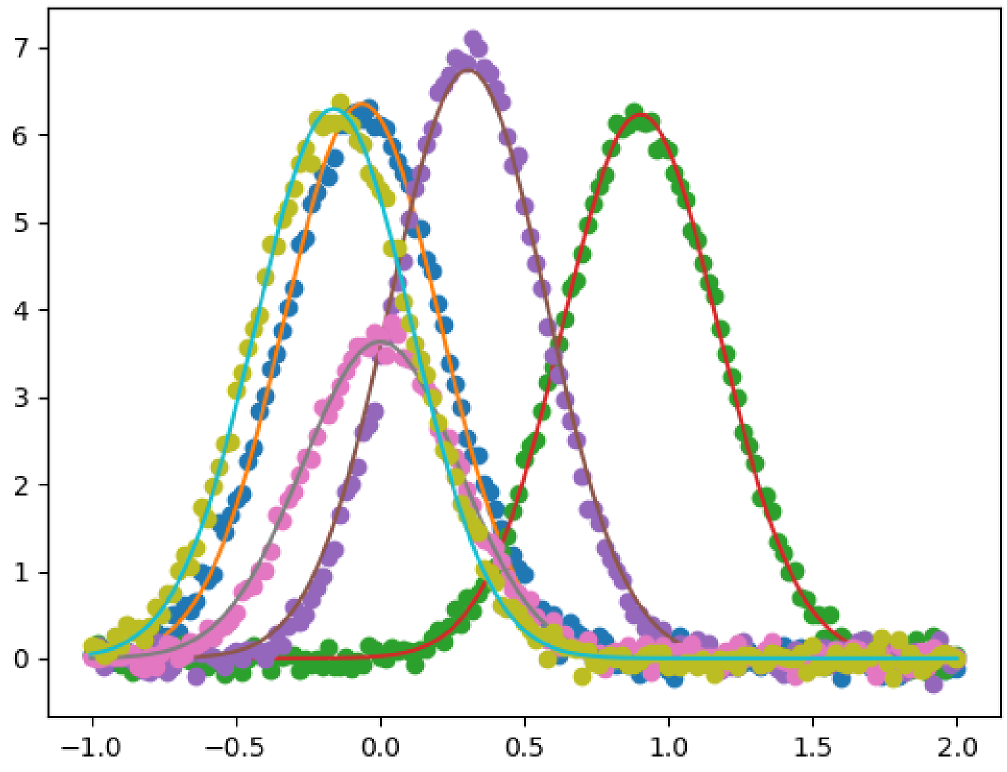
(continues on next page)

(continued from previous page)

```
    cen_5: -0.15885172 +/- 0.00142452 (0.90%) (init = 0.4)
    sig_5:  0.27096733 +/- 6.7335e-04 (0.25%) == 'sig_1'
[[Correlations]] (unreported correlations are < 0.100)
    C(sig_1, amp_3) = -0.337
    C(amp_1, sig_1) = -0.320
    C(sig_1, amp_5) = -0.317
    C(sig_1, amp_2) = -0.314
    C(sig_1, amp_4) = -0.189
    C(amp_1, amp_3) =  0.108
    C(amp_3, amp_5) =  0.107
    C(amp_2, amp_3) =  0.106
    C(amp_1, amp_5) =  0.101
    C(amp_1, amp_2) =  0.100
```

Plot the data sets and fits

```
plt.figure()
for i in range(5):
    y_fit = gauss_dataset(out.params, i, x)
    plt.plot(x, data[i, :], 'o', x, y_fit, '-')
plt.show()
```



**Total running time of the script:** ( 0 minutes 0.447 seconds)

## 13.10 Fit Specifying a Function to Compute the Jacobian

Specifying an analytical function to calculate the Jacobian can speed-up the fitting procedure.

```
import matplotlib.pyplot as plt
import numpy as np

from lmfit import Minimizer, Parameters


def func(pars, x, data=None):
    a, b, c = pars['a'], pars['b'], pars['c']
    model = a * np.exp(-b*x) + c
    if data is None:
        return model
    return model - data


def dfunc(pars, x, data=None):
    a, b = pars['a'], pars['b']
    v = np.exp(-b*x)
    return np.array([v, -a*x*v, np.ones(len(x))])


def f(var, x):
    return var[0] * np.exp(-var[1]*x) + var[2]


params = Parameters()
params.add('a', value=10)
params.add('b', value=10)
params.add('c', value=10)

a, b, c = 2.5, 1.3, 0.8
x = np.linspace(0, 4, 50)
y = f([a, b, c], x)
data = y + 0.15*np.random.normal(size=x.size)

# fit without analytic derivative
min1 = Minimizer(func, params, fcn_args=(x,), fcn_kws={'data': data})
out1 = min1.leastsq()
fit1 = func(out1.params, x)

# fit with analytic derivative
min2 = Minimizer(func, params, fcn_args=(x,), fcn_kws={'data': data})
out2 = min2.leastsq(Dfun=dfunc, col_deriv=1)
fit2 = func(out2.params, x)
```

Comparison of fit to exponential decay with/without analytical derivatives to model = a*exp(-b*x) + c

```
print('''
"true" parameters are: a = %.3f, b = %.3f, c = %.3f

==============================================
Statistic/Parameter|   Without    | With       |
----------------------------------------------
N Function Calls   |   %3i        |   %3i      |
```

(continues on next page)

(continued from previous page)

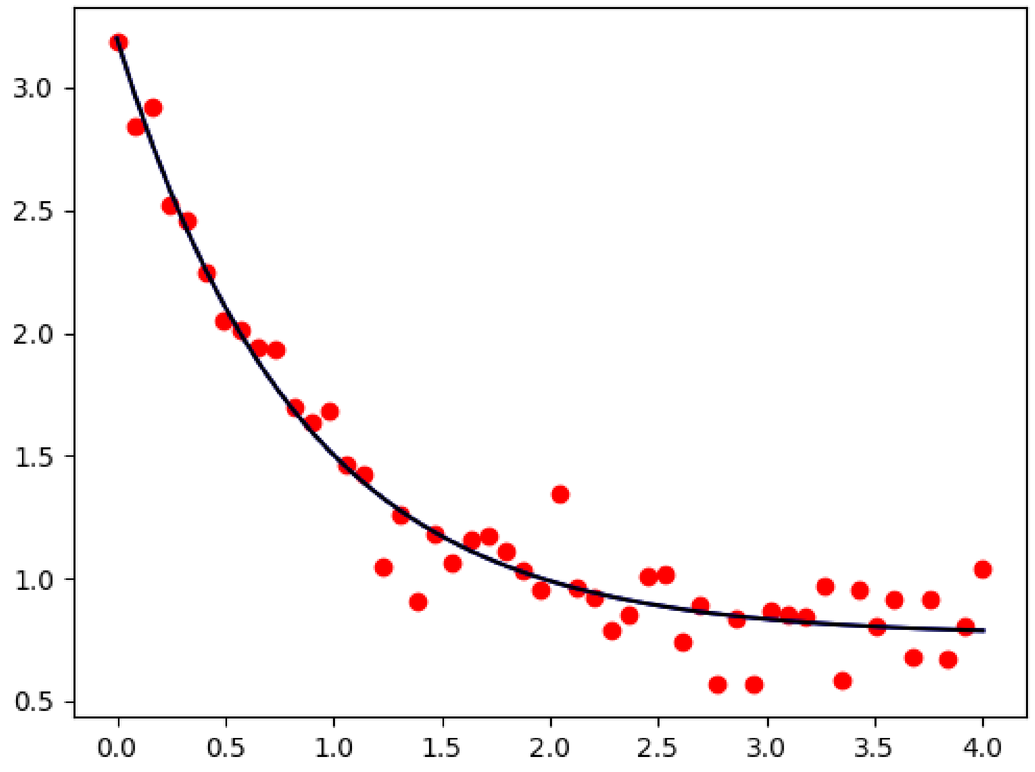
```
Chi-square          |   %.4f    |   %.4f  |
   a                |   %.4f    |   %.4f  |
   b                |   %.4f    |   %.4f  |
   c                |   %.4f    |   %.4f  |
----------------------------------------------
''' % (a, b, c,
       out1.nfev, out2.nfev,
       out1.chisqr, out2.chisqr,
       out1.params['a'], out2.params['a'],
       out1.params['b'], out2.params['b'],
       out1.params['c'], out2.params['c']))
```

Out:

```
"true" parameters are: a = 2.500, b = 1.300, c = 0.800

==============================================
Statistic/Parameter|   Without   | With      |
----------------------------------------------
N Function Calls   |    44       |    14     |
Chi-square         |   0.8974    |   0.8974  |
   a               |   2.4327    |   2.4327  |
   b               |   1.1946    |   1.1946  |
   c               |   0.7669    |   0.7669  |
----------------------------------------------
```

and the best-fit to the synthetic data (with added noise) is the same for both methods:

```
plt.plot(x, data, 'ro')
plt.plot(x, fit1, 'b')
plt.plot(x, fit2, 'k')
plt.show()
```



**Total running time of the script:** ( 0 minutes 0.267 seconds)

## 13.11 Fit using the Model interface

This notebook shows a simple example of using the `lmfit.Model` class. For more information please refer to: https://lmfit.github.io/lmfit-py/model.html#the-model-class.

```
import numpy as np
from pandas import Series

from lmfit import Model, Parameter, report_fit
```

The `Model` class is a flexible, concise curve fitter. I will illustrate fitting example data to an exponential decay.

```
def decay(t, N, tau):
    return N*np.exp(-t/tau)
```

The parameters are in no particular order. We'll need some example data. I will use `N=7` and `tau=3`, and add a little noise.

```
t = np.linspace(0, 5, num=1000)
data = decay(t, 7, 3) + np.random.randn(t.size)
```

**Simplest Usage**

```
model = Model(decay, independent_vars=['t'])
result = model.fit(data, t=t, N=10, tau=1)
```

The Model infers the parameter names by inspecting the arguments of the function, `decay`. Then I passed the independent variable, `t`, and initial guesses for each parameter. A residual function is automatically defined, and a least-squared regression is performed.

We can immediately see the best-fit values:

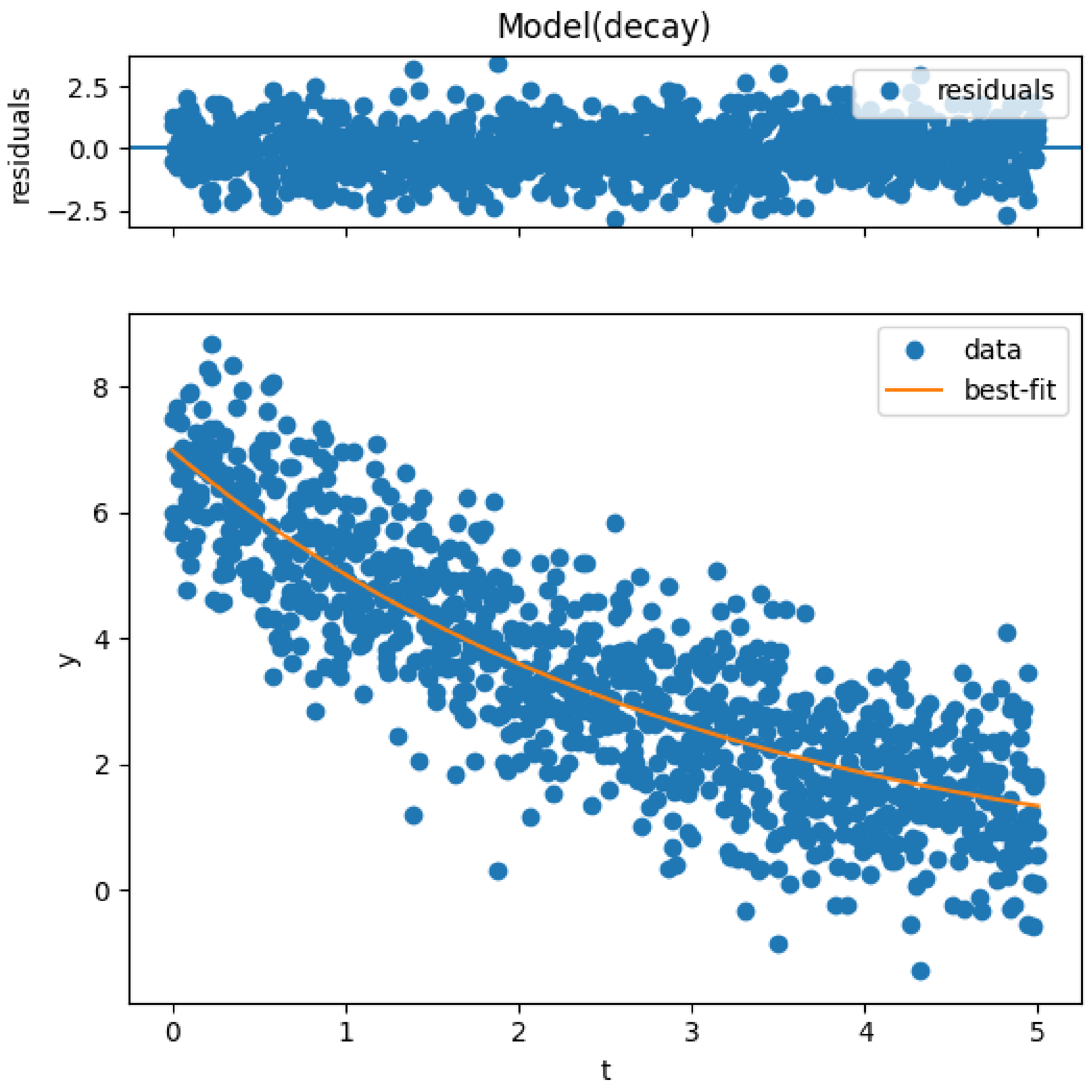
```
print(result.values)
```

Out:

```
{'N': 6.984064047700289, 'tau': 3.0113542953210715}
```

and use these best-fit parameters for plotting with the `plot` function:

```
result.plot()
```



Out:

```
(<Figure size 640x640 with 2 Axes>, GridSpec(2, 1, height_ratios=[1, 4]))
```

We can review the best-fit *Parameters* by accessing *result.params*:

```
result.params.pretty_print()
```

Out:

```
Name      Value      Min      Max    Stderr     Vary     Expr Brute_Step
N         6.984     -inf      inf   0.08845     True     None     None
tau       3.011     -inf      inf     0.066     True     None     None
```

More information about the fit is stored in the result, which is an `lmfit.MimimizerResult` object (see: https://lmfit.github.io/lmfit-py/fitting.html#lmfit.minimizer.MinimizerResult)

### Specifying Bounds and Holding Parameters Constant

Above, the `Model` class implicitly builds `Parameter` objects from keyword arguments of `fit` that match the arguments of `decay`. You can build the `Parameter` objects explicitly; the following is equivalent.

```
result = model.fit(data, t=t,
                   N=Parameter('N', value=10),
                   tau=Parameter('tau', value=1))
report_fit(result.params)
```

Out:

```
[[Variables]]
    N:    6.98406405 +/- 0.08844670 (1.27%) (init = 10)
    tau:  3.01135430 +/- 0.06600023 (2.19%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(N, tau) = -0.756
```

By building `Parameter` objects explicitly, you can specify bounds (`min`, `max`) and set parameters constant (`vary=False`).

```
result = model.fit(data, t=t,
                   N=Parameter('N', value=7, vary=False),
                   tau=Parameter('tau', value=1, min=0))
report_fit(result.params)
```

Out:

```
[[Variables]]
    N:    7 (fixed)
    tau:  3.00247383 +/- 0.04292226 (1.43%) (init = 1)
```

### Defining Parameters in Advance

Passing parameters to `fit` can become unwieldy. As an alternative, you can extract the parameters from `model` like so, set them individually, and pass them to `fit`.

```
params = model.make_params()

params['N'].value = 10
params['tau'].value = 1
params['tau'].min = 0

result = model.fit(data, params, t=t)
report_fit(result.params)
```

Out:

```
[[Variables]]
    N:    6.98406394 +/- 0.08844674 (1.27%) (init = 10)
    tau:  3.01135441 +/- 0.06599962 (2.19%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(N, tau) = -0.756
```

Keyword arguments override `params`, resetting `value` and all other properties (`min`, `max`, `vary`).

```
result = model.fit(data, params, t=t, tau=1)
report_fit(result.params)
```

Out:

```
[[Variables]]
    N:    6.98406394 +/- 0.08844674 (1.27%) (init = 10)
    tau:  3.01135441 +/- 0.06599962 (2.19%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(N, tau) = -0.756
```

The input parameters are not modified by `fit`. They can be reused, retaining the same initial value. If you want to use the result of one fit as the initial guess for the next, simply pass `params=result.params`.

#TODO/FIXME: not sure if there ever way a "helpful exception", but currently #it raises a `ValueError: The input contains nan values`.

#*A Helpful Exception*

#All this implicit magic makes it very easy for the user to neglect to set a #parameter. The `fit` function checks for this and raises a helpful exception.

```
# #result = model.fit(data, t=t, tau=1)  # N unspecified
```

#An *extra* parameter that cannot be matched to the model function will #throw a `UserWarning`, but it will not raise, leaving open the possibility #of unforeseen extensions calling for some parameters.

*Weighted Fits*

Use the `sigma` argument to perform a weighted fit. If you prefer to think of the fit in term of `weights`, `sigma=1/weights`.

```
weights = np.arange(len(data))
result = model.fit(data, params, t=t, weights=weights)
report_fit(result.params)
```

Out:

```
[[Variables]]
    N:    7.33054986 +/- 0.26968212 (3.68%) (init = 10)
    tau:  2.84538079 +/- 0.09473421 (3.33%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(N, tau) = -0.929
```

*Handling Missing Data*

By default, attempting to fit data that includes a `NaN`, which conventionally indicates a "missing" observation, raises a lengthy exception. You can choose to `omit` (i.e., skip over) missing values instead.

```
data_with_holes = data.copy()
data_with_holes[[5, 500, 700]] = np.nan  # Replace arbitrary values with NaN.

model = Model(decay, independent_vars=['t'], nan_policy='omit')
result = model.fit(data_with_holes, params, t=t)
report_fit(result.params)
```

Out:

```
[[Variables]]
    N:    6.99349787 +/- 0.08886695 (1.27%) (init = 10)
    tau:  3.00311360 +/- 0.06590577 (2.19%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(N, tau) = -0.757
```

If you don't want to ignore missing values, you can set the model to raise proactively, checking for missing values before attempting the fit.

Uncomment to see the error #model = Model(decay, independent_vars=['t'], nan_policy='raise') #result = model.fit(data_with_holes, params, t=t)

The default setting is `nan_policy='raise'`, which does check for NaNs and raises an exception when present.

Null-checking relies on `pandas.isnull` if it is available. If pandas cannot be imported, it silently falls back on `numpy.isnan`.

*Data Alignment*

Imagine a collection of time series data with different lengths. It would be convenient to define one sufficiently long array `t` and use it for each time series, regardless of length. `pandas` (https://pandas.pydata.org/pandas-docs/stable/) provides tools for aligning indexed data. And, unlike most wrappers to `scipy.leastsq`, `Model` can handle pandas objects out of the box, using its data alignment features.

Here I take just a slice of the `data` and fit it to the full `t`. It is automatically aligned to the correct section of `t` using Series' index.

```
model = Model(decay, independent_vars=['t'])
truncated_data = Series(data)[200:800]  # data points 200-800
t = Series(t)  # all 1000 points
result = model.fit(truncated_data, params, t=t)
report_fit(result.params)
```

Out:

```
[[Variables]]
    N:    7.44820825 +/- 0.24899892 (3.34%) (init = 10)
    tau:  2.80229162 +/- 0.12228516 (4.36%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(N, tau) = -0.932
```

Data with missing entries and an unequal length still aligns properly.

```
model = Model(decay, independent_vars=['t'], nan_policy='omit')
truncated_data_with_holes = Series(data_with_holes)[200:800]
result = model.fit(truncated_data_with_holes, params, t=t)
report_fit(result.params)
```

Out:

```
[[Variables]]
    N:    7.46253421 +/- 0.24975620 (3.35%) (init = 10)
    tau:  2.79210670 +/- 0.12172740 (4.36%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(N, tau) = -0.932
```

**Total running time of the script:** ( 0 minutes 0.700 seconds)

## 13.12 Outlier detection via leave-one-out

Outliers can sometimes be identified by assessing the influence of each datapoint. To assess the influence of one point, we fit the dataset while the point and compare the result with the fit of the full dataset. The code below shows how to do this with lmfit. Note that the presented method is very basic.

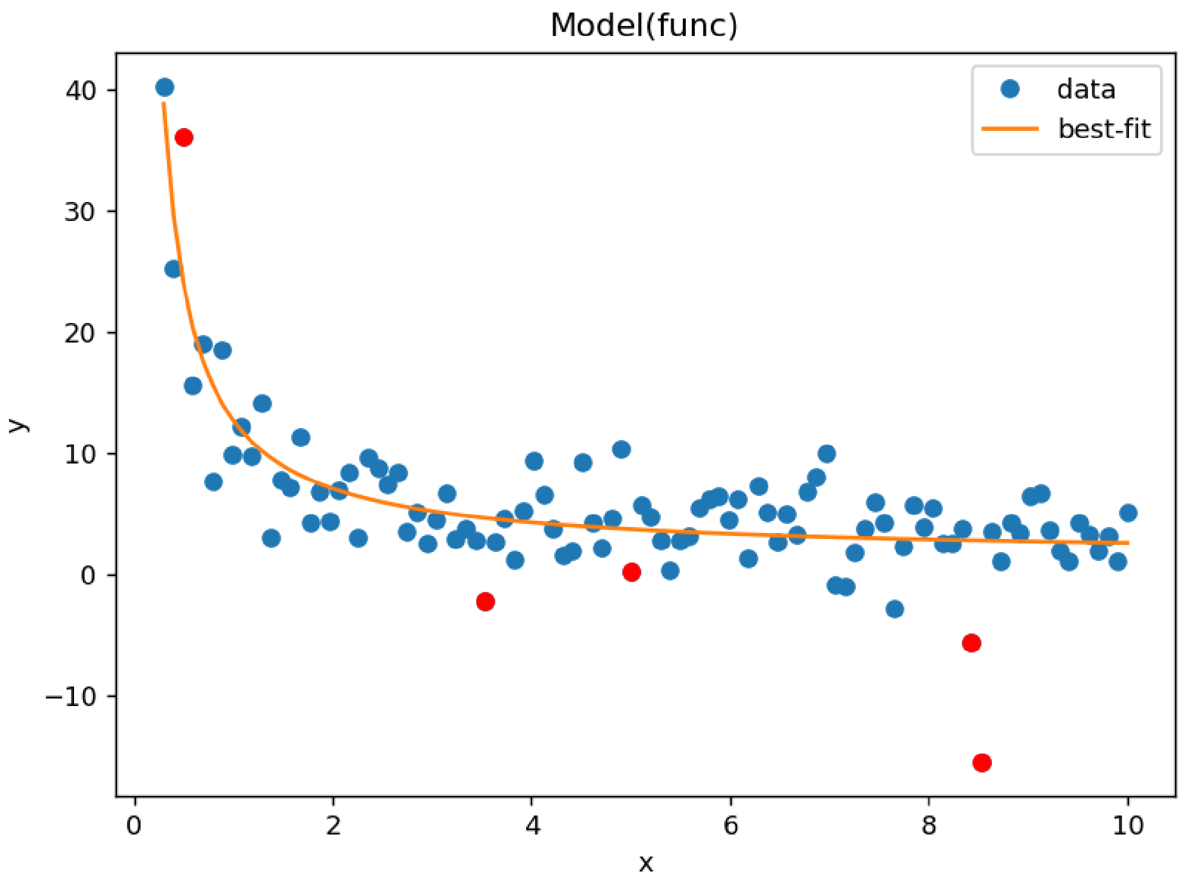
```
from collections import defaultdict

import matplotlib.pyplot as plt
import numpy as np

import lmfit

plt.rcParams['figure.dpi'] = 130
plt.rcParams['figure.autolayout'] = True
```

Generate test data and model. Apply the model to the data

```
x = np.linspace(0.3, 10, 100)
np.random.seed(1)
y = 1.0 / (0.1 * x) + 2.0 + 3 * np.random.randn(x.size)

params = lmfit.Parameters()
params.add_many(('a', 0.1), ('b', 1))


def func(x, a, b):
    return 1.0 / (a * x) + b


# Make 5 points outliers
idx = np.random.randint(0, x.size, 5)
y[idx] += 10 * np.random.randn(idx.size)

# Fit the data
model = lmfit.Model(func, independent_vars=['x'])
fit_result = model.fit(y, x=x, a=0.1, b=2)
```

and gives the plot and fitting results below:

```
fit_result.plot_fit()
plt.plot(x[idx], y[idx], 'o', color='r', label='outliers')
plt.show()
print(fit_result.fit_report())
```



Out:

```
[[Model]]
    Model(func)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 12
    # data points      = 100
    # variables        = 2
    chi-square         = 1338.34458
    reduced chi-square = 13.6565773
    Akaike info crit   = 263.401856
    Bayesian info crit = 268.612196
[[Variables]]
    a:  0.08937623 +/- 0.00590168 (6.60%) (init = 0.1)
    b:  1.51298992 +/- 0.46229147 (30.55%) (init = 2)
[[Correlations]] (unreported correlations are < 0.100)
    C(a, b) =  0.601
```

Fit the dataset while omitting one data point

```
best_vals = defaultdict(lambda: np.zeros(x.size))
stderrs = defaultdict(lambda: np.zeros(x.size))
chi_sq = np.zeros_like(x)
for i in range(x.size):
    idx2 = np.arange(0, x.size)
```

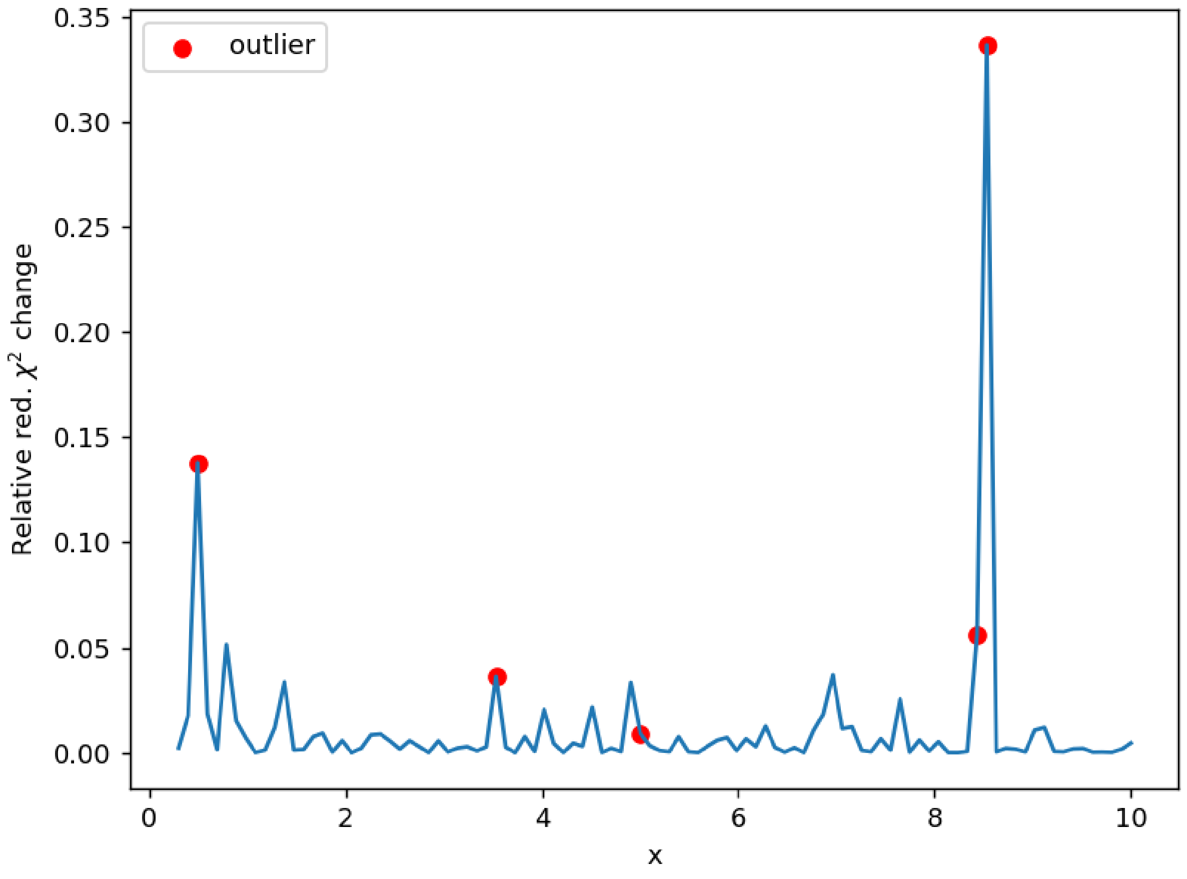
(continues on next page)

(continued from previous page)

```
    idx2 = np.delete(idx2, i)
    tmp_x = x[idx2]
    tmp = model.fit(y[idx2],
                    x=tmp_x,
                    a=fit_result.params['a'],
                    b=fit_result.params['b'])
    chi_sq[i] = tmp.chisqr
    for p in tmp.params:
        tpar = tmp.params[p]
        best_vals[p][i] = tpar.value
        stderrs[p][i] = (tpar.stderr / fit_result.params[p].stderr)
```

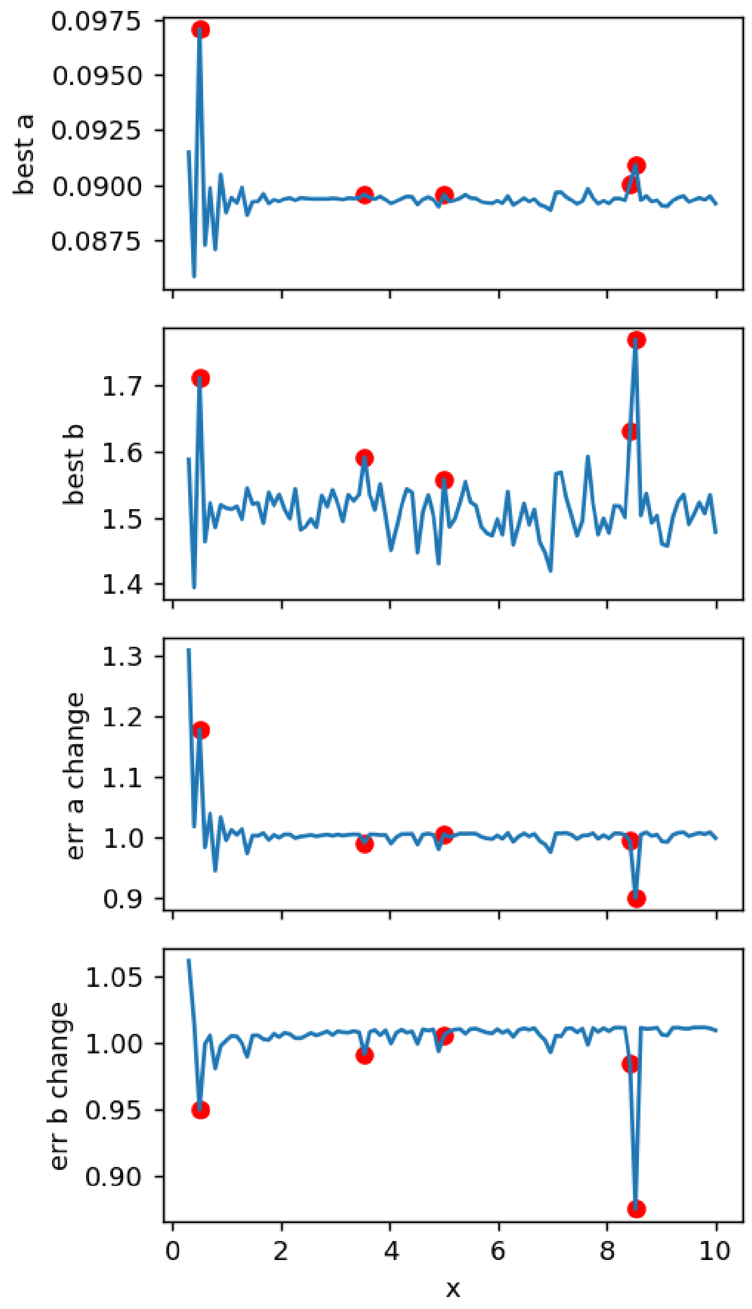
Plot the influence on the red. chisqr of each point

```
fig, ax = plt.subplots()
ax.plot(x, (fit_result.chisqr - chi_sq) / chi_sq)
ax.scatter(x[idx],
           fit_result.chisqr / chi_sq[idx] - 1,
           color='r',
           label='outlier')
ax.set_ylabel(r'Relative red. $\chi^2$ change')
ax.set_xlabel('x')
ax.legend()
```



Plot the influence on the parameter value and error of each point

```
fig, axs = plt.subplots(4, figsize=(4, 7), sharex='col')
axs[0].plot(x, best_vals['a'])
axs[0].scatter(x[idx], best_vals['a'][idx], color='r', label='outlier')
axs[0].set_ylabel('best a')

axs[1].plot(x, best_vals['b'])
axs[1].scatter(x[idx], best_vals['b'][idx], color='r', label='outlier')
axs[1].set_ylabel('best b')

axs[2].plot(x, stderrs['a'])
axs[2].scatter(x[idx], stderrs['a'][idx], color='r', label='outlier')
axs[2].set_ylabel('err a change')

axs[3].plot(x, stderrs['b'])
axs[3].scatter(x[idx], stderrs['b'][idx], color='r', label='outlier')
axs[3].set_ylabel('err b change')

axs[3].set_xlabel('x')
```



**Total running time of the script:** ( 0 minutes 2.777 seconds)

## 13.13 Emcee and the Model Interface

```
import corner
import matplotlib.pyplot as plt
import numpy as np

import lmfit
```

Set up a double-exponential function and create a Model

```
def double_exp(x, a1, t1, a2, t2):
    return a1*np.exp(-x/t1) + a2*np.exp(-(x-0.1) / t2)


model = lmfit.Model(double_exp)
```

Generate some fake data from the model with added noise

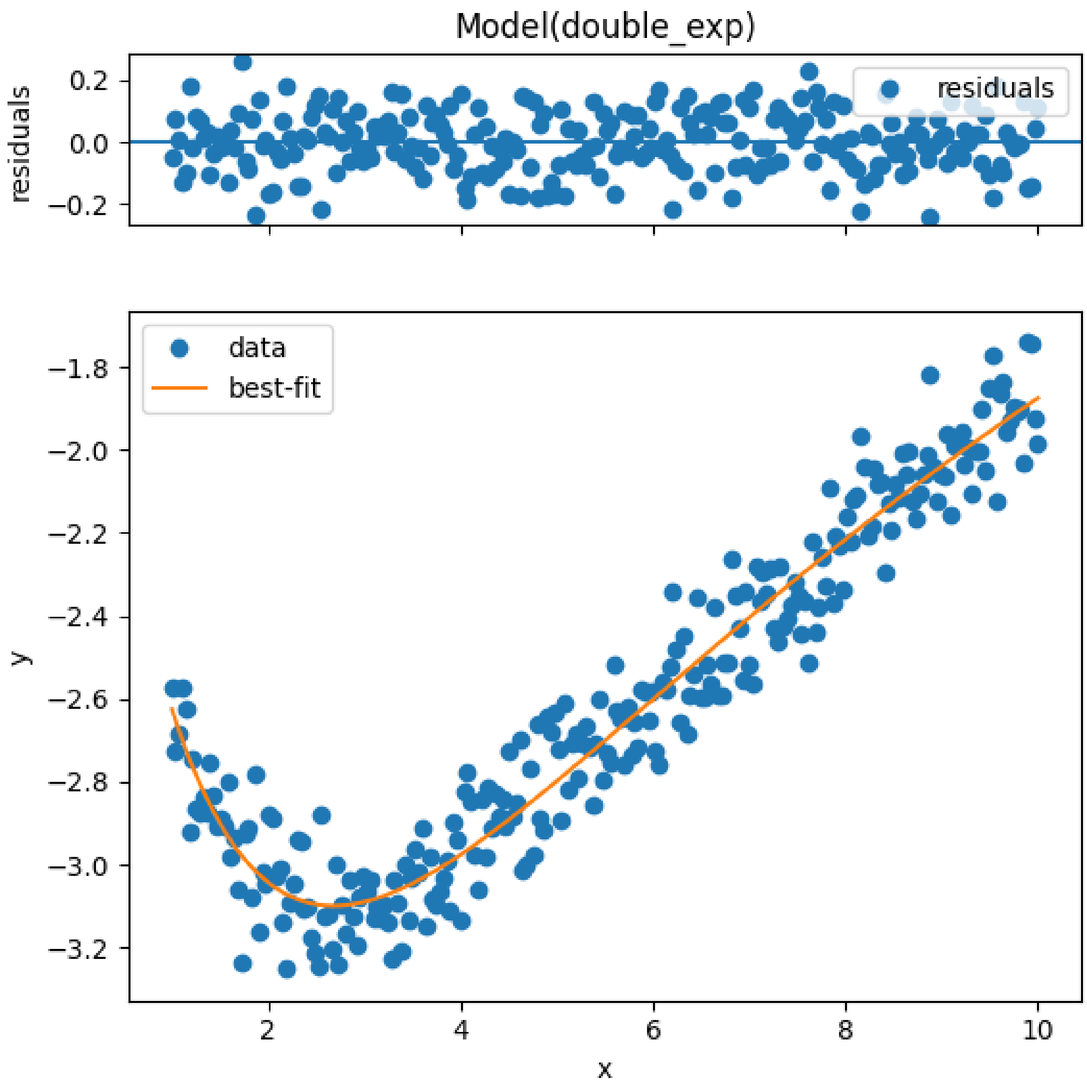
```
truths = (3.0, 2.0, -5.0, 10.0)
x = np.linspace(1, 10, 250)
np.random.seed(0)
y = double_exp(x, *truths)+0.1*np.random.randn(x.size)
```

Create model parameters and give them initial values

```
p = model.make_params(a1=4, t1=3, a2=4, t2=3)
```

Fit the model using a traditional minimizer, and show the output:

```
result = model.fit(data=y, params=p, x=x, method='Nelder', nan_policy='omit')

lmfit.report_fit(result)
result.plot()
```



Out:

```
[[Fit Statistics]]
    # fitting method   = Nelder-Mead
    # function evals   = 609
    # data points      = 250
    # variables        = 4
    chi-square         = 2.33333982
    reduced chi-square = 0.00948512
    Akaike info crit   = -1160.54007
    Bayesian info crit = -1146.45423
[[Variables]]
    a1:  2.98623689 +/- 0.15010519 (5.03%) (init = 4)
    t1:  1.30993186 +/- 0.13449656 (10.27%) (init = 3)
    a2: -4.33525597 +/- 0.11765824 (2.71%) (init = 4)
```

(continues on next page)

(continued from previous page)

```
    t2:  11.8240752 +/- 0.47172610 (3.99%) (init = 3)
[[Correlations]] (unreported correlations are < 0.100)
    C(a2, t2) =  0.988
    C(t1, a2) = -0.928
    C(t1, t2) = -0.885
    C(a1, t1) = -0.609
    C(a1, a2) =  0.297
    C(a1, t2) =  0.232

(<Figure size 640x640 with 2 Axes>, GridSpec(2, 1, height_ratios=[1, 4]))
```

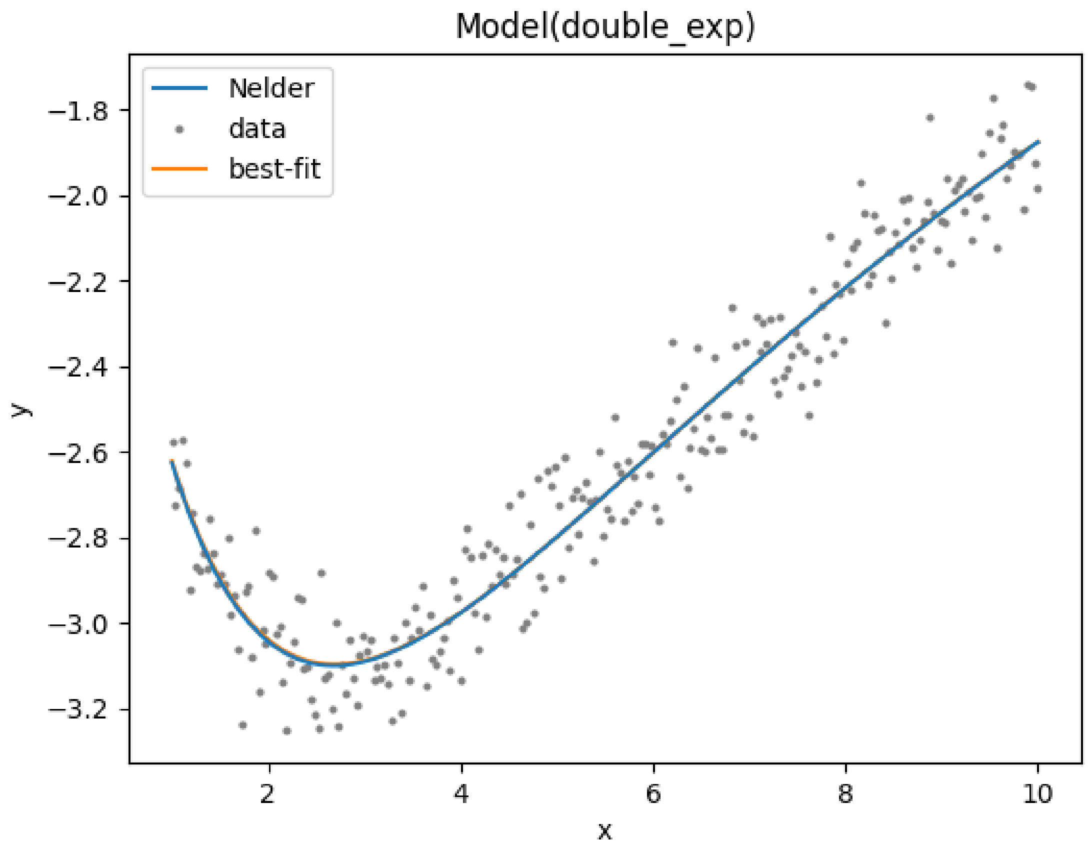
Calculate parameter covariance using emcee:

- start the walkers out at the best-fit values
- set is_weighted to False to estimate the noise weights
- set some sensible priors on the uncertainty to keep the MCMC in check

```
emcee_kws = dict(steps=1000, burn=300, thin=20, is_weighted=False,
                 progress=False)
emcee_params = result.params.copy()
emcee_params.add('__lnsigma', value=np.log(0.1), min=np.log(0.001), max=np.log(2.0))
```

run the MCMC algorithm and show the results:

```
result_emcee = model.fit(data=y, x=x, params=emcee_params, method='emcee',
                         nan_policy='omit', fit_kws=emcee_kws)

lmfit.report_fit(result_emcee)

ax = plt.plot(x, model.eval(params=result.params, x=x), label='Nelder', zorder=100)
result_emcee.plot_fit(ax=ax, data_kws=dict(color='gray', markersize=2))
plt.show()
```



Out:

```
The chain is shorter than 50 times the integrated autocorrelation time for 5␣
↪parameter(s). Use this estimate with caution and run a longer chain!
N/50 = 20;
tau: [42.15512406 49.13844172 47.97313337 48.69759859 39.90675732]
[[Fit Statistics]]
    # fitting method   = emcee
    # function evals   = 100000
    # data points      = 250
    # variables        = 5
    chi-square         = 245.611753
    reduced chi-square = 1.00249695
    Akaike info crit   = 5.57278242
    Bayesian info crit = 23.1800870
[[Variables]]
    a1:         2.99014520 +/- 0.15080185 (5.04%) (init = 2.986237)
    t1:         1.32465508 +/- 0.14246577 (10.75%) (init = 1.309932)
    a2:        -4.34626582 +/- 0.12518881 (2.88%) (init = -4.335256)
    t2:         11.7839459 +/- 0.48230303 (4.09%) (init = 11.82408)
    __lnsigma: -2.32797933 +/- 0.04489245 (1.93%) (init = -2.302585)
[[Correlations]] (unreported correlations are < 0.100)
    C(a2, t2) =  0.981
    C(t1, a2) = -0.939
    C(t1, t2) = -0.898
    C(a1, t1) = -0.540
```

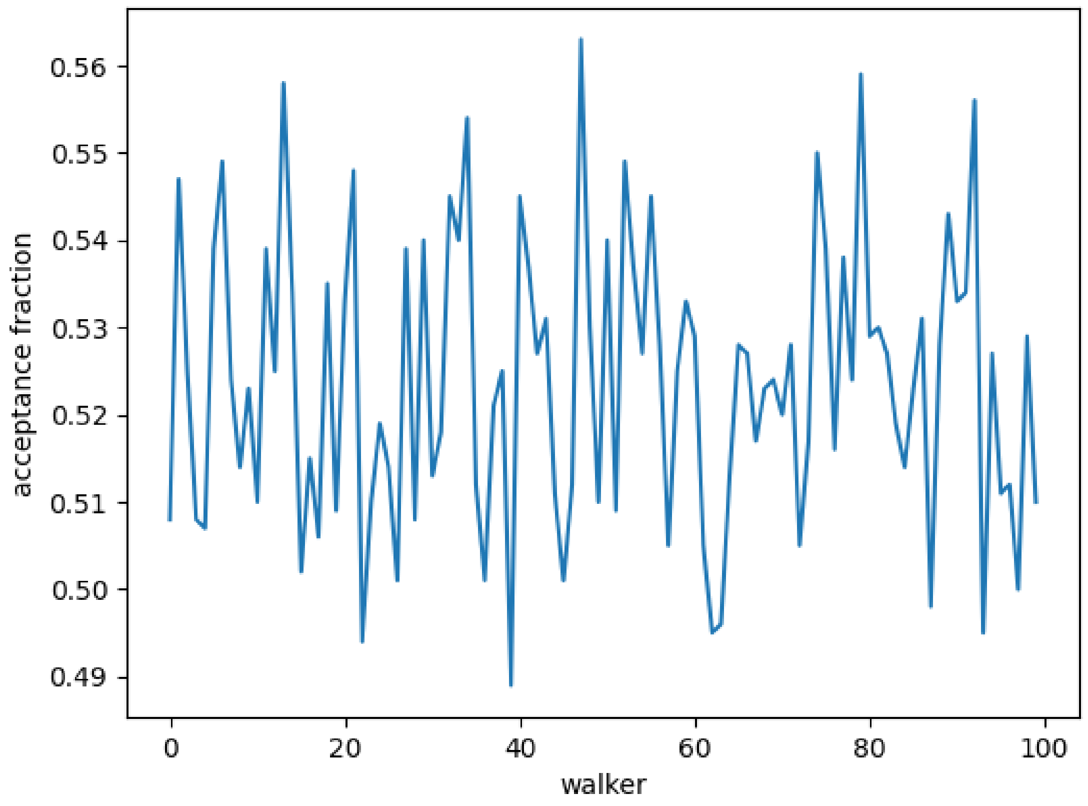
(continues on next page)

(continued from previous page)

```
    C(a1, a2) =  0.257
    C(a1, t2) =  0.225
```

check the acceptance fraction to see whether emcee performed well

```
plt.plot(result_emcee.acceptance_fraction)
plt.xlabel('walker')
plt.ylabel('acceptance fraction')
plt.show()
```

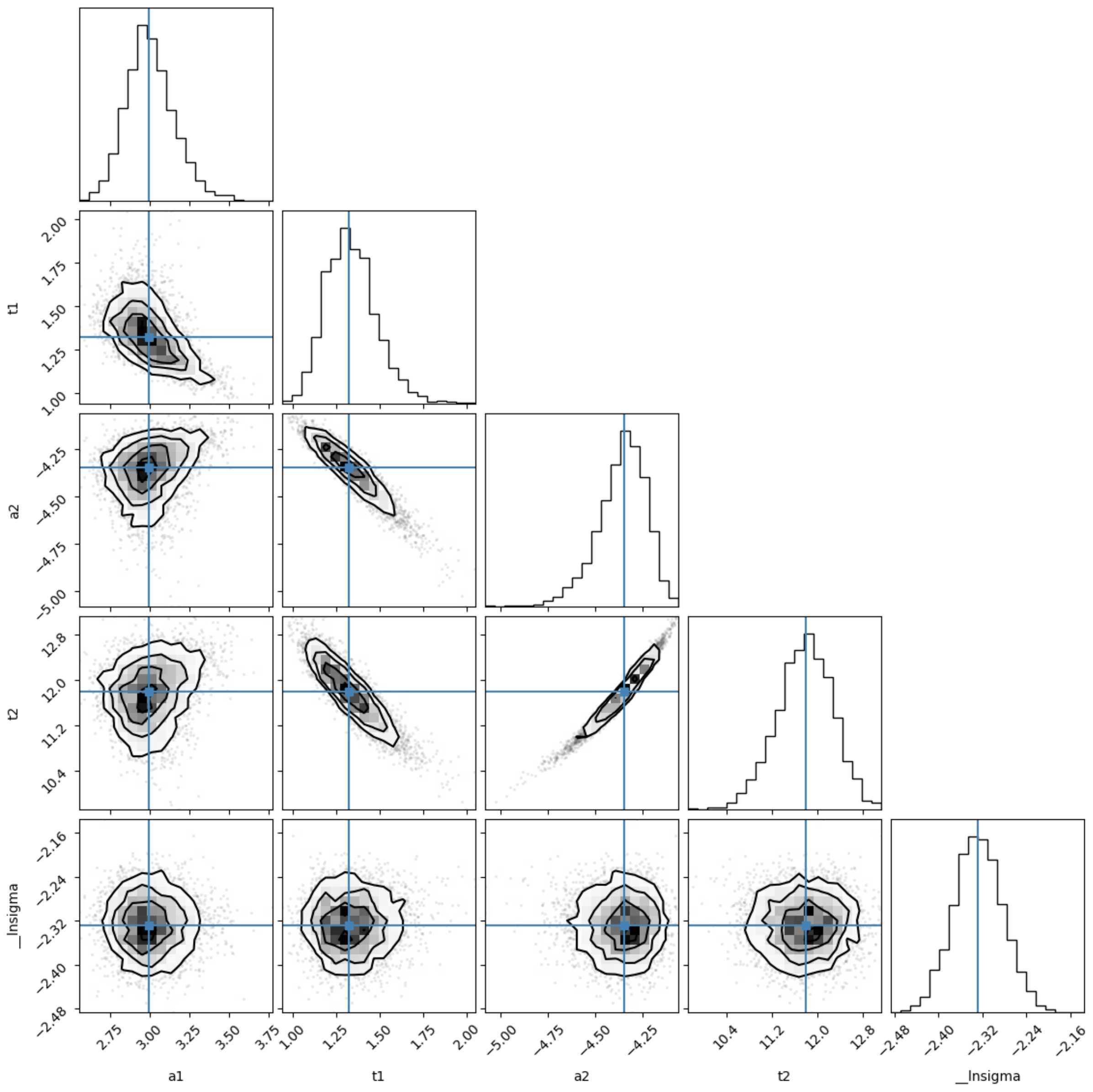


try to compute the autocorrelation time

```
if hasattr(result_emcee, "acor"):
    print("Autocorrelation time for the parameters:")
    print("----------------------------------------")
    for i, p in enumerate(result.params):
        print(p, result.acor[i])
```

Plot the parameter covariances returned by emcee using corner

```
emcee_corner = corner.corner(result_emcee.flatchain, labels=result_emcee.var_names,
                             truths=list(result_emcee.params.valuesdict().values()))
```



```
print("\nmedian of posterior probability distribution")
print('--------------------------------------------')
lmfit.report_fit(result_emcee.params)

# find the maximum likelihood solution
highest_prob = np.argmax(result_emcee.lnprob)
hp_loc = np.unravel_index(highest_prob, result_emcee.lnprob.shape)
mle_soln = result_emcee.chain[hp_loc]
print("\nMaximum likelihood Estimation")
print('-----------------------------')
for ix, param in enumerate(emcee_params):
    print(param + ': ' + str(mle_soln[ix]))

quantiles = np.percentile(result_emcee.flatchain['t1'], [2.28, 15.9, 50, 84.2, 97.7])
```

(continues on next page)

(continued from previous page)

```
print("\n\n1 sigma spread", 0.5 * (quantiles[3] - quantiles[1]))
print("2 sigma spread", 0.5 * (quantiles[4] - quantiles[0]))
```

Out:

```
median of posterior probability distribution
--------------------------------------------
[[Variables]]
    a1:         2.99014520 +/- 0.15080185 (5.04%) (init = 2.986237)
    t1:         1.32465508 +/- 0.14246577 (10.75%) (init = 1.309932)
    a2:        -4.34626582 +/- 0.12518881 (2.88%) (init = -4.335256)
    t2:         11.7839459 +/- 0.48230303 (4.09%) (init = 11.82408)
    __lnsigma: -2.32797933 +/- 0.04489245 (1.93%) (init = -2.302585)
[[Correlations]] (unreported correlations are < 0.100)
    C(a2, t2) =  0.981
    C(t1, a2) = -0.939
    C(t1, t2) = -0.898
    C(a1, t1) = -0.540
    C(a1, a2) =  0.257
    C(a1, t2) =  0.225

Maximum likelihood Estimation
-----------------------------
a1: 2.9704980735108304
t1: 1.3209758260802968
a2: -4.338098241979906
t2: 11.838963811736026
__lnsigma: -2.3403986558500596


1 sigma spread 0.14233438845386792
2 sigma spread 0.2943016210688928
```

**Total running time of the script:** ( 0 minutes 26.500 seconds)

## 13.14 Calculate Confidence Intervals

```
import matplotlib.pyplot as plt
from numpy import argsort, exp, linspace, pi, random, sign, sin, unique
from scipy.interpolate import interp1d

from lmfit import (Minimizer, Parameters, conf_interval, conf_interval2d,
                   report_ci, report_fit)
```

Define the residual function, specify “true” parameter values, and generate a synthetic data set with some noise:

```
def residual(pars, x, data=None):
    argu = (x*pars['decay'])**2
    shift = pars['shift']
    if abs(shift) > pi/2:
        shift = shift - sign(shift)*pi
    model = pars['amp']*sin(shift + x/pars['period']) * exp(-argu)
    if data is None:
        return model
```

(continues on next page)

(continued from previous page)

```
    return model - data


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

x = linspace(0.0, 250.0, 2500)
noise = random.normal(scale=0.7215, size=x.size)
data = residual(p_true, x) + noise
```

Create fitting parameters and set initial values:

```
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)
```

Set-up the minimizer and perform the fit using leastsq algorithm, and show the report:

```
mini = Minimizer(residual, fit_params, fcn_args=(x,), fcn_kws={'data': data})
out = mini.leastsq()

fit = residual(out.params, x)
report_fit(out)
```

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 90
    # data points      = 2500
    # variables        = 4
    chi-square         = 1212.08286
    reduced chi-square = 0.48561012
    Akaike info crit   = -1801.87620
    Bayesian info crit = -1778.58002
[[Variables]]
    amp:     13.9952862 +/- 0.04810194 (0.34%) (init = 13)
    period:  5.32947061 +/- 0.00267669 (0.05%) (init = 2)
    shift:   0.12201493 +/- 0.00472513 (3.87%) (init = 0)
    decay:   0.01002913 +/- 3.9913e-05 (0.40%) (init = 0.02)
[[Correlations]] (unreported correlations are < 0.100)
    C(period, shift) =  0.800
    C(amp, decay)    =  0.576
```

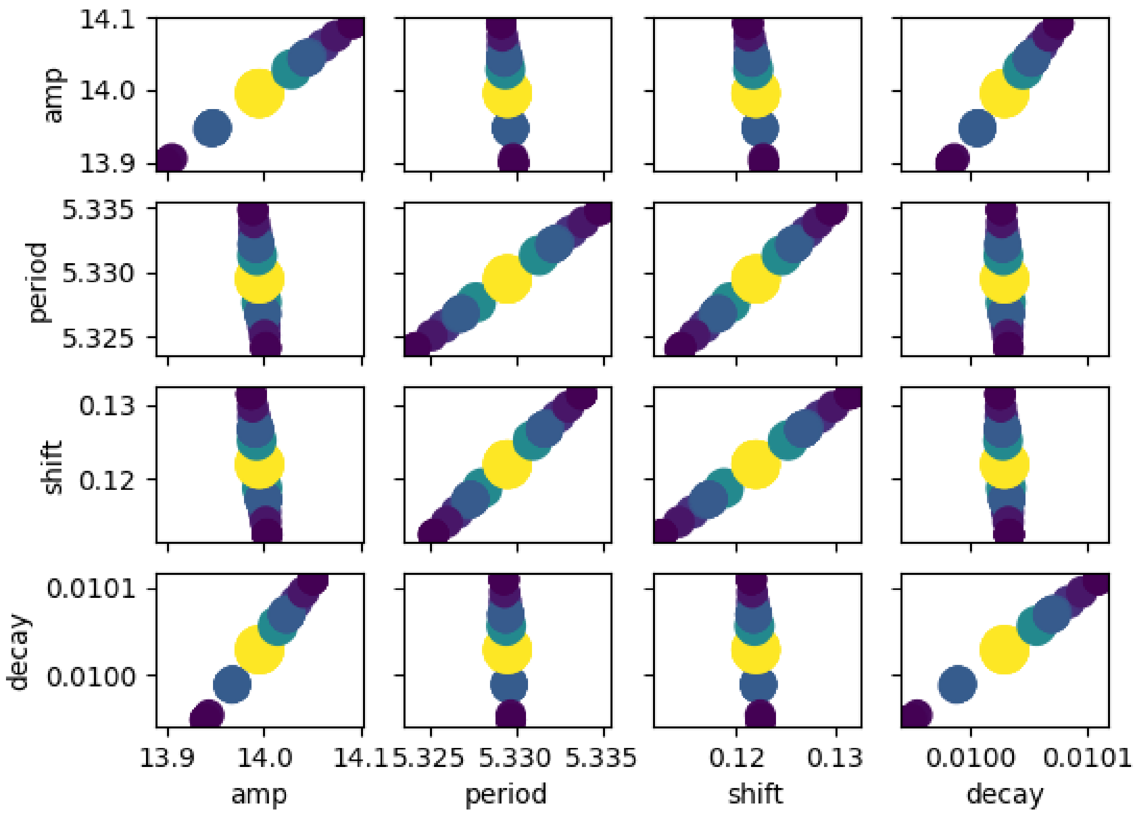
Calculate the confidence intervals for parameters and display the results:

```
ci, tr = conf_interval(mini, out, trace=True)

report_ci(ci)

names = out.params.keys()
```

(continues on next page)

(continued from previous page)

```
i = 0
gs = plt.GridSpec(4, 4)
sx = {}
sy = {}
for fixed in names:
    j = 0
    for free in names:
        if j in sx and i in sy:
            ax = plt.subplot(gs[i, j], sharex=sx[j], sharey=sy[i])
        elif i in sy:
            ax = plt.subplot(gs[i, j], sharey=sy[i])
            sx[j] = ax
        elif j in sx:
            ax = plt.subplot(gs[i, j], sharex=sx[j])
            sy[i] = ax
        else:
            ax = plt.subplot(gs[i, j])
            sy[i] = ax
            sx[j] = ax
        if i < 3:
            plt.setp(ax.get_xticklabels(), visible=False)
        else:
            ax.set_xlabel(free)

        if j > 0:
            plt.setp(ax.get_yticklabels(), visible=False)
        else:
            ax.set_ylabel(fixed)

        res = tr[fixed]
        prob = res['prob']
        f = prob < 0.96

        x, y = res[free], res[fixed]
        ax.scatter(x[f], y[f], c=1-prob[f], s=200*(1-prob[f]+0.5))
        ax.autoscale(1, 1)
        j += 1
    i += 1
```



Out:

```
            99.73%    95.45%    68.27%    _BEST_    68.27%    95.45%    99.73%
amp    :  -0.14396  -0.09606  -0.04806  13.99529  +0.04794  +0.09637  +0.14466
period:  -0.00807  -0.00532  -0.00267   5.32947  +0.00267  +0.00534  +0.00811
shift  :  -0.01420  -0.00947  -0.00473   0.12201  +0.00474  +0.00948  +0.01423
decay  :  -0.00012  -0.00008  -0.00004   0.01003  +0.00004  +0.00008  +0.00012
```

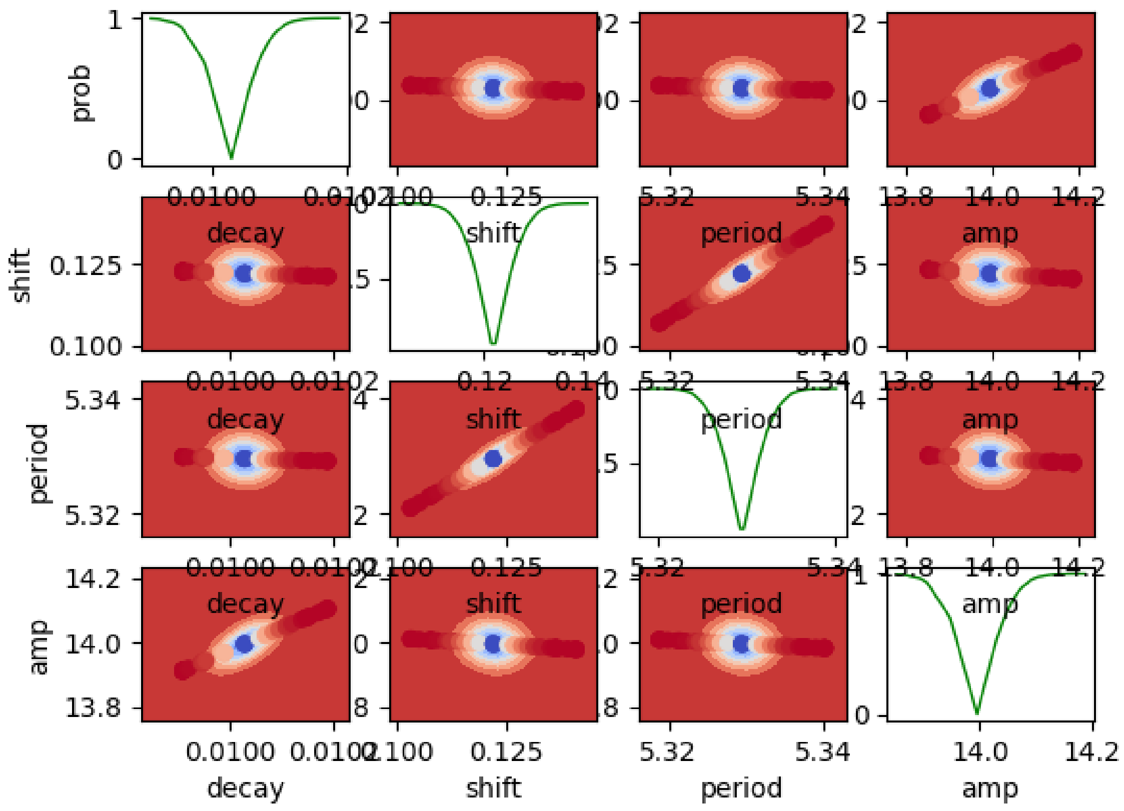
It is also possible to calculate the confidence regions for two fixed parameters using the function conf_interval2d:

```
names = list(out.params.keys())

plt.figure()
cm = plt.cm.coolwarm
for i in range(4):
    for j in range(4):
        plt.subplot(4, 4, 16-j*4-i)
        if i != j:
            x, y, m = conf_interval2d(mini, out, names[i], names[j], 20, 20)
            plt.contourf(x, y, m, linspace(0, 1, 10), cmap=cm)
            plt.xlabel(names[i])
            plt.ylabel(names[j])

            x = tr[names[i]][names[i]]
            y = tr[names[i]][names[j]]
```

(continues on next page)

(continued from previous page)

```
            pr = tr[names[i]]['prob']
            s = argsort(x)
            plt.scatter(x[s], y[s], c=pr[s], s=30, lw=1, cmap=cm)
        else:
            x = tr[names[i]][names[i]]
            y = tr[names[i]]['prob']

            t, s = unique(x, True)
            f = interp1d(t, y[s], 'slinear')
            xn = linspace(x.min(), x.max(), 50)
            plt.plot(xn, f(xn), 'g', lw=1)
            plt.xlabel(names[i])
            plt.ylabel('prob')

plt.show()
```



**Total running time of the script:** ( 0 minutes 51.867 seconds)

## 13.15 Complex Resonator Model

This notebook shows how to fit the parameters of a complex resonator, using *lmfit.Model* and defining a custom *Model* class.

Following Khalil et al. (https://arxiv.org/abs/1108.3117), we can model the forward transmission of a microwave resonator with total quality factor $Q$, coupling quality factor $Q_e$, and resonant frequency $f_0$ using:

$$S_{21}(f) = 1 - \frac{QQ_e^{-1}}{1 + 2jQ(f - f_0)/f_0}$$

$S_{21}$ is thus a complex function of a real frequency.

By allowing $Q_e$ to be complex, this model can take into account mismatches in the input and output transmission impedances.

```
import matplotlib.pyplot as plt
import numpy as np

import lmfit
```

Since `scipy.optimize` and `lmfit` require real parameters, we represent $Q_e$ as `Q_e_real + 1j*Q_e_imag`.

```
def linear_resonator(f, f_0, Q, Q_e_real, Q_e_imag):
    Q_e = Q_e_real + 1j*Q_e_imag
    return 1 - (Q * Q_e**-1 / (1 + 2j * Q * (f - f_0) / f_0))
```

The standard practice of defining an `lmfit` model is as follows:

```
class ResonatorModel(lmfit.model.Model):
    __doc__ = "resonator model" + lmfit.models.COMMON_DOC

    def __init__(self, *args, **kwargs):
        # pass in the defining equation so the user doesn't have to later.
        super().__init__(linear_resonator, *args, **kwargs)

        self.set_param_hint('Q', min=0)  # Enforce Q is positive

    def guess(self, data, f=None, **kwargs):
        verbose = kwargs.pop('verbose', None)
        if f is None:
            return
        argmin_s21 = np.abs(data).argmin()
        fmin = f.min()
        fmax = f.max()
        f_0_guess = f[argmin_s21]  # guess that the resonance is the lowest point
        Q_min = 0.1 * (f_0_guess/(fmax-fmin))  # assume the user isn't trying to fit␣
↪just a small part of a resonance curve.
        delta_f = np.diff(f)  # assume f is sorted
        min_delta_f = delta_f[delta_f > 0].min()
        Q_max = f_0_guess/min_delta_f  # assume data actually samples the resonance␣
↪reasonably
        Q_guess = np.sqrt(Q_min*Q_max)  # geometric mean, why not?
        Q_e_real_guess = Q_guess/(1-np.abs(data[argmin_s21]))
        if verbose:
            print("fmin=", fmin, "fmax=", fmax, "f_0_guess=", f_0_guess)
            print("Qmin=", Q_min, "Q_max=", Q_max, "Q_guess=", Q_guess, "Q_e_real_
↪guess=", Q_e_real_guess)
```

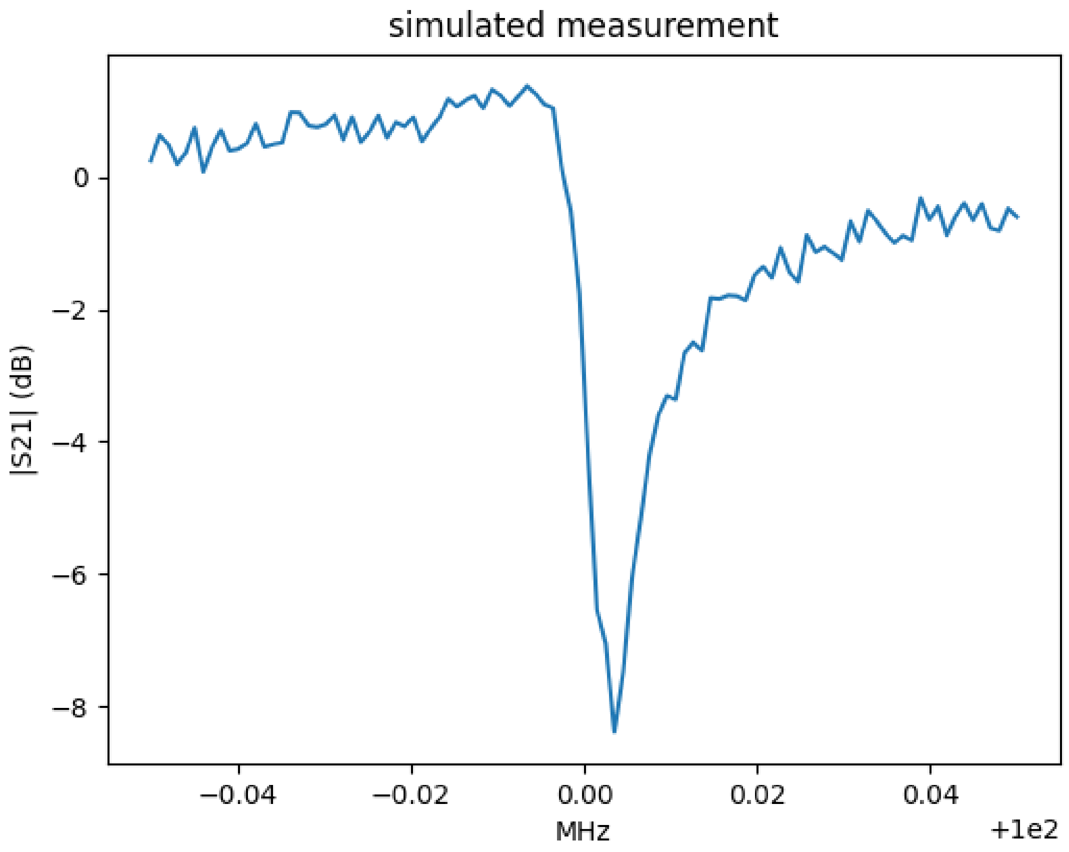
(continues on next page)

(continued from previous page)

```
        params = self.make_params(Q=Q_guess, Q_e_real=Q_e_real_guess, Q_e_imag=0, f_
↪0=f_0_guess)
        params['%sQ' % self.prefix].set(min=Q_min, max=Q_max)
        params['%sf_0' % self.prefix].set(min=fmin, max=fmax)
        return lmfit.models.update_param_vals(params, self.prefix, **kwargs)
```

Now let's use the model to generate some fake data:

```
resonator = ResonatorModel()
true_params = resonator.make_params(f_0=100, Q=10000, Q_e_real=9000, Q_e_imag=-9000)

f = np.linspace(99.95, 100.05, 100)
true_s21 = resonator.eval(params=true_params, f=f)
noise_scale = 0.02
np.random.seed(123)
measured_s21 = true_s21 + noise_scale*(np.random.randn(100) + 1j*np.random.randn(100))

plt.figure()
plt.plot(f, 20*np.log10(np.abs(measured_s21)))
plt.ylabel('|S21| (dB)')
plt.xlabel('MHz')
plt.title('simulated measurement')
```



Try out the guess method we added:

```
guess = resonator.guess(measured_s21, f=f, verbose=True)
```

Out:

```
fmin= 99.95 fmax= 100.05 f_0_guess= 100.00353535353536
Qmin= 100.00353535354105 Q_max= 99003.50000055433 Q_guess= 3146.537781821432 Q_e_real_
↪guess= 5082.2474265369565
```

And now fit the data using the guess as a starting point:

```
result = resonator.fit(measured_s21, params=guess, f=f, verbose=True)

print(result.fit_report() + '\n')
result.params.pretty_print()
```

Out:

```
[[Model]]
    Model(linear_resonator)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 36
    # data points      = 200
    # variables        = 4
    chi-square         = 0.08533642
    reduced chi-square = 4.3539e-04
    Akaike info crit   = -1543.89425
    Bayesian info crit = -1530.70099
[[Variables]]
    f_0:        100.000096 +/- 7.0378e-05 (0.00%) (init = 100.0035)
    Q:          10059.4926 +/- 142.294761 (1.41%) (init = 3146.538)
    Q_e_real:   9180.61935 +/- 133.776862 (1.46%) (init = 5082.247)
    Q_e_imag: -9137.03303 +/- 133.770211 (1.46%) (init = 0)
[[Correlations]] (unreported correlations are < 0.100)
    C(Q, Q_e_real)    =  0.518
    C(f_0, Q_e_imag) =  0.517
    C(f_0, Q_e_real) =  0.515
    C(Q, Q_e_imag)    = -0.515

Name          Value      Min       Max   Stderr     Vary     Expr Brute_Step
Q          1.006e+04       100   9.9e+04    142.3     True     None      None
Q_e_imag      -9137      -inf       inf    133.8     True     None      None
Q_e_real       9181      -inf       inf    133.8     True     None      None
f_0             100     99.95       100 7.038e-05     True     None      None
```

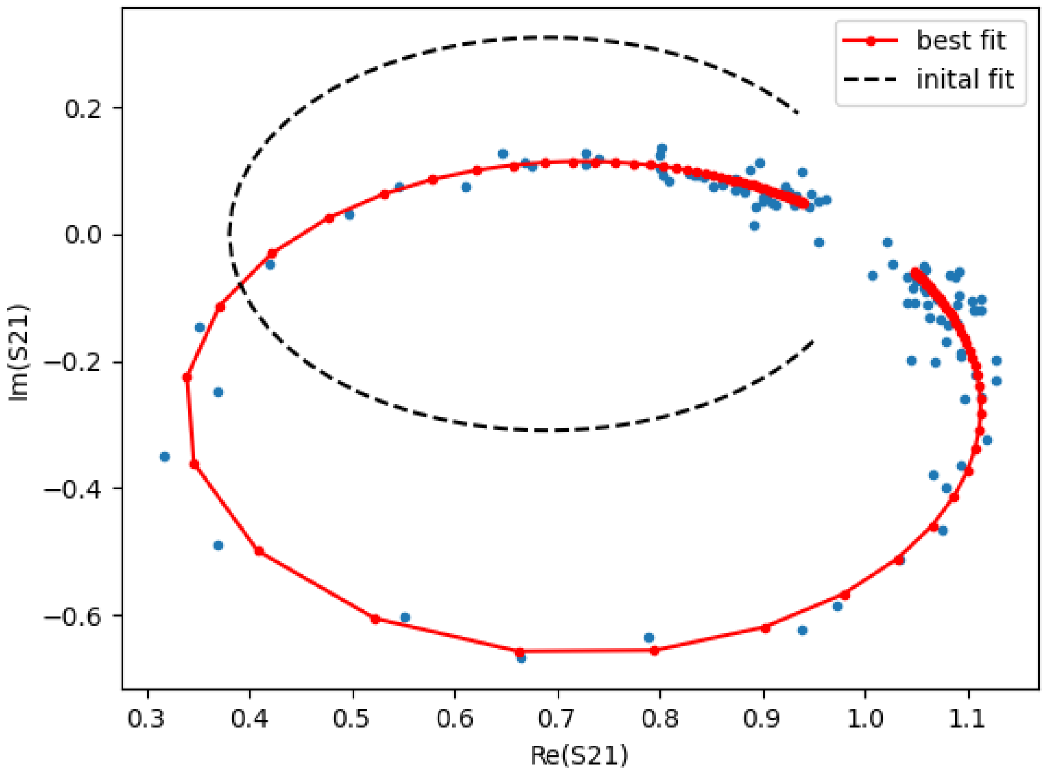
Now we'll make some plots of the data and fit. Define a convenience function for plotting complex quantities:

```
def plot_ri(data, *args, **kwargs):
    plt.plot(data.real, data.imag, *args, **kwargs)


fit_s21 = resonator.eval(params=result.params, f=f)
guess_s21 = resonator.eval(params=guess, f=f)

plt.figure()
plot_ri(measured_s21, '.')
plot_ri(fit_s21, 'r.-', label='best fit')
```

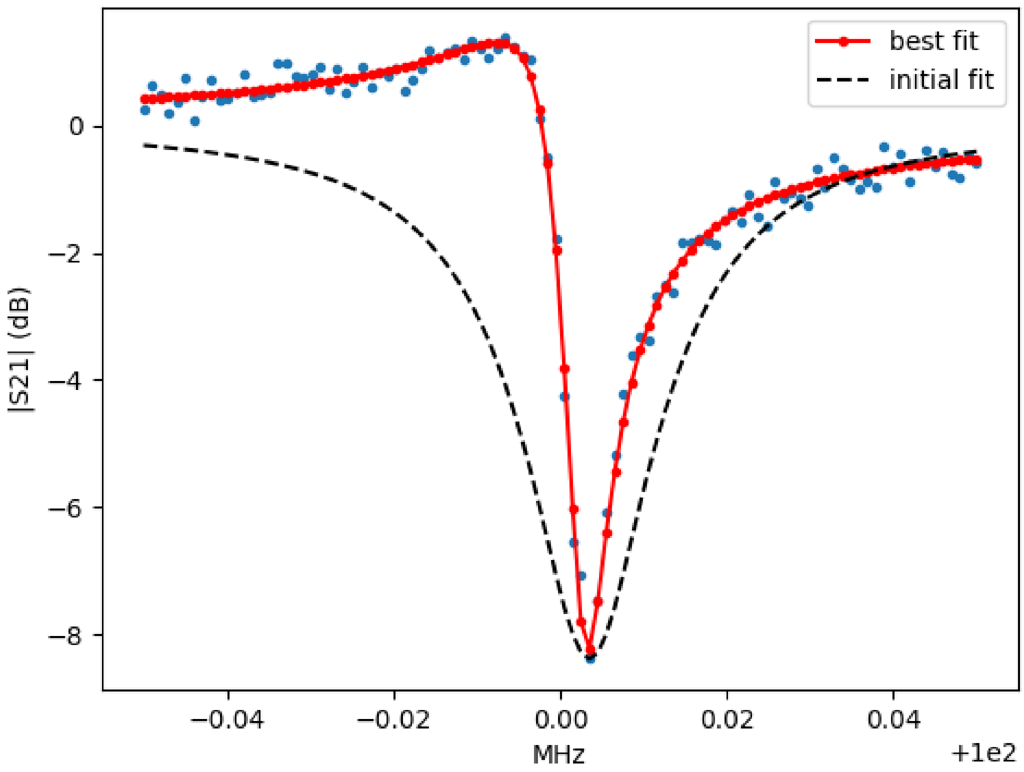
(continues on next page)

(continued from previous page)

```
plot_ri(guess_s21, 'k--', label='inital fit')
plt.legend(loc='best')
plt.xlabel('Re(S21)')
plt.ylabel('Im(S21)')

plt.figure()
plt.plot(f, 20*np.log10(np.abs(measured_s21)), '.')
plt.plot(f, 20*np.log10(np.abs(fit_s21)), 'r.-', label='best fit')
plt.plot(f, 20*np.log10(np.abs(guess_s21)), 'k--', label='initial fit')
plt.legend(loc='best')
plt.ylabel('|S21| (dB)')
plt.xlabel('MHz')
```



•



•

**Total running time of the script:** ( 0 minutes 0.781 seconds)

## 13.16 Model Selection using lmfit and emcee

FIXME: this is a useful examples; however, it doesn't run correctly anymore as the PTSampler was removed in emcee v3…

*lmfit.emcee* can be used to obtain the posterior probability distribution of parameters, given a set of experimental data. This notebook shows how it can be used for Bayesian model selection.

```
import matplotlib.pyplot as plt
import numpy as np

import lmfit
```

Define a Gaussian lineshape and generate some data:

```
def gauss(x, a_max, loc, sd):
    return a_max * np.exp(-((x - loc) / sd)**2)


x = np.linspace(3, 7, 250)
np.random.seed(0)
y = 4 + 10 * x + gauss(x, 200, 5, 0.5) + gauss(x, 60, 5.8, 0.2)
```

(continues on next page)

(continued from previous page)

```
dy = np.sqrt(y)
y += dy * np.random.randn(y.size)
```

Plot the data:

```
plt.errorbar(x, y)
```

Define the normalised residual for the data:

```
def residual(p, just_generative=False):
    v = p.valuesdict()
    generative = v['a'] + v['b'] * x
    M = 0
    while 'a_max%d' % M in v:
        generative += gauss(x, v['a_max%d' % M], v['loc%d' % M], v['sd%d' % M])
        M += 1

    if just_generative:
        return generative
    return (generative - y) / dy
```

Create a Parameter set for the initial guesses:

```
def initial_peak_params(M):
    p = lmfit.Parameters()

    # a and b give a linear background
    a = np.mean(y)
    b = 1

    # a_max, loc and sd are the amplitude, location and SD of each Gaussian
    # component
    a_max = np.max(y)
    loc = np.mean(x)
    sd = (np.max(x) - np.min(x)) * 0.5

    p.add_many(('a', a, True, 0, 10), ('b', b, True, 1, 15))

    for i in range(M):
        p.add_many(('a_max%d' % i, 0.5 * a_max, True, 10, a_max),
                   ('loc%d' % i, loc, True, np.min(x), np.max(x)),
                   ('sd%d' % i, sd, True, 0.1, np.max(x) - np.min(x)))
    return p
```

Solving with *minimize* gives the Maximum Likelihood solution.

```
p1 = initial_peak_params(1)
mi1 = lmfit.minimize(residual, p1, method='differential_evolution')

lmfit.printfuncs.report_fit(mi1.params, min_correl=0.5)
```

From inspection of the data above we can tell that there is going to be more than 1 Gaussian component, but how many are there? A Bayesian approach can be used for this model selection problem. We can do this with *lmfit.emcee*, which uses the *emcee* package to do a Markov Chain Monte Carlo sampling of the posterior probability distribution. *lmfit.emcee* requires a function that returns the log-posterior probability. The log-posterior probability is a sum of the log-prior probability and log-likelihood functions.

The log-prior probability encodes information about what you already believe about the system. *lmfit.emcee* assumes that this log-prior probability is zero if all the parameters are within their bounds and *-np.inf* if any of the parameters are outside their bounds. As such it's a uniform prior.

The log-likelihood function is given below. To use non-uniform priors then should include these terms in *lnprob*. This is the log-likelihood probability for the sampling.

```
def lnprob(p):
    resid = residual(p, just_generative=True)
    return -0.5 * np.sum(((resid - y) / dy)**2 + np.log(2 * np.pi * dy**2))
```

To start with we have to create the minimizers and *burn* them in. We create 4 different minimizers representing 0, 1, 2 or 3 Gaussian contributions. To do the model selection we have to integrate the over the log-posterior distribution to see which has the higher probability. This is done using the *thermodynamic_integration_log_evidence* method of the *sampler* attribute contained in the *lmfit.Minimizer* object.

```
# Work out the log-evidence for different numbers of peaks:
total_steps = 310
burn = 300
thin = 10
ntemps = 15
workers = 1  # the multiprocessing does not work with sphinx-gallery
log_evidence = []
res = []

# set up the Minimizers
for i in range(4):
    p0 = initial_peak_params(i)
    # you can't use lnprob as a userfcn with minimize because it needs to be
    # maximised
    mini = lmfit.Minimizer(residual, p0)
    out = mini.minimize(method='differential_evolution')
    res.append(out)

mini = []
# burn in the samplers
for i in range(4):
    # do the sampling
    mini.append(lmfit.Minimizer(lnprob, res[i].params))
    out = mini[i].emcee(steps=total_steps, ntemps=ntemps, workers=workers,
                        reuse_sampler=False, float_behavior='posterior',
                        progress=False)
    # get the evidence
    print(i, total_steps, mini[i].sampler.thermodynamic_integration_log_evidence())
    log_evidence.append(mini[i].sampler.thermodynamic_integration_log_evidence()[0])
```

Once we've burned in the samplers we have to do a collection run. We thin out the MCMC chain to reduce autocorrelation between successive samples.

```
for j in range(6):
    total_steps += 100
    for i in range(4):
        # do the sampling
        res = mini[i].emcee(burn=burn, steps=100, thin=thin, ntemps=ntemps,
                            workers=workers, reuse_sampler=True, progress=False)
        # get the evidence
        print(i, total_steps, mini[i].sampler.thermodynamic_integration_log_
↪evidence())
```

(continues on next page)

(continued from previous page)

```
        log_evidence.append(mini[i].sampler.thermodynamic_integration_log_
↪evidence()[0])


plt.plot(log_evidence[-4:])
plt.ylabel('Log-evidence')
plt.xlabel('number of peaks')
```

The Bayes factor is related to the exponential of the difference between the log-evidence values. Thus, 0 peaks is not very likely compared to 1 peak. But 1 peak is not as good as 2 peaks. 3 peaks is not that much better than 2 peaks.

```
r01 = np.exp(log_evidence[-4] - log_evidence[-3])
r12 = np.exp(log_evidence[-3] - log_evidence[-2])
r23 = np.exp(log_evidence[-2] - log_evidence[-1])

print(r01, r12, r23)
```

These numbers tell us that zero peaks is 0 times as likely as one peak. Two peaks is 7e49 times more likely than one peak. Three peaks is 1.1 times more likely than two peaks. With this data one would say that two peaks is sufficient. Caution has to be taken with these values. The log-priors for this sampling are uniform but improper, i.e. they are not normalised properly. Internally the lnprior probability is calculated as 0 if all parameters are within their bounds and *-np.inf* if any parameter is outside the bounds. The *lnprob* function defined above is the log-likelihood alone. Remember, that the log-posterior probability is equal to the sum of the log-prior and log-likelihood probabilities. Extra terms can be added to the lnprob function to calculate the normalised log-probability. These terms would look something like:

$$\log(\prod_i \frac{1}{max_i - min_i})$$

where $max_i$ and $min_i$ are the upper and lower bounds for the parameter, and the prior is a uniform distribution. Other types of prior are possible. For example, you might expect the prior to be Gaussian.

**Total running time of the script:** ( 0 minutes 0.000 seconds)

## 13.17 Fit Two Dimensional Peaks

This example illustrates how to handle two-dimensional data with lmfit.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import griddata

import lmfit
from lmfit.lineshapes import gaussian2d, lorentzian
```

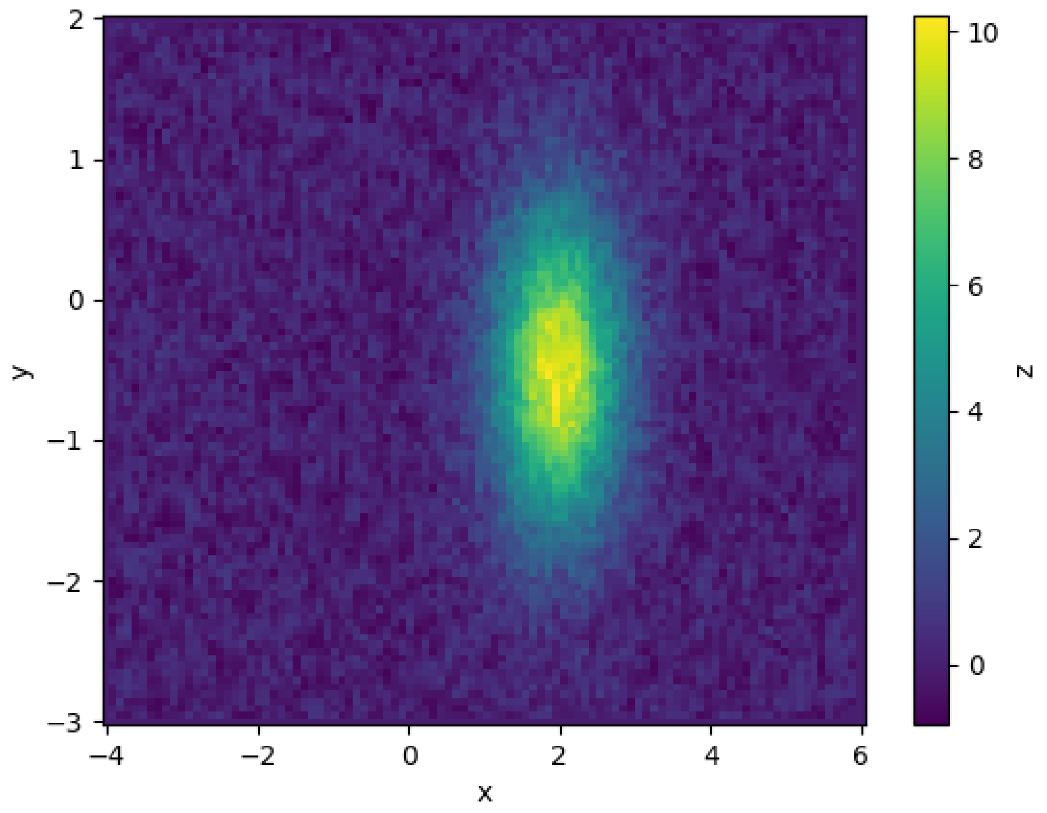
### 13.17.1 Two-dimensional Gaussian

We start by considering a simple two-dimensional gaussian function, which depends on coordinates *(x, y)*. The most general case of experimental data will be irregularly sampled and noisy. Let's simulate some:

```
npoints = 10000
x = np.random.rand(npoints)*10 - 4
y = np.random.rand(npoints)*5 - 3
z = gaussian2d(x, y, amplitude=30, centerx=2, centery=-.5, sigmax=.6, sigmay=.8)
z += 2*(np.random.rand(*z.shape)-.5)
error = np.sqrt(z+1)
```

To plot this, we can interpolate the data onto a grid.

```
X, Y = np.meshgrid(np.linspace(x.min(), x.max(), 100),
                   np.linspace(y.min(), y.max(), 100))
Z = griddata((x, y), z, (X, Y), method='linear', fill_value=0)

fig, ax = plt.subplots()
art = ax.pcolor(X, Y, Z, shading='auto')
plt.colorbar(art, ax=ax, label='z')
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```



In this case, we can use a built-in model to fit

```
model = lmfit.models.Gaussian2dModel()
params = model.guess(z, x, y)
result = model.fit(z, x=x, y=y, params=params, weights=1/error)
lmfit.report_fit(result)
```

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 74
    # data points      = 10000
    # variables        = 5
    chi-square         = 23221.7077
    reduced chi-square = 2.32333244
    Akaike info crit   = 8435.02425
    Bayesian info crit = 8471.07595
[[Variables]]
    amplitude:  27.6616344 +/- 0.68573297 (2.48%) (init = 19.87323)
    centerx:    1.99810718 +/- 0.01471383 (0.74%) (init = 2.123473)
    centery:   -0.49717812 +/- 0.02011906 (4.05%) (init = -0.6082653)
    sigmax:     0.55187429 +/- 0.01262250 (2.29%) (init = 1.66637)
    sigmay:     0.73663336 +/- 0.01696361 (2.30%) (init = 0.8328972)
    fwhmx:      1.29956462 +/- 0.02972371 (2.29%) == '2.3548200*sigmax'
    fwhmy:      1.73463898 +/- 0.03994626 (2.30%) == '2.3548200*sigmay'
    height:     106.882390 +/- 3.63826498 (3.40%) == '1.5707963*amplitude/(max(1e-15,␣
↪sigmax)*max(1e-15, sigmay))'
[[Correlations]] (unreported correlations are < 0.100)
    C(amplitude, sigmay) =  0.253
    C(amplitude, sigmax) =  0.220
```

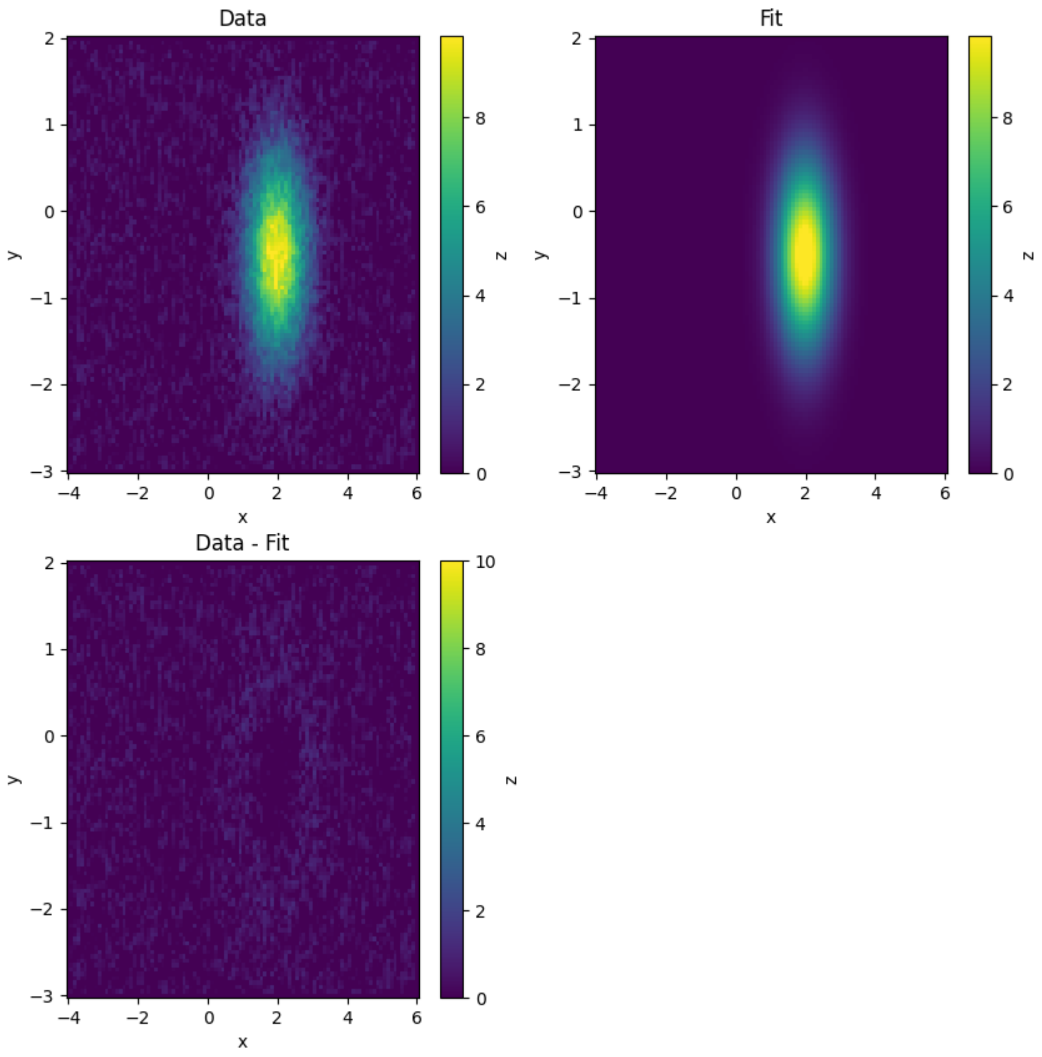
To check the fit, we can evaluate the function on the same grid we used before and make plots of the data, the fit and the difference between the two.

```
fig, axs = plt.subplots(2, 2, figsize=(10, 10))

vmax = np.nanpercentile(Z, 99.9)

ax = axs[0, 0]
art = ax.pcolor(X, Y, Z, vmin=0, vmax=vmax, shading='auto')
plt.colorbar(art, ax=ax, label='z')
ax.set_title('Data')

ax = axs[0, 1]
fit = model.func(X, Y, **result.best_values)
art = ax.pcolor(X, Y, fit, vmin=0, vmax=vmax, shading='auto')
plt.colorbar(art, ax=ax, label='z')
ax.set_title('Fit')

ax = axs[1, 0]
fit = model.func(X, Y, **result.best_values)
art = ax.pcolor(X, Y, Z-fit, vmin=0, vmax=10, shading='auto')
plt.colorbar(art, ax=ax, label='z')
ax.set_title('Data - Fit')

for ax in axs.ravel():
    ax.set_xlabel('x')
    ax.set_ylabel('y')
```

(continues on next page)

(continued from previous page)

```
axs[1, 1].remove()
plt.show()
```



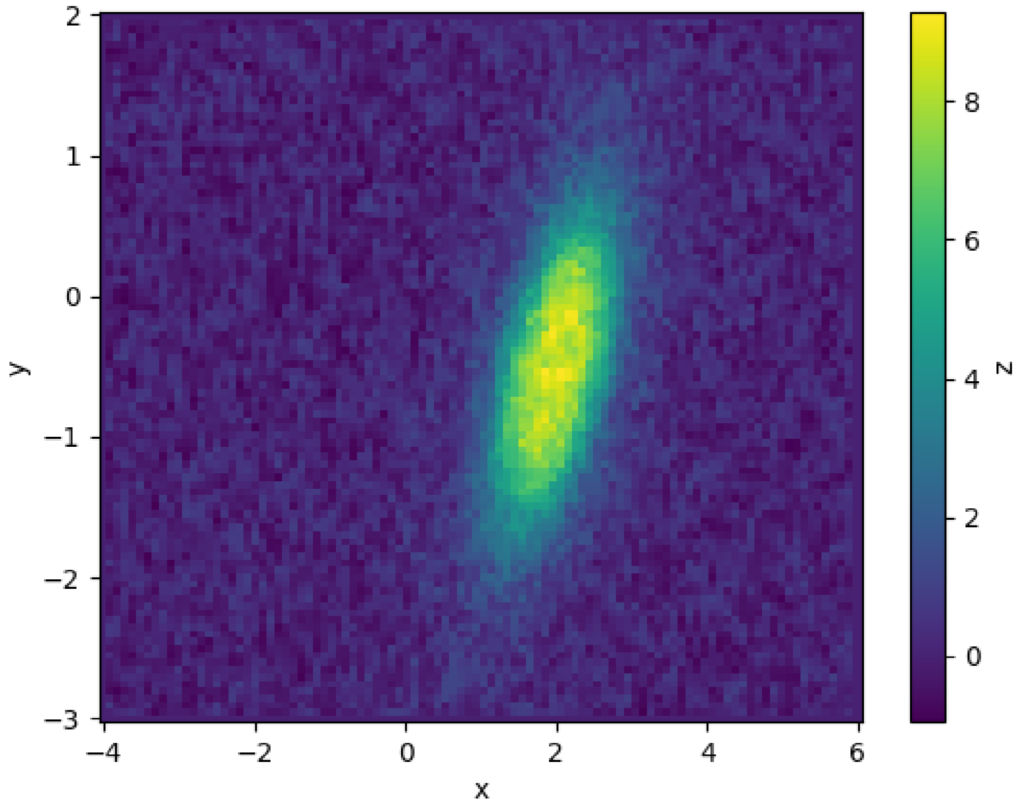
### 13.17.2 Two-dimensional off-axis Lorentzian

We now go on to show a harder example, in which the peak has a Lorentzian profile and an off-axis anisotropic shape. This can be handled by applying a suitable coordinate transform and then using the *lorentzian* function that lmfit provides in the lineshapes module.

```
def lorentzian2d(x, y, amplitude=1., centerx=0., centery=0., sigmax=1., sigmay=1.,
                 rotation=0):
    """Return a two dimensional lorentzian.

    The maximum of the peak occurs at ``centerx`` and ``centery``
    with widths ``sigmax`` and ``sigmay`` in the x and y directions
    respectively. The peak can be rotated by choosing the value of ``rotation``
    in radians.
    """
    xp = (x - centerx)*np.cos(rotation) - (y - centery)*np.sin(rotation)
    yp = (x - centerx)*np.sin(rotation) + (y - centery)*np.cos(rotation)
    R = (xp/sigmax)**2 + (yp/sigmay)**2

    return 2*amplitude*lorentzian(R)/(np.pi*sigmax*sigmay)
```

Data can be simulated and plotted in the same way as we did before.

```
npoints = 10000
x = np.random.rand(npoints)*10 - 4
y = np.random.rand(npoints)*5 - 3
z = lorentzian2d(x, y, amplitude=30, centerx=2, centery=-.5, sigmax=.6,
                 sigmay=1.2, rotation=30*np.pi/180)
z += 2*(np.random.rand(*z.shape)-.5)
error = np.sqrt(z+1)

X, Y = np.meshgrid(np.linspace(x.min(), x.max(), 100),
                   np.linspace(y.min(), y.max(), 100))
Z = griddata((x, y), z, (X, Y), method='linear', fill_value=0)

fig, ax = plt.subplots()
ax.set_xlabel('x')
ax.set_ylabel('y')
art = ax.pcolor(X, Y, Z, shading='auto')
plt.colorbar(art, ax=ax, label='z')
plt.show()
```



To fit, create a model from the function. Don't forget to tell lmfit that both *x* and *y* are independent variables. Keep in mind that lmfit will take the function keywords as default initial guesses in this case and that it will not know that certain parameters only make physical sense over restricted ranges. For example, peak widths should be positive and the rotation can be restricted over a quarter circle.

```
model = lmfit.Model(lorentzian2d, independent_vars=['x', 'y'])
params = model.make_params(amplitude=10, centerx=x[np.argmax(z)],
                           centery=y[np.argmax(z)])
params['rotation'].set(value=.1, min=0, max=np.pi/2)
params['sigmax'].set(value=1, min=0)
params['sigmay'].set(value=2, min=0)

result = model.fit(z, x=x, y=y, params=params, weights=1/error)
lmfit.report_fit(result)
```

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 73
    # data points      = 10000
    # variables        = 6
    chi-square         = 11554.1100
    reduced chi-square = 1.15610466
    Akaike info crit   = 1456.56121
```

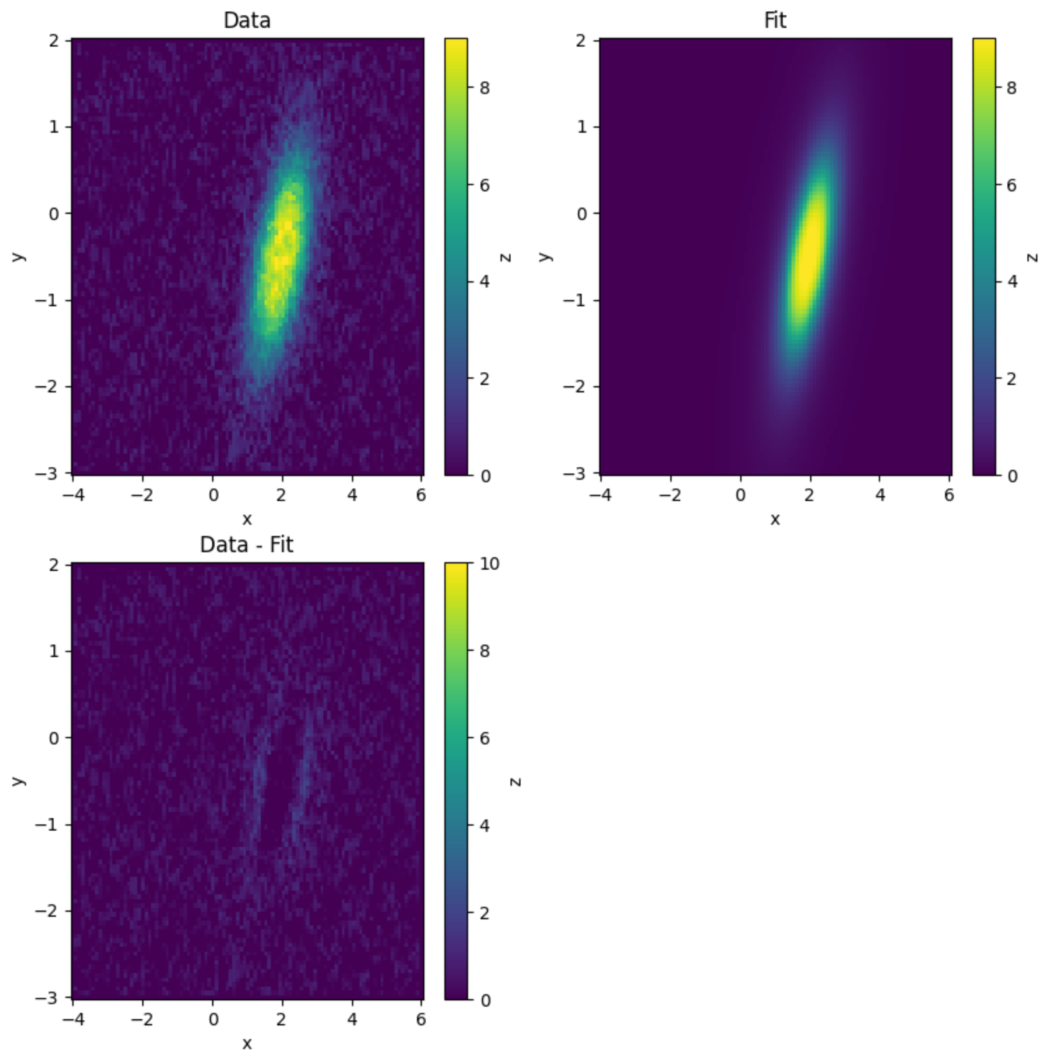
(continues on next page)

(continued from previous page)

```
    Bayesian info crit = 1499.82325
[[Variables]]
    amplitude:  25.4520941 +/- 0.50893773 (2.00%) (init = 10)
    centerx:    1.99532738 +/- 0.01162425 (0.58%) (init = 2.077652)
    centery:   -0.50221822 +/- 0.01657490 (3.30%) (init = -0.5359784)
    sigmax:     0.50576995 +/- 0.01047067 (2.07%) (init = 1)
    sigmay:     1.11968508 +/- 0.02191517 (1.96%) (init = 2)
    rotation:   0.48387509 +/- 0.01534530 (3.17%) (init = 0.1)
[[Correlations]] (unreported correlations are < 0.100)
    C(centerx, centery)   =  0.537
    C(amplitude, sigmax) =  0.371
    C(amplitude, sigmay) =  0.250
```

The process of making plots to check it worked is the same as before.

```
fig, axs = plt.subplots(2, 2, figsize=(10, 10))

vmax = np.nanpercentile(Z, 99.9)

ax = axs[0, 0]
art = ax.pcolor(X, Y, Z, vmin=0, vmax=vmax, shading='auto')
plt.colorbar(art, ax=ax, label='z')
ax.set_title('Data')

ax = axs[0, 1]
fit = model.func(X, Y, **result.best_values)
art = ax.pcolor(X, Y, fit, vmin=0, vmax=vmax, shading='auto')
plt.colorbar(art, ax=ax, label='z')
ax.set_title('Fit')

ax = axs[1, 0]
fit = model.func(X, Y, **result.best_values)
art = ax.pcolor(X, Y, Z-fit, vmin=0, vmax=10, shading='auto')
plt.colorbar(art, ax=ax, label='z')
ax.set_title('Data - Fit')

for ax in axs.ravel():
    ax.set_xlabel('x')
    ax.set_ylabel('y')
axs[1, 1].remove()
plt.show()
```



**Total running time of the script:** ( 2 minutes 37.017 seconds)

## 13.18 Global minimization using the `brute` method (a.k.a. grid search)

This notebook shows a simple example of using `lmfit.minimize.brute` that uses the method with the same name from `scipy.optimize`.

The method computes the function's value at each point of a multidimensional grid of points, to find the global minimum of the function. It behaves identically to `scipy.optimize.brute` in case finite bounds are given on all varying parameters, but will also deal with non-bounded parameters (see below).

```python
import copy

from matplotlib.colors import LogNorm
import matplotlib.pyplot as plt
import numpy as np

from lmfit import Minimizer, Parameters, fit_report
```

Let's start with the example given in the documentation of SciPy:

"We illustrate the use of brute to seek the global minimum of a function of two variables that is given as the sum of a positive-definite quadratic and two deep "Gaussian-shaped" craters. Specifically, define the objective function f as the sum of three other functions, `f = f1 + f2 + f3`. We suppose each of these has a signature `(z, *params)`, where `z = (x, y)`, and params and the functions are as defined below."

First, we create a set of Parameters where all variables except `x` and `y` are given fixed values.

```python
params = Parameters()
params.add_many(
        ('a', 2, False),
        ('b', 3, False),
        ('c', 7, False),
        ('d', 8, False),
        ('e', 9, False),
        ('f', 10, False),
        ('g', 44, False),
        ('h', -1, False),
        ('i', 2, False),
        ('j', 26, False),
        ('k', 1, False),
        ('l', -2, False),
        ('scale', 0.5, False),
        ('x', 0.0, True),
        ('y', 0.0, True))
```

Second, create the three functions and the objective function:

```python
def f1(p):
    par = p.valuesdict()
    return (par['a'] * par['x']**2 + par['b'] * par['x'] * par['y'] +
            par['c'] * par['y']**2 + par['d']*par['x'] + par['e']*par['y'] +
            par['f'])


def f2(p):
    par = p.valuesdict()
    return (-1.0*par['g']*np.exp(-((par['x']-par['h'])**2 +
                                   (par['y']-par['i'])**2) / par['scale']))


def f3(p):
    par = p.valuesdict()
    return (-1.0*par['j']*np.exp(-((par['x']-par['k'])**2 +
                                   (par['y']-par['l'])**2) / par['scale']))


def f(params):
    return f1(params) + f2(params) + f3(params)
```

Just as in the documentation we will do a grid search between `-4` and `4` and use a stepsize of `0.25`. The bounds can be set as usual with the `min` and `max` attributes, and the stepsize is set using `brute_step`.

```
params['x'].set(min=-4, max=4, brute_step=0.25)
params['y'].set(min=-4, max=4, brute_step=0.25)
```

Performing the actual grid search is done with:

```
fitter = Minimizer(f, params)
result = fitter.minimize(method='brute')
```

, which will increment `x` and `y` between `-4` in increments of `0.25` until `4` (not inclusive).

```
grid_x, grid_y = [np.unique(par.ravel()) for par in result.brute_grid]
print(grid_x)
```

Out:

```
[-4.   -3.75 -3.5  -3.25 -3.   -2.75 -2.5  -2.25 -2.   -1.75 -1.5  -1.25
 -1.   -0.75 -0.5  -0.25  0.    0.25  0.5   0.75  1.    1.25  1.5   1.75
  2.    2.25  2.5   2.75  3.    3.25  3.5   3.75]
```

The objective function is evaluated on this grid, and the raw output from `scipy.optimize.brute` is stored in the MinimizerResult as `brute_<parname>` attributes. These attributes are:

`result.brute_x0` – A 1-D array containing the coordinates of a point at which the objective function had its minimum value.

```
print(result.brute_x0)
```

Out:

```
[-1.    1.75]
```

`result.brute_fval` – Function value at the point x0.

```
print(result.brute_fval)
```

Out:

```
-2.8923637137222027
```

`result.brute_grid` – Representation of the evaluation grid. It has the same length as x0.

```
print(result.brute_grid)
```

Out:

```
[[[-4.   -4.   -4.   ... -4.   -4.   -4.  ]
  [-3.75 -3.75 -3.75 ... -3.75 -3.75 -3.75]
  [-3.5  -3.5  -3.5  ... -3.5  -3.5  -3.5 ]
  ...
  [ 3.25  3.25  3.25 ...  3.25  3.25  3.25]
  [ 3.5   3.5   3.5  ...  3.5   3.5   3.5 ]
  [ 3.75  3.75  3.75 ...  3.75  3.75  3.75]]

 [[-4.   -3.75 -3.5  ...  3.25  3.5   3.75]
  [-4.   -3.75 -3.5  ...  3.25  3.5   3.75]
```

(continues on next page)

(continued from previous page)

```
  [-4.   -3.75 -3.5  ...  3.25  3.5   3.75]
  ...
  [-4.   -3.75 -3.5  ...  3.25  3.5   3.75]
  [-4.   -3.75 -3.5  ...  3.25  3.5   3.75]
  [-4.   -3.75 -3.5  ...  3.25  3.5   3.75]]]
```

`result.brute_Jout` – Function values at each point of the evaluation grid, i.e., Jout = func(*grid).

```
print(result.brute_Jout)
```

Out:

```
[[134.         119.6875     106.25        ...  74.18749997  85.24999999
   97.1875    ]
 [129.125      115.         101.75        ...  74.74999948  85.99999987
   98.12499997]
 [124.5        110.5625      97.5         ...  75.5624928   86.99999818
   99.31249964]
 ...
 [ 94.12499965  85.24999772  77.24998843 ... 192.         208.5
  225.875     ]
 [ 96.49999997  87.81249979  79.99999892 ... 199.8125     216.5
  234.0625    ]
 [ 99.125       90.62499998  82.99999992 ... 207.875      224.75
  242.5       ]]
```

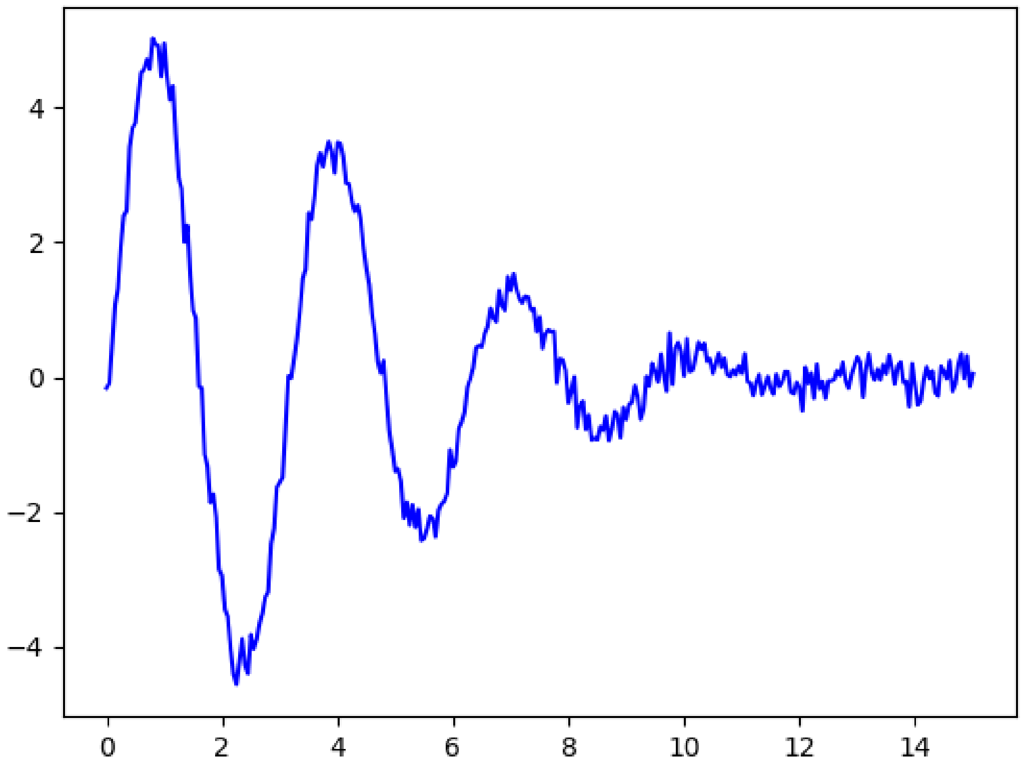
**Reassuringly, the obtained results are identical to using the method in SciPy directly!**

Example 2: fit of a decaying sine wave

In this example, we will explain some of the options of the algorithm.

We start off by generating some synthetic data with noise for a decaying sine wave, define an objective function and create a Parameter set.

```
x = np.linspace(0, 15, 301)
np.random.seed(7)
noise = np.random.normal(size=x.size, scale=0.2)
data = (5. * np.sin(2*x - 0.1) * np.exp(-x*x*0.025) + noise)
plt.plot(x, data, 'b')


def fcn2min(params, x, data):
    """Model decaying sine wave, subtract data."""
    amp = params['amp']
    shift = params['shift']
    omega = params['omega']
    decay = params['decay']
    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=7, min=2.5)
params.add('decay', value=0.05)
params.add('shift', value=0.0, min=-np.pi/2., max=np.pi/2)
params.add('omega', value=3, max=5)
```



In contrast to the implementation in SciPy (as shown in the first example), varying parameters do not need to have finite bounds in lmfit. However, in that case they **do** need the `brute_step` attribute specified, so let's do that:

```
params['amp'].set(brute_step=0.25)
params['decay'].set(brute_step=0.005)
params['omega'].set(brute_step=0.25)
```

Our initial parameter set is now defined as shown below and this will determine how the grid is set-up.

```
params.pretty_print()
```

Out:

```
Name       Value       Min       Max    Stderr      Vary      Expr Brute_Step
amp            7       2.5       inf      None      True      None       0.25
decay       0.05      -inf       inf      None      True      None      0.005
omega          3      -inf         5      None      True      None       0.25
shift          0    -1.571     1.571      None      True      None       None
```

First, we initialize a Minimizer and perform the grid search:

```
fitter = Minimizer(fcn2min, params, fcn_args=(x, data))
result_brute = fitter.minimize(method='brute', Ns=25, keep=25)
```

We used two new parameters here: `Ns` and `keep`. The parameter `Ns` determines the 'number of grid points along the axes' similarly to its usage in SciPy. Together with `brute_step`, `min` and `max` for a Parameter it will dictate how the grid is set-up:

**(1)** finite bounds are specified ("SciPy implementation"): uses `brute_step` if present (in the example above) or uses `Ns` to generate the grid. The latter scenario that interpolates `Ns` points from `min` to `max` (inclusive), is here shown for the parameter `shift`:

```
par_name = 'shift'
indx_shift = result_brute.var_names.index(par_name)
grid_shift = np.unique(result_brute.brute_grid[indx_shift].ravel())
print("parameter = {}\nnumber of steps = {}\ngrid = {}".format(par_name,
                                                               len(grid_shift),
                                                               grid_shift))
```

Out:

```
parameter = shift
number of steps = 25
grid = [-1.57079633 -1.43989663 -1.30899694 -1.17809725 -1.04719755 -0.91629786
 -0.78539816 -0.65449847 -0.52359878 -0.39269908 -0.26179939 -0.13089969
  0.          0.13089969  0.26179939  0.39269908  0.52359878  0.65449847
  0.78539816  0.91629786  1.04719755  1.17809725  1.30899694  1.43989663
  1.57079633]
```

If finite bounds are not set for a certain parameter then the user **must** specify `brute_step` - three more scenarios are considered here:

**(2)** lower bound (min) and brute_step are specified: range = (min, min + Ns * brute_step, brute_step)

```
par_name = 'amp'
indx_shift = result_brute.var_names.index(par_name)
grid_shift = np.unique(result_brute.brute_grid[indx_shift].ravel())
print("parameter = {}\nnumber of steps = {}\ngrid = {}".format(par_name, len(grid_
↪shift), grid_shift))
```

Out:

```
parameter = amp
number of steps = 25
grid = [2.5  2.75 3.   3.25 3.5  3.75 4.   4.25 4.5  4.75 5.   5.25 5.5  5.75
 6.   6.25 6.5  6.75 7.   7.25 7.5  7.75 8.   8.25 8.5 ]
```

**(3)** upper bound (max) and brute_step are specified: range = (max - Ns * brute_step, max, brute_step)

```
par_name = 'omega'
indx_shift = result_brute.var_names.index(par_name)
grid_shift = np.unique(result_brute.brute_grid[indx_shift].ravel())
print("parameter = {}\nnumber of steps = {}\ngrid = {}".format(par_name, len(grid_
↪shift), grid_shift))
```

Out:

```
parameter = omega
number of steps = 25
grid = [-1.25 -1.   -0.75 -0.5  -0.25  0.    0.25  0.5   0.75  1.    1.25  1.5
  1.75  2.    2.25  2.5   2.75  3.    3.25  3.5   3.75  4.    4.25  4.5
  4.75]
```

**(4)** numerical value (value) and brute_step are specified: range = (value - (Ns//2) * brute_step, value + (Ns//2) * brute_step, brute_step)

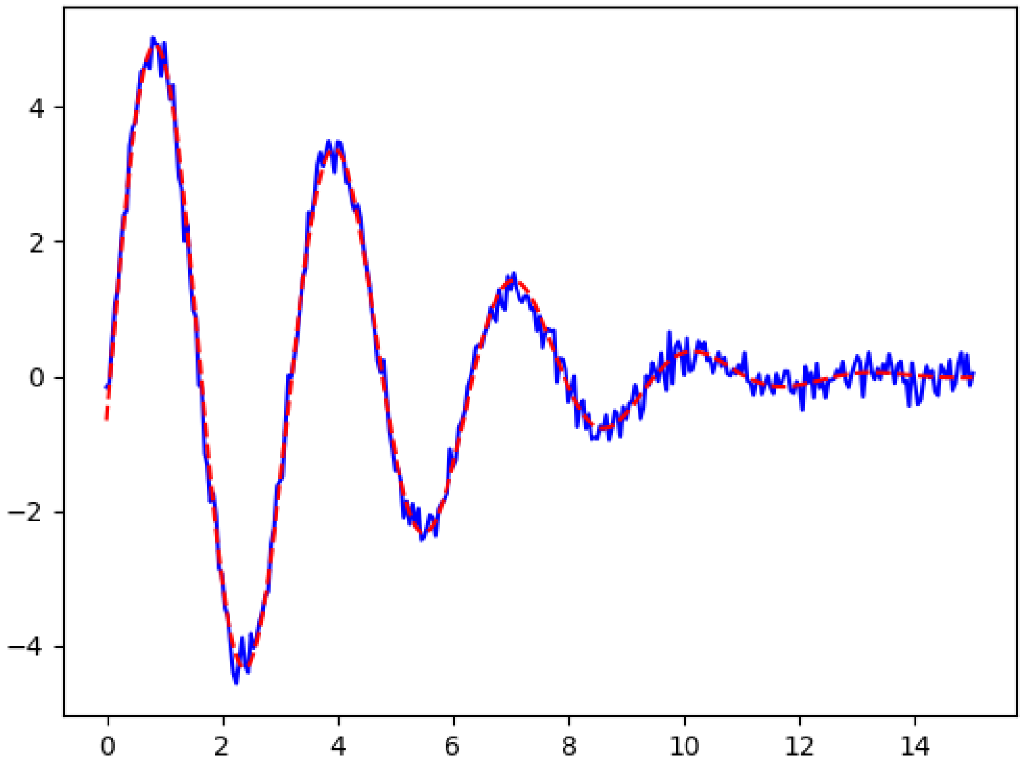
```
par_name = 'decay'
indx_shift = result_brute.var_names.index(par_name)
grid_shift = np.unique(result_brute.brute_grid[indx_shift].ravel())
print("parameter = {}\nnumber of steps = {}\ngrid = {}".format(par_name, len(grid_
↪shift), grid_shift))
```

Out:

```
parameter = decay
number of steps = 24
grid = [-1.00000000e-02 -5.00000000e-03  5.20417043e-18  5.00000000e-03
  1.00000000e-02  1.50000000e-02  2.00000000e-02  2.50000000e-02
  3.00000000e-02  3.50000000e-02  4.00000000e-02  4.50000000e-02
  5.00000000e-02  5.50000000e-02  6.00000000e-02  6.50000000e-02
  7.00000000e-02  7.50000000e-02  8.00000000e-02  8.50000000e-02
  9.00000000e-02  9.50000000e-02  1.00000000e-01  1.05000000e-01]
```

The `MinimizerResult` contains all the usual best-fit parameters and fitting statistics. For example, the optimal solution from the grid search is given below together with a plot:

```
print(fit_report(result_brute))
plt.plot(x, data, 'b')
plt.plot(x, data + fcn2min(result_brute.params, x, data), 'r--')
```



Out:

```
[[Fit Statistics]]
    # fitting method   = brute
    # function evals   = 375000
    # data points      = 301
    # variables        = 4
    chi-square         = 11.9353671
    reduced chi-square = 0.04018642
    Akaike info crit   = -963.508878
    Bayesian info crit = -948.680437
##  Warning: uncertainties could not be estimated:
[[Variables]]
    amp:    5.00000000 (init = 7)
    decay:  0.02500000 (init = 0.05)
    shift: -0.13089969 (init = 0)
    omega:  2.00000000 (init = 3)

[<matplotlib.lines.Line2D object at 0x1243d8070>]
```

We can see that this fit is already very good, which is what we should expect since our `brute` force grid is sampled rather finely and encompasses the “correct” values.

In a more realistic, complicated example the `brute` method will be used to get reasonable values for the parameters and perform another minimization (e.g., using `leastsq`) using those as starting values. That is where the *keep*` parameter comes into play: it determines the “number of best candidates from the brute force method that are stored in the `candidates` attribute”. In the example above we store the best-ranking 25 solutions (the default value is `50` and storing all the grid points can be accomplished by choosing `all`). The `candidates` attribute contains the parameters and `chisqr` from the brute force method as a namedtuple, `('Candidate', ['params', 'score'])`, sorted on the (lowest) `chisqr` value. To access the values for a particular candidate one can use `result.candidate[#].params` or `result.candidate[#].score`, where a lower # represents a better candidate. The `show_candidates(#)` uses the `pretty_print()` method to show a specific candidate-# or all candidates when no number is specified.

The optimal fit is, as usual, stored in the `MinimizerResult.params` attribute and is, therefore, identical to `result_brute.show_candidates(1)`.

```
result_brute.show_candidates(1)
```

Out:

```
Candidate #1, chisqr = 11.935
Name      Value      Min      Max   Stderr     Vary     Expr Brute_Step
amp           5      2.5      inf     None     True     None     0.25
decay     0.025     -inf      inf     None     True     None    0.005
omega         2     -inf        5     None     True     None     0.25
shift   -0.1309   -1.571    1.571     None     True     None     None
```

In this case, the next-best scoring candidate has already a `chisqr` that increased quite a bit:

```
result_brute.show_candidates(2)
```

Out:

```
Candidate #2, chisqr = 13.994
Name      Value      Min      Max   Stderr     Vary     Expr Brute_Step
amp        4.75      2.5      inf     None     True     None     0.25
decay     0.025     -inf      inf     None     True     None    0.005
```

(continues on next page)

(continued from previous page)

```
omega          2      -inf         5      None      True      None      0.25
shift    -0.1309    -1.571     1.571      None      True      None      None
```

and is, therefore, probably not so likely... However, as said above, in most cases you'll want to do another minimization using the solutions from the `brute` method as starting values. That can be easily accomplished as shown in the code below, where we now perform a `leastsq` minimization starting from the top-25 solutions and accept the solution if the `chisqr` is lower than the previously 'optimal' solution:

```
best_result = copy.deepcopy(result_brute)

for candidate in result_brute.candidates:
    trial = fitter.minimize(method='leastsq', params=candidate.params)
    if trial.chisqr < best_result.chisqr:
        best_result = trial
```

From the `leastsq` minimization we obtain the following parameters for the most optimal result:

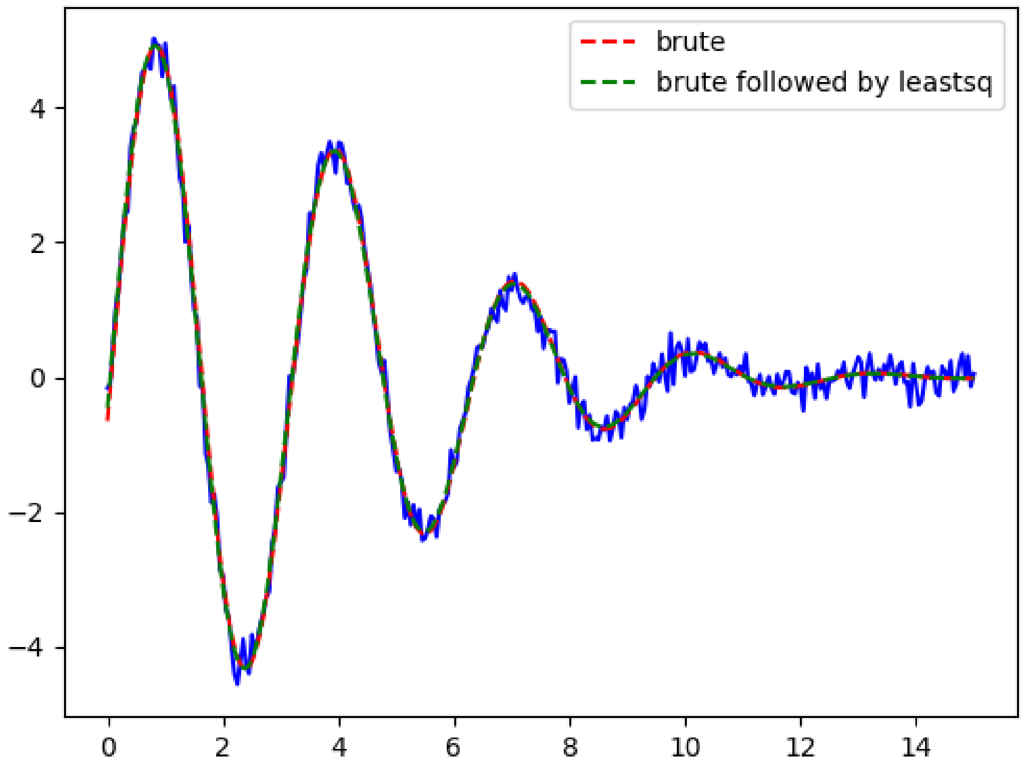
```
print(fit_report(best_result))
```

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 21
    # data points      = 301
    # variables        = 4
    chi-square         = 10.8653514
    reduced chi-square = 0.03658367
    Akaike info crit   = -991.780924
    Bayesian info crit = -976.952483
[[Variables]]
    amp:     5.00323088 +/- 0.03805940 (0.76%) (init = 5.25)
    decay:   0.02563850 +/- 4.4571e-04 (1.74%) (init = 0.025)
    shift: -0.09162988 +/- 0.00978382 (10.68%) (init = 0)
    omega:   1.99611629 +/- 0.00316225 (0.16%) (init = 2)
[[Correlations]] (unreported correlations are < 0.100)
    C(shift, omega) = -0.785
    C(amp, decay)   =  0.584
    C(amp, shift)   = -0.121
```

As expected the parameters have not changed significantly as they were already very close to the "real" values, which can also be appreciated from the plots below.

```
plt.plot(x, data, 'b')
plt.plot(x, data + fcn2min(result_brute.params, x, data), 'r--',
         label='brute')
plt.plot(x, data + fcn2min(best_result.params, x, data), 'g--',
         label='brute followed by leastsq')
plt.legend()
```



Finally, the results from the `brute` force grid-search can be visualized using the rather lengthy Python function below (which might get incorporated in lmfit at some point).

```
def plot_results_brute(result, best_vals=True, varlabels=None,
                       output=None):
    """Visualize the result of the brute force grid search.

    The output file will display the chi-square value per parameter and contour
    plots for all combination of two parameters.

    Inspired by the `corner` package (https://github.com/dfm/corner.py).

    Parameters
    ----------
    result : :class:`~lmfit.minimizer.MinimizerResult`
        Contains the results from the :meth:`brute` method.

    best_vals : bool, optional
        Whether to show the best values from the grid search (default is True).

    varlabels : list, optional
        If None (default), use `result.var_names` as axis labels, otherwise
        use the names specified in `varlabels`.

    output : str, optional
        Name of the output PDF file (default is 'None')
```

(continues on next page)

(continued from previous page)

```
    """
    npars = len(result.var_names)
    _fig, axes = plt.subplots(npars, npars)

    if not varlabels:
        varlabels = result.var_names
    if best_vals and isinstance(best_vals, bool):
        best_vals = result.params

    for i, par1 in enumerate(result.var_names):
        for j, par2 in enumerate(result.var_names):

            # parameter vs chi2 in case of only one parameter
            if npars == 1:
                axes.plot(result.brute_grid, result.brute_Jout, 'o', ms=3)
                axes.set_ylabel(r'$\chi^{2}$')
                axes.set_xlabel(varlabels[i])
                if best_vals:
                    axes.axvline(best_vals[par1].value, ls='dashed', color='r')

            # parameter vs chi2 profile on top
            elif i == j and j < npars-1:
                if i == 0:
                    axes[0, 0].axis('off')
                ax = axes[i, j+1]
                red_axis = tuple([a for a in range(npars) if a != i])
                ax.plot(np.unique(result.brute_grid[i]),
                        np.minimum.reduce(result.brute_Jout, axis=red_axis),
                        'o', ms=3)
                ax.set_ylabel(r'$\chi^{2}$')
                ax.yaxis.set_label_position("right")
                ax.yaxis.set_ticks_position('right')
                ax.set_xticks([])
                if best_vals:
                    ax.axvline(best_vals[par1].value, ls='dashed', color='r')

            # parameter vs chi2 profile on the left
            elif j == 0 and i > 0:
                ax = axes[i, j]
                red_axis = tuple([a for a in range(npars) if a != i])
                ax.plot(np.minimum.reduce(result.brute_Jout, axis=red_axis),
                        np.unique(result.brute_grid[i]), 'o', ms=3)
                ax.invert_xaxis()
                ax.set_ylabel(varlabels[i])
                if i != npars-1:
                    ax.set_xticks([])
                else:
                    ax.set_xlabel(r'$\chi^{2}$')
                if best_vals:
                    ax.axhline(best_vals[par1].value, ls='dashed', color='r')

            # contour plots for all combinations of two parameters
            elif j > i:
                ax = axes[j, i+1]
                red_axis = tuple([a for a in range(npars) if a not in (i, j)])
                X, Y = np.meshgrid(np.unique(result.brute_grid[i]),
                                   np.unique(result.brute_grid[j]))
```

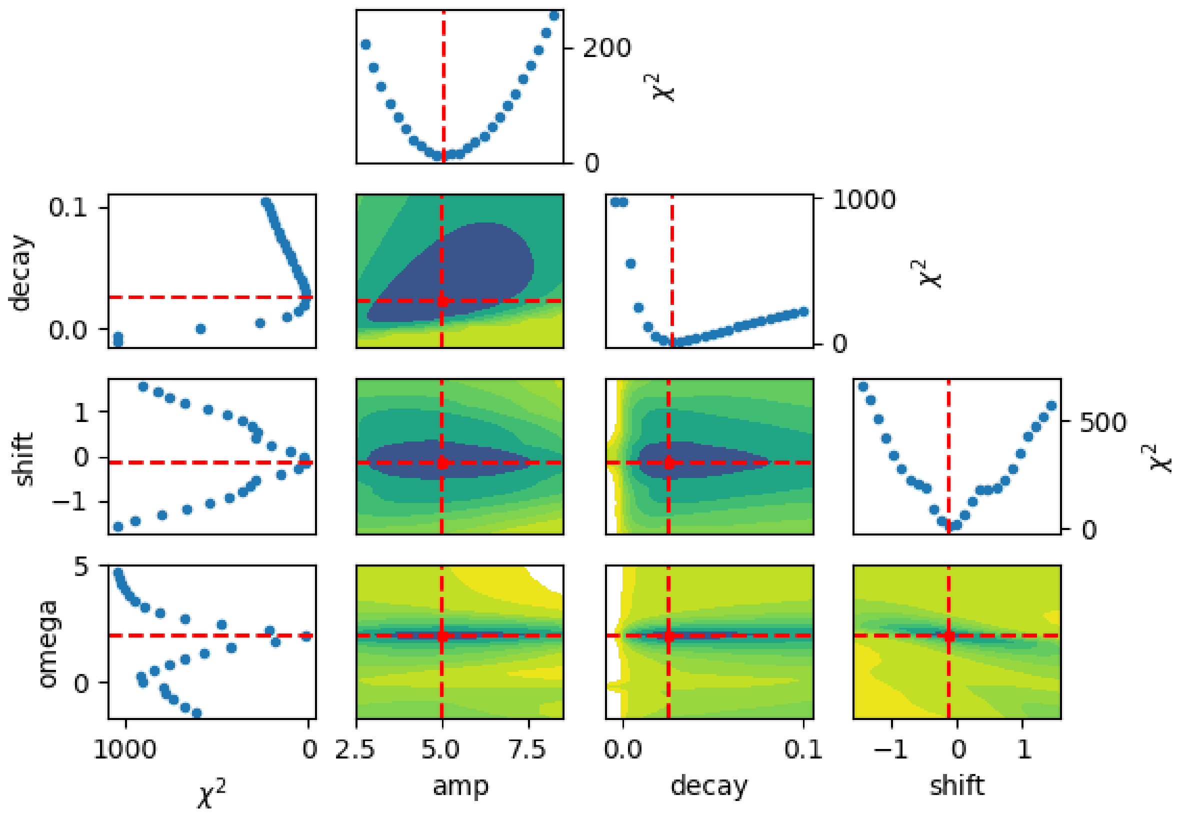
(continues on next page)

(continued from previous page)

```
                lvls1 = np.linspace(result.brute_Jout.min(),
                                    np.median(result.brute_Jout)/2.0, 7, dtype='int')
                lvls2 = np.linspace(np.median(result.brute_Jout)/2.0,
                                    np.median(result.brute_Jout), 3, dtype='int')
                lvls = np.unique(np.concatenate((lvls1, lvls2)))
                ax.contourf(X.T, Y.T, np.minimum.reduce(result.brute_Jout, axis=red_
↪axis),
                            lvls, norm=LogNorm())
                ax.set_yticks([])
                if best_vals:
                    ax.axvline(best_vals[par1].value, ls='dashed', color='r')
                    ax.axhline(best_vals[par2].value, ls='dashed', color='r')
                    ax.plot(best_vals[par1].value, best_vals[par2].value, 'rs', ms=3)
                if j != npars-1:
                    ax.set_xticks([])
                else:
                    ax.set_xlabel(varlabels[i])
                if j - i >= 2:
                    axes[i, j].axis('off')

    if output is not None:
        plt.savefig(output)
```

and finally, to generated the figure:

```
plot_results_brute(result_brute, best_vals=True, varlabels=None)
```



**Total running time of the script:** ( 1 minutes 8.961 seconds)

CHAPTER

# FOURTEEN

# EXAMPLES FROM THE DOCUMENTATION

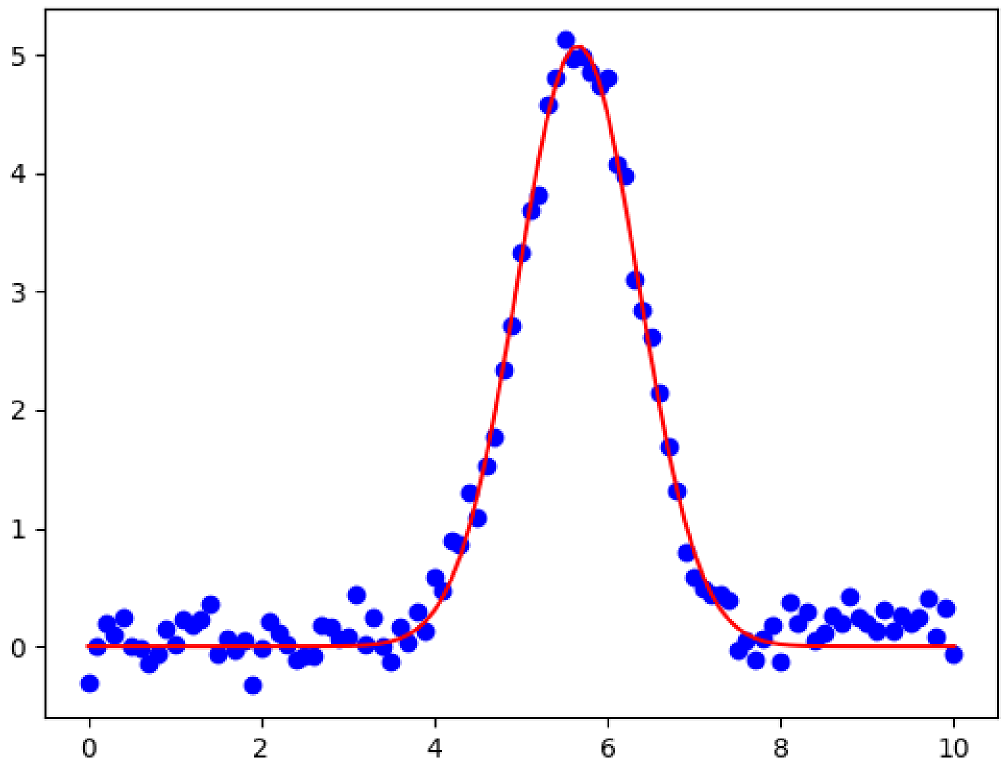
Below are all the examples that are part of the lmfit documentation.

## 14.1 doc_model_savemodel.py

```
# <examples/doc_model_savemodel.py>
import numpy as np

from lmfit.model import Model, save_model


def mysine(x, amp, freq, shift):
    return amp * np.sin(x*freq + shift)


sinemodel = Model(mysine)
pars = sinemodel.make_params(amp=1, freq=0.25, shift=0)

save_model(sinemodel, 'sinemodel.sav')
# <end examples/doc_model_savemodel.py>
```

**Total running time of the script:** ( 0 minutes 0.005 seconds)

## 14.2 doc_model_loadmodelresult.py



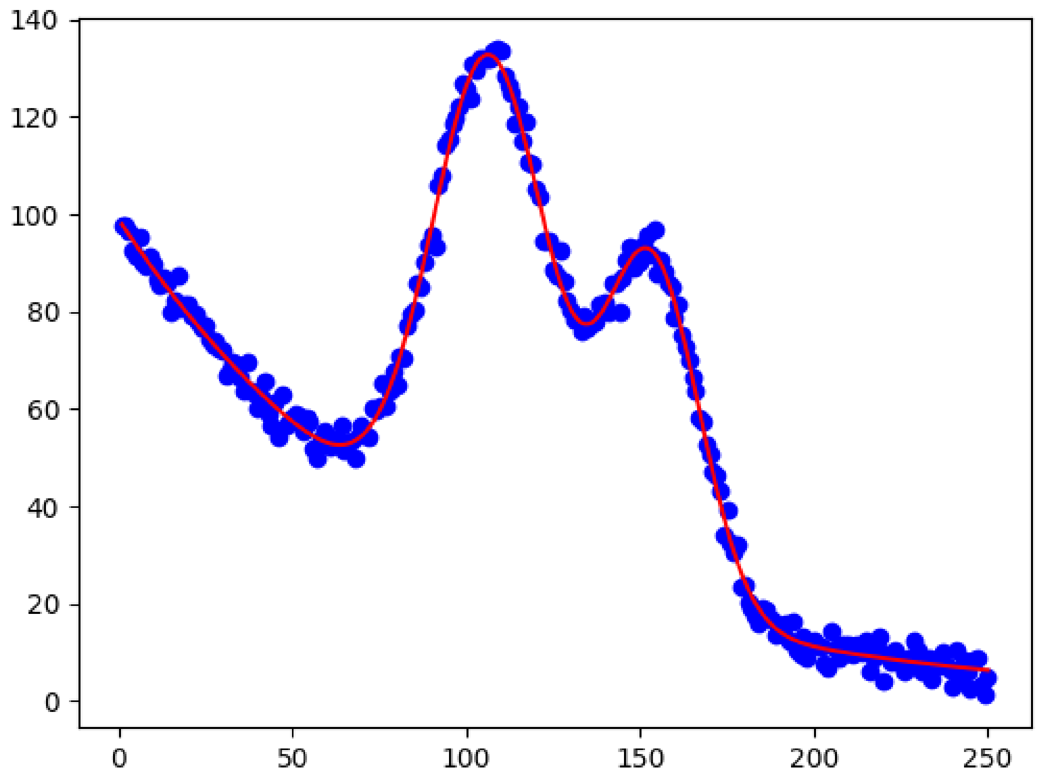
Out:

```
[[Model]]
    Model(gaussian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 29
    # data points      = 101
    # variables        = 3
    chi-square         = 3.40883599
    reduced chi-square = 0.03478404
    Akaike info crit   = -336.263713
    Bayesian info crit = -328.418352
[[Variables]]
    amplitude:  8.88022277 +/- 0.11359552 (1.28%) (init = 5)
    center:     5.65866081 +/- 0.01030506 (0.18%) (init = 5)
    sigma:      0.69765538 +/- 0.01030503 (1.48%) (init = 1)
    fwhm:       1.64285285 +/- 0.02426649 (1.48%) == '2.3548200*sigma'
    height:     5.07800352 +/- 0.06495781 (1.28%) == '0.3989423*amplitude/max(1e-15,␣
↪sigma)'
[[Correlations]] (unreported correlations are < 0.100)
    C(amplitude, sigma) =  0.577
```

```
# <examples/doc_model_loadmodelresult.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.model import load_modelresult

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

result = load_modelresult('gauss_modelresult.sav')
print(result.fit_report())

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

**Total running time of the script:** ( 0 minutes 0.241 seconds)

## 14.3 doc_model_loadmodelresult2.py



Out:

```
[[Model]]
    ((Model(gaussian, prefix='g1_') + Model(gaussian, prefix='g2_')) +
↪Model(exponential, prefix='exp_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 46
    # data points      = 250
    # variables        = 8
    chi-square         = 1247.52821
    reduced chi-square = 5.15507524
    Akaike info crit   = 417.864631
    Bayesian info crit = 446.036318
[[Variables]]
    exp_amplitude:  99.0183283 +/- 0.53748735 (0.54%) (init = 162.2102)
    exp_decay:      90.9508860 +/- 1.10310509 (1.21%) (init = 93.24905)
    g1_amplitude:   4257.77319 +/- 42.3833645 (1.00%) (init = 2000)
    g1_center:      107.030954 +/- 0.15006786 (0.14%) (init = 105)
    g1_sigma:       16.6725753 +/- 0.16048161 (0.96%) (init = 15)
    g1_fwhm:        39.2609138 +/- 0.37790530 (0.96%) == '2.3548200*g1_sigma'
    g1_height:      101.880231 +/- 0.59217099 (0.58%) == '0.3989423*g1_amplitude/
↪max(1e-15, g1_sigma)'
```

(continues on next page)

(continued from previous page)

```
    g2_amplitude:   2493.41771 +/- 36.1694731 (1.45%) (init = 2000)
    g2_center:      153.270101 +/- 0.19466743 (0.13%) (init = 155)
    g2_sigma:       13.8069484 +/- 0.18679415 (1.35%) (init = 15)
    g2_fwhm:        32.5128783 +/- 0.43986659 (1.35%) == '2.3548200*g2_sigma'
    g2_height:      72.0455934 +/- 0.61722094 (0.86%) == '0.3989423*g2_amplitude/
↪max(1e-15, g2_sigma)'
[[Correlations]] (unreported correlations are < 0.100)
    C(g1_amplitude, g1_sigma)      =  0.824
    C(g2_amplitude, g2_sigma)      =  0.815
    C(exp_amplitude, exp_decay)    = -0.695
    C(g1_sigma, g2_center)         =  0.684
    C(g1_center, g2_amplitude)     = -0.669
    C(g1_center, g2_sigma)         = -0.652
    C(g1_amplitude, g2_center)     =  0.648
    C(g1_center, g2_center)        =  0.621
    C(g1_center, g1_sigma)         =  0.507
    C(exp_decay, g1_amplitude)     = -0.507
    C(g1_sigma, g2_amplitude)      = -0.491
    C(g2_center, g2_sigma)         = -0.489
    C(g1_sigma, g2_sigma)          = -0.483
    C(g2_amplitude, g2_center)     = -0.476
    C(exp_decay, g2_amplitude)     = -0.427
    C(g1_amplitude, g1_center)     =  0.418
    C(g1_amplitude, g2_sigma)      = -0.401
    C(g1_amplitude, g2_amplitude)  = -0.307
    C(exp_amplitude, g2_amplitude) =  0.282
    C(exp_decay, g1_sigma)         = -0.252
    C(exp_decay, g2_sigma)         = -0.233
    C(exp_amplitude, g2_sigma)     =  0.171
    C(exp_decay, g2_center)        = -0.151
    C(exp_amplitude, g1_amplitude) =  0.148
    C(exp_decay, g1_center)        =  0.105
```

```python
# <examples/doc_model_loadmodelresult2.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.model import load_modelresult

dat = np.loadtxt('NIST_Gauss2.dat')
x = dat[:, 1]
y = dat[:, 0]

result = load_modelresult('nistgauss_modelresult.sav')
print(result.fit_report())

plt.plot(x, y, 'bo')
plt.plot(x, result.best_fit, 'r-')
plt.show()
# <end examples/doc_model_loadmodelresult2.py>
```

**Total running time of the script:** ( 0 minutes 0.299 seconds)

## 14.4 doc_model_savemodelresult.py

Out:

```
[[Model]]
    Model(gaussian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 29
    # data points      = 101
    # variables        = 3
    chi-square         = 3.40883599
    reduced chi-square = 0.03478404
    Akaike info crit   = -336.263713
    Bayesian info crit = -328.418352
[[Variables]]
    amplitude:  8.88022277 +/- 0.11359552 (1.28%) (init = 5)
    center:     5.65866081 +/- 0.01030506 (0.18%) (init = 5)
    sigma:      0.69765538 +/- 0.01030503 (1.48%) (init = 1)
    fwhm:       1.64285285 +/- 0.02426649 (1.48%) == '2.3548200*sigma'
    height:     5.07800352 +/- 0.06495781 (1.28%) == '0.3989423*amplitude/max(1e-15,␣
↪sigma)'
[[Correlations]] (unreported correlations are < 0.100)
    C(amplitude, sigma) =  0.577
```

```python
# <examples/doc_model_savemodelresult.py>
import numpy as np

from lmfit.model import save_modelresult
from lmfit.models import GaussianModel

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

gmodel = GaussianModel()
result = gmodel.fit(y, x=x, amplitude=5, center=5, sigma=1)

save_modelresult(result, 'gauss_modelresult.sav')

print(result.fit_report())
# <end examples/doc_model_savemodelresult.py>
```

**Total running time of the script:** ( 0 minutes 0.063 seconds)

## 14.5 doc_confidence_basic.py

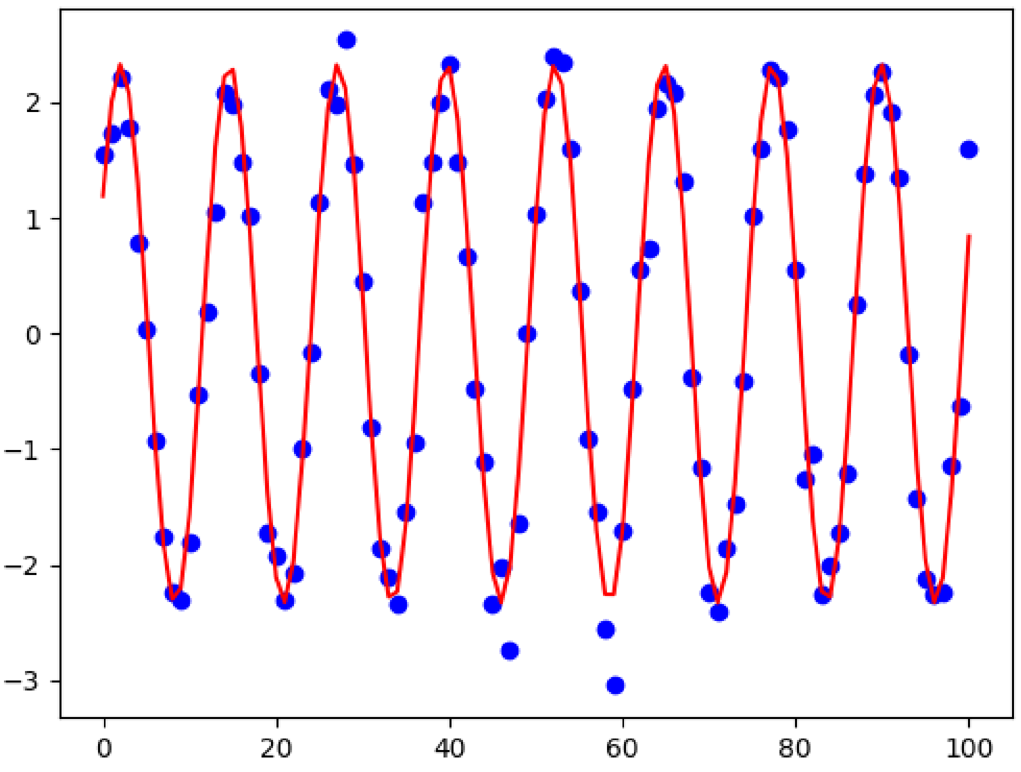
Out:

```
[[Variables]]
    a:  0.09943896 +/- 1.9322e-04 (0.19%) (init = 0.1)
    b:  1.98476942 +/- 0.01222678 (0.62%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(a, b) =  0.601
      99.73%    95.45%    68.27%    _BEST_    68.27%    95.45%    99.73%
 a:  -0.00059  -0.00039  -0.00019   0.09944  +0.00019  +0.00039  +0.00060
 b:  -0.03766  -0.02478  -0.01230   1.98477  +0.01230  +0.02478  +0.03761
```

```python
# <examples/doc_confidence_basic.py>
import numpy as np

import lmfit

x = np.linspace(0.3, 10, 100)
np.random.seed(0)
y = 1/(0.1*x) + 2 + 0.1*np.random.randn(x.size)

pars = lmfit.Parameters()
pars.add_many(('a', 0.1), ('b', 1))


def residual(p):
    return 1/(p['a']*x) + p['b'] - y


mini = lmfit.Minimizer(residual, pars)
result = mini.minimize()

print(lmfit.fit_report(result.params))

ci = lmfit.conf_interval(mini, result)
lmfit.printfuncs.report_ci(ci)
# <end examples/doc_confidence_basic.py>
```

**Total running time of the script:** ( 0 minutes 0.944 seconds)

## 14.6 doc_model_loadmodel.py



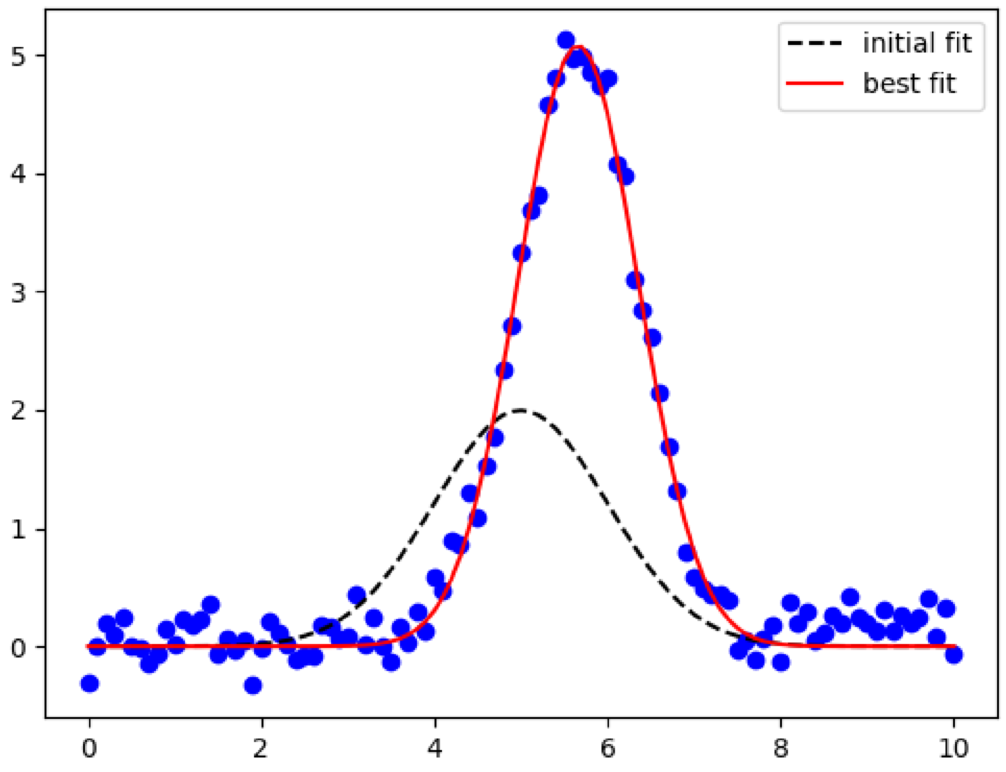
Out:

```
[[Model]]
    Model(mysine)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 21
    # data points      = 101
    # variables        = 3
    chi-square         = 7.68903767
    reduced chi-square = 0.07845957
    Akaike info crit   = -254.107813
    Bayesian info crit = -246.262452
[[Variables]]
    amp:    2.32733714 +/- 0.03950803 (1.70%) (init = 3)
    freq:   0.50098755 +/- 5.7726e-04 (0.12%) (init = 0.52)
    shift:  0.53604521 +/- 0.03383118 (6.31%) (init = 0)
[[Correlations]] (unreported correlations are < 0.100)
    C(freq, shift) = -0.866
```

```python
# <examples/doc_model_loadmodel.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.model import load_model


def mysine(x, amp, freq, shift):
    return amp * np.sin(x*freq + shift)


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

model = load_model('sinemodel.sav', funcdefs={'mysine': mysine})
params = model.make_params(amp=3, freq=0.52, shift=0)
params['shift'].max = 1
params['shift'].min = -1
params['amp'].min = 0.0

result = model.fit(y, params, x=x)
print(result.fit_report())

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

**Total running time of the script:** ( 0 minutes 0.269 seconds)

## 14.7 doc_model_gaussian.py



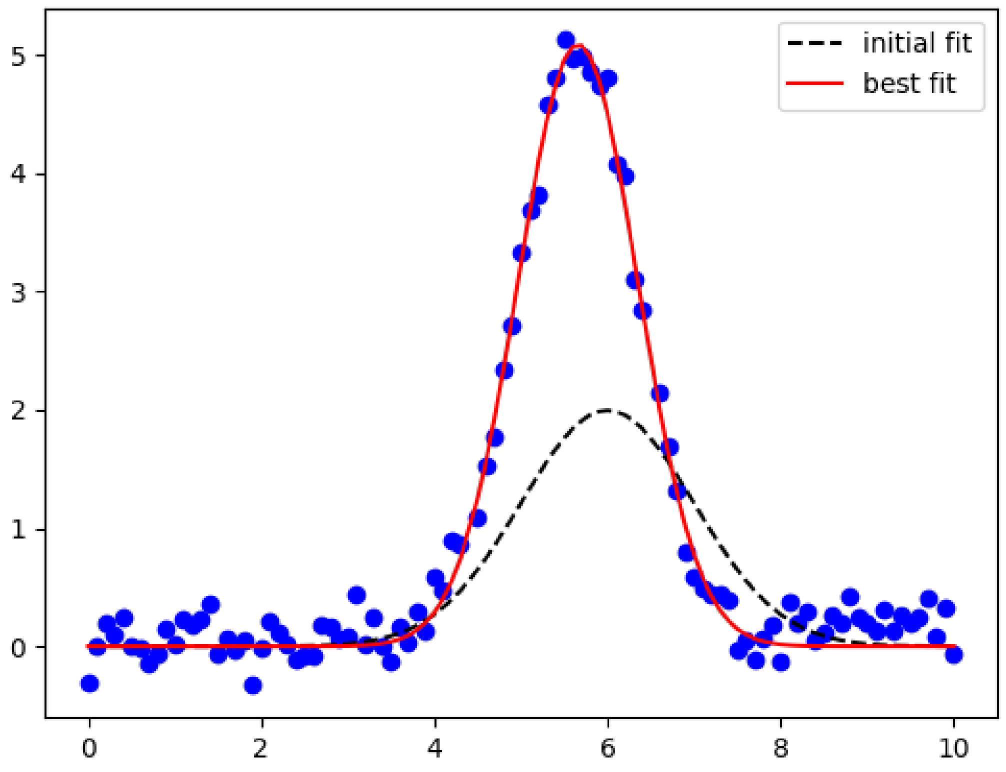
Out:

```
[[Model]]
    Model(gaussian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 33
    # data points      = 101
    # variables        = 3
    chi-square         = 3.40883599
    reduced chi-square = 0.03478404
    Akaike info crit   = -336.263713
    Bayesian info crit = -328.418352
[[Variables]]
    amp:  8.88021830 +/- 0.11359492 (1.28%) (init = 5)
    cen:  5.65866102 +/- 0.01030495 (0.18%) (init = 5)
    wid:  0.69765468 +/- 0.01030495 (1.48%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(amp, wid) =  0.577
```

```
# <examples/doc_model_gaussian.py>
import matplotlib.pyplot as plt
from numpy import exp, loadtxt, pi, sqrt

from lmfit import Model

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))


gmodel = Model(gaussian)
result = gmodel.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--', label='initial fit')
plt.plot(x, result.best_fit, 'r-', label='best fit')
plt.legend(loc='best')
plt.show()
# <end examples/doc_model_gaussian.py>
```

**Total running time of the script:** ( 0 minutes 0.262 seconds)

## 14.8 doc_model_with_nan_policy.py



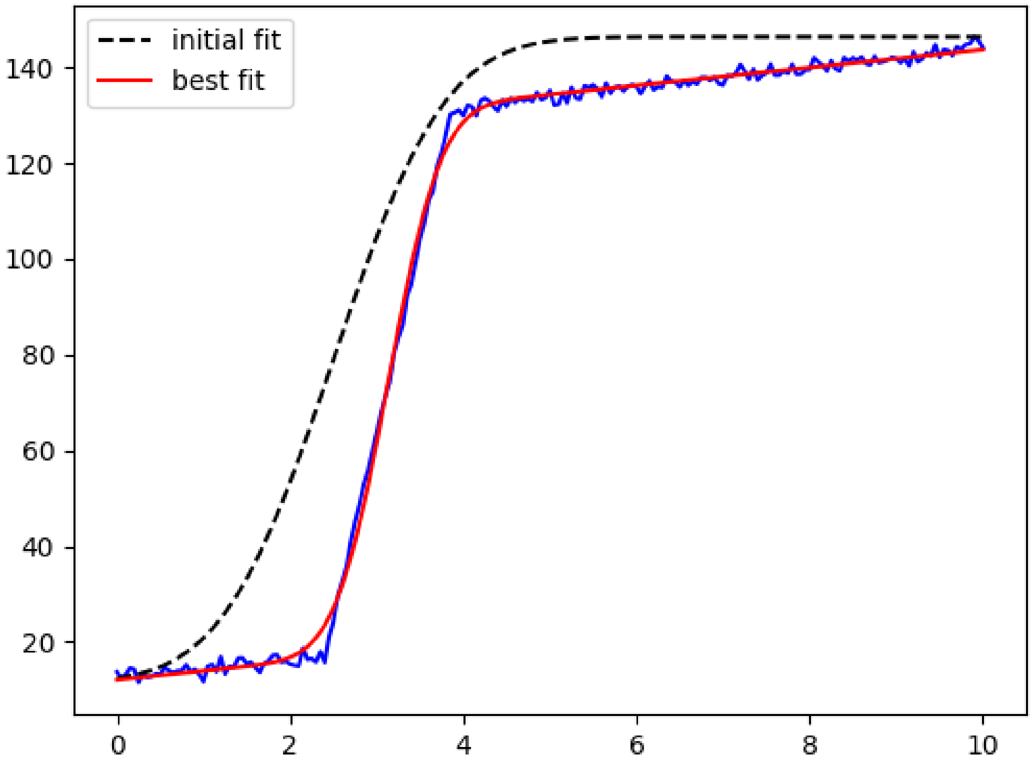
Out:

```
[[Model]]
    Model(gaussian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 22
    # data points      = 99
    # variables        = 3
    chi-square         = 3.27990355
    reduced chi-square = 0.03416566
    Akaike info crit   = -331.323278
    Bayesian info crit = -323.537918
[[Variables]]
    amplitude:  8.82064765 +/- 0.11686065 (1.32%) (init = 5)
    center:     5.65906365 +/- 0.01055590 (0.19%) (init = 6)
    sigma:      0.69165290 +/- 0.01060625 (1.53%) (init = 1)
    fwhm:       1.62871808 +/- 0.02497581 (1.53%) == '2.3548200*sigma'
    height:     5.08771012 +/- 0.06488211 (1.28%) == '0.3989423*amplitude/max(1e-15,␣
↪sigma)'
[[Correlations]] (unreported correlations are < 0.100)
    C(amplitude, sigma) =  0.610
```

```
# <examples/doc_model_with_nan_policy.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.models import GaussianModel

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

y[44] = np.nan
y[65] = np.nan

# nan_policy = 'raise'
# nan_policy = 'propagate'
nan_policy = 'omit'

gmodel = GaussianModel()
result = gmodel.fit(y, x=x, amplitude=5, center=6, sigma=1,
                    nan_policy=nan_policy)

print(result.fit_report())

# make sure nans are removed for plotting:
x_ = x[np.where(np.isfinite(y))]
y_ = y[np.where(np.isfinite(y))]

plt.plot(x_, y_, 'bo')
plt.plot(x_, result.init_fit, 'k--', label='initial fit')
plt.plot(x_, result.best_fit, 'r-', label='best fit')
plt.legend(loc='best')
plt.show()
# <end examples/doc_model_with_nan_policy.py>
```

**Total running time of the script:** ( 0 minutes 0.298 seconds)

## 14.9 doc_builtinmodels_stepmodel.py



Out:

```
[[Model]]
    (Model(step, prefix='step_', form='erf') + Model(linear, prefix='line_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 49
    # data points      = 201
    # variables        = 5
    chi-square         = 593.709622
    reduced chi-square = 3.02913072
    Akaike info crit   = 227.700173
    Bayesian info crit = 244.216698
[[Variables]]
    line_slope:       1.87164656 +/- 0.09318713 (4.98%) (init = 0)
    line_intercept:   12.0964833 +/- 0.27606235 (2.28%) (init = 11.58574)
    step_amplitude:   112.858376 +/- 0.65392947 (0.58%) (init = 134.7378)
    step_center:      3.13494792 +/- 0.00516615 (0.16%) (init = 2.5)
    step_sigma:       0.67392841 +/- 0.01091168 (1.62%) (init = 1.428571)
[[Correlations]] (unreported correlations are < 0.100)
    C(line_slope, step_amplitude)     = -0.879
    C(step_amplitude, step_sigma)     =  0.564
    C(line_slope, step_sigma)         = -0.457
```

(continues on next page)

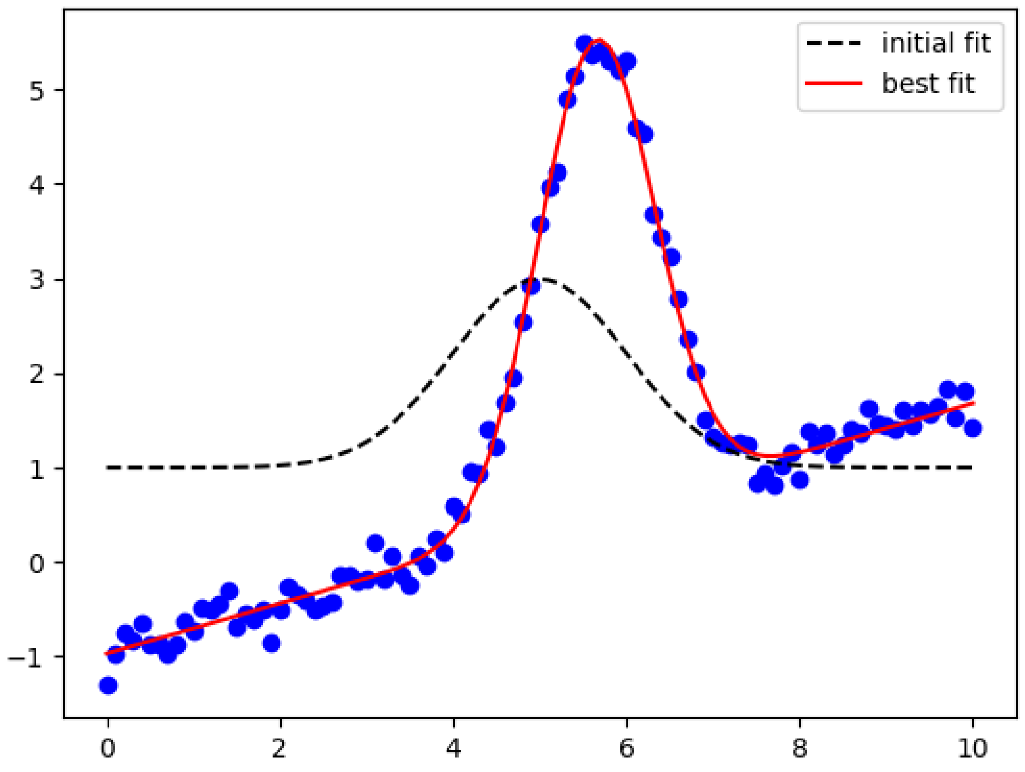
(continued from previous page)

```
    C(line_intercept, step_center)    =  0.427
    C(line_slope, line_intercept)     = -0.309
    C(line_slope, step_center)        = -0.234
    C(line_intercept, step_sigma)     = -0.137
    C(line_intercept, step_amplitude) = -0.117
    C(step_amplitude, step_center)    =  0.109
```

```
# <examples/doc_builtinmodels_stepmodel.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.models import LinearModel, StepModel

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)
np.random.seed(0)
y = 110.2 * (y + 9e-3*np.random.randn(x.size)) + 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, 'b')
plt.plot(x, out.init_fit, 'k--', label='initial fit')
plt.plot(x, out.best_fit, 'r-', label='best fit')
plt.legend(loc='best')
plt.show()
# <end examples/doc_builtinmodels_stepmodel.py>
```

**Total running time of the script:** ( 0 minutes 0.326 seconds)

## 14.10 doc_model_two_components.py



Out:

```
[[Model]]
    (Model(gaussian) + Model(line))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 44
    # data points      = 101
    # variables        = 5
    chi-square         = 2.57855517
    reduced chi-square = 0.02685995
    Akaike info crit   = -360.457020
    Bayesian info crit = -347.381417
[[Variables]]
    amp:        8.45931062 +/- 0.12414515 (1.47%) (init = 5)
    cen:        5.65547873 +/- 0.00917678 (0.16%) (init = 5)
    wid:        0.67545524 +/- 0.00991686 (1.47%) (init = 1)
    slope:      0.26484404 +/- 0.00574892 (2.17%) (init = 0)
    intercept: -0.96860202 +/- 0.03352202 (3.46%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(slope, intercept) = -0.795
    C(amp, wid)         =  0.666
    C(amp, intercept)   = -0.222
```

(continues on next page)

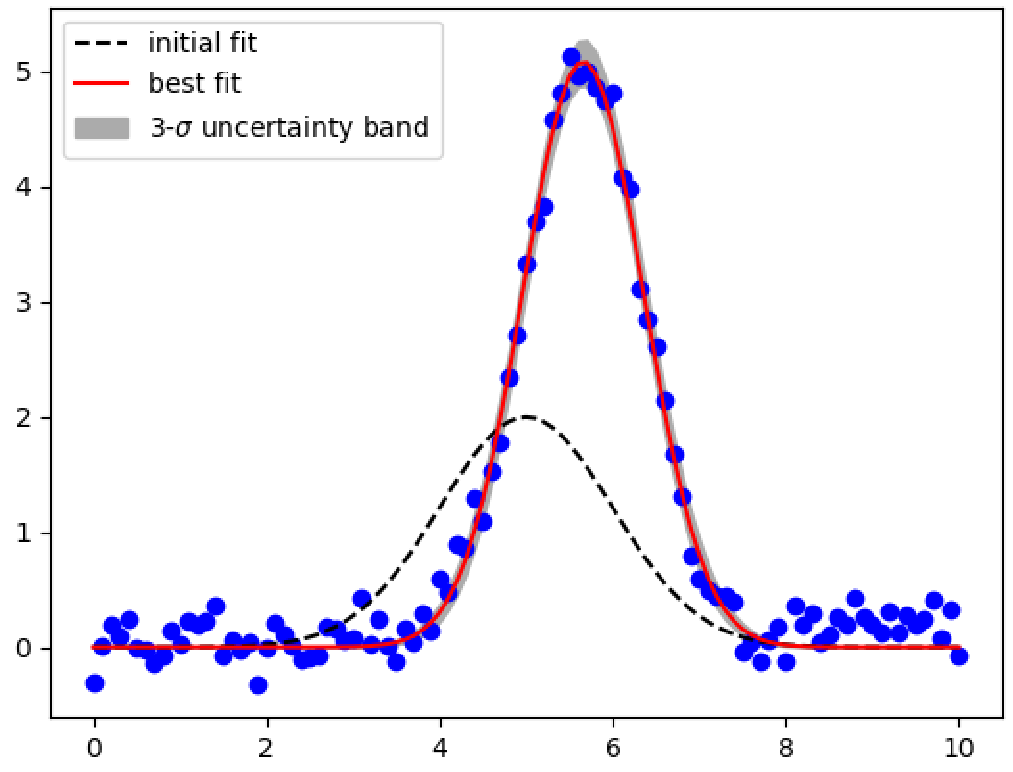
(continued from previous page)

```
    C(amp, slope)        = -0.169
    C(cen, slope)        = -0.162
    C(wid, intercept)    = -0.148
    C(cen, intercept)    =  0.129
    C(wid, slope)        = -0.113
```

```
# <examples/doc_model_two_components.py>
import matplotlib.pyplot as plt
from numpy import exp, loadtxt, pi, sqrt

from lmfit import Model

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):
    """a line"""
    return slope*x + intercept


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

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

print(result.fit_report())

plt.plot(x, y, 'bo')
plt.plot(x, result.init_fit, 'k--', label='initial fit')
plt.plot(x, result.best_fit, 'r-', label='best fit')
plt.legend(loc='best')
plt.show()
# <end examples/doc_model_two_components.py>
```

**Total running time of the script:** ( 0 minutes 0.294 seconds)

## 14.11 doc_model_uncertainty.py



Out:

```
[[Model]]
    Model(gaussian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 33
    # data points      = 101
    # variables        = 3
    chi-square         = 3.40883599
    reduced chi-square = 0.03478404
    Akaike info crit   = -336.263713
    Bayesian info crit = -328.418352
[[Variables]]
    amp:  8.88021830 +/- 0.11359492 (1.28%) (init = 5)
    cen:  5.65866102 +/- 0.01030495 (0.18%) (init = 5)
    wid:  0.69765468 +/- 0.01030495 (1.48%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(amp, wid) =  0.577
```

```
# <examples/doc_model_uncertainty.py>
import matplotlib.pyplot as plt
from numpy import exp, loadtxt, pi, sqrt

from lmfit import Model

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))


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

print(result.fit_report())

dely = result.eval_uncertainty(sigma=3)

plt.plot(x, y, 'bo')
plt.plot(x, result.init_fit, 'k--', label='initial fit')
plt.plot(x, result.best_fit, 'r-', label='best fit')
plt.fill_between(x, result.best_fit-dely, result.best_fit+dely,
                 color="#ABABAB", label=r'3-$\sigma$ uncertainty band')
plt.legend(loc='best')
plt.show()
# <end examples/doc_model_uncertainty.py>
```

**Total running time of the script:** ( 0 minutes 0.579 seconds)

## 14.12 doc_model_savemodelresult2.py

Out:

```
[[Model]]
    ((Model(gaussian, prefix='g1_') + Model(gaussian, prefix='g2_')) +
↪Model(exponential, prefix='exp_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 46
    # data points      = 250
    # variables        = 8
    chi-square         = 1247.52821
    reduced chi-square = 5.15507524
    Akaike info crit   = 417.864631
    Bayesian info crit = 446.036318
[[Variables]]
    exp_amplitude:  99.0183283 +/- 0.53748735 (0.54%) (init = 162.2102)
    exp_decay:      90.9508860 +/- 1.10310509 (1.21%) (init = 93.24905)
    g1_amplitude:   4257.77319 +/- 42.3833645 (1.00%) (init = 2000)
    g1_center:      107.030954 +/- 0.15006786 (0.14%) (init = 105)
```

(continues on next page)

(continued from previous page)

```
    g1_sigma:       16.6725753 +/- 0.16048161 (0.96%) (init = 15)
    g1_fwhm:        39.2609138 +/- 0.37790530 (0.96%) == '2.3548200*g1_sigma'
    g1_height:      101.880231 +/- 0.59217099 (0.58%) == '0.3989423*g1_amplitude/
↪max(1e-15, g1_sigma)'
    g2_amplitude:   2493.41771 +/- 36.1694731 (1.45%) (init = 2000)
    g2_center:      153.270101 +/- 0.19466743 (0.13%) (init = 155)
    g2_sigma:       13.8069484 +/- 0.18679415 (1.35%) (init = 15)
    g2_fwhm:        32.5128783 +/- 0.43986659 (1.35%) == '2.3548200*g2_sigma'
    g2_height:      72.0455934 +/- 0.61722094 (0.86%) == '0.3989423*g2_amplitude/
↪max(1e-15, g2_sigma)'
[[Correlations]] (unreported correlations are < 0.100)
    C(g1_amplitude, g1_sigma)      =  0.824
    C(g2_amplitude, g2_sigma)      =  0.815
    C(exp_amplitude, exp_decay)    = -0.695
    C(g1_sigma, g2_center)         =  0.684
    C(g1_center, g2_amplitude)     = -0.669
    C(g1_center, g2_sigma)         = -0.652
    C(g1_amplitude, g2_center)     =  0.648
    C(g1_center, g2_center)        =  0.621
    C(g1_center, g1_sigma)         =  0.507
    C(exp_decay, g1_amplitude)     = -0.507
    C(g1_sigma, g2_amplitude)      = -0.491
    C(g2_center, g2_sigma)         = -0.489
    C(g1_sigma, g2_sigma)          = -0.483
    C(g2_amplitude, g2_center)     = -0.476
    C(exp_decay, g2_amplitude)     = -0.427
    C(g1_amplitude, g1_center)     =  0.418
    C(g1_amplitude, g2_sigma)      = -0.401
    C(g1_amplitude, g2_amplitude)  = -0.307
    C(exp_amplitude, g2_amplitude) =  0.282
    C(exp_decay, g1_sigma)         = -0.252
    C(exp_decay, g2_sigma)         = -0.233
    C(exp_amplitude, g2_sigma)     =  0.171
    C(exp_decay, g2_center)        = -0.151
    C(exp_amplitude, g1_amplitude) =  0.148
    C(exp_decay, g1_center)        =  0.105
```

```python
# <examples/doc_model_savemodelresult2.py>
import numpy as np

from lmfit.model import save_modelresult
from lmfit.models import ExponentialModel, GaussianModel

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_')
```

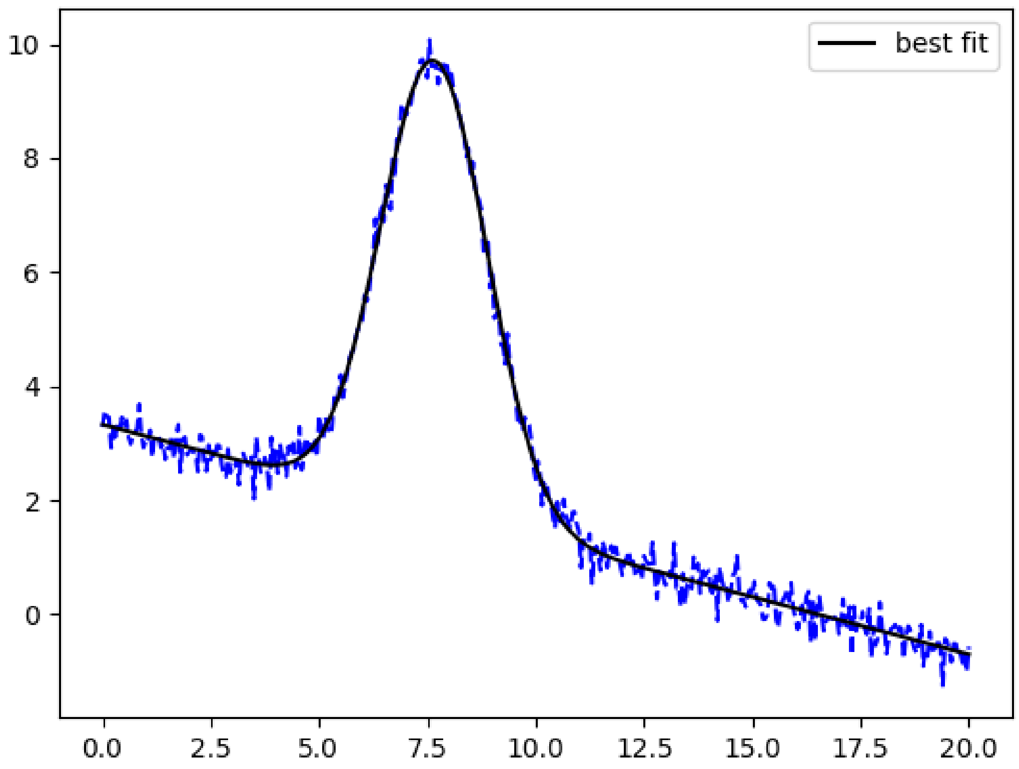
(continues on next page)

(continued from previous page)

```
pars.update(gauss1.make_params())
pars['g1_center'].set(value=105, min=75, max=125)
pars['g1_sigma'].set(value=15, min=3)
pars['g1_amplitude'].set(value=2000, min=10)

gauss2 = GaussianModel(prefix='g2_')
pars.update(gauss2.make_params())
pars['g2_center'].set(value=155, min=125, max=175)
pars['g2_sigma'].set(value=15, min=3)
pars['g2_amplitude'].set(value=2000, min=10)

mod = gauss1 + gauss2 + exp_mod

init = mod.eval(pars, x=x)

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

save_modelresult(result, 'nistgauss_modelresult.sav')

print(result.fit_report())
# <end examples/doc_model_savemodelresult2.py>
```

**Total running time of the script:** ( 0 minutes 0.222 seconds)

## 14.13 doc_model_with_iter_callback.py



Out:

```
ITER  -1 ['3.00000', '6.00000', '2.00000', '0.00000', '0.00000', '4.70964', '0.59841
↪']
ITER  0 ['3.00000', '6.00000', '2.00000', '0.00000', '0.00000', '4.70964', '0.59841']
ITER  1 ['3.00000', '6.00000', '2.00000', '0.00000', '0.00000', '4.70964', '0.59841']
ITER  2 ['3.00000', '6.00000', '2.00000', '0.00000', '0.00000', '4.70964', '0.59841']
ITER  3 ['3.00000', '6.00000', '2.00000', '0.00000', '0.00000', '4.70964', '0.59841']
ITER  4 ['3.00000', '6.00000', '2.00000', '0.00000', '0.00000', '4.70964', '0.59841']
ITER  5 ['3.00000', '6.00000', '2.00000', '0.00000', '0.00000', '4.70964', '0.59841']
ITER  6 ['3.00000', '6.00000', '2.00000', '0.00000', '0.00000', '4.70964', '0.59841']
ITER  7 ['27.41810', '24.48443', '6.86407', '-0.20617', '3.22509', '16.16365', '1.
↪59355']
ITER  8 ['27.41810', '24.48443', '6.86407', '-0.20617', '3.22509', '16.16365', '1.
↪59355']
ITER  9 ['27.41810', '24.48443', '6.86407', '-0.20617', '3.22509', '16.16365', '1.
↪59355']
ITER  10 ['27.41810', '24.48443', '6.86407', '-0.20617', '3.22509', '16.16365', '1.
↪59355']
ITER  11 ['27.41810', '24.48443', '6.86407', '-0.20617', '3.22509', '16.16365', '1.
↪59355']
ITER  12 ['27.41810', '24.48443', '6.86407', '-0.20617', '3.22509', '16.16365', '1.
↪59355']
```

(continues on next page)

(continued from previous page)

```
ITER  13 ['6.96502', '104.53101', '17.24485', '0.39526', '3.20217', '40.60853', '0.
↪16113']
ITER  14 ['25.52956', '29.38325', '12.08679', '-0.19932', '3.90951', '28.46221', '0.
↪84264']
ITER  15 ['25.52956', '29.38325', '12.08679', '-0.19932', '3.90951', '28.46221', '0.
↪84264']
ITER  16 ['25.52956', '29.38325', '12.08679', '-0.19932', '3.90951', '28.46221', '0.
↪84264']
ITER  17 ['25.52956', '29.38325', '12.08679', '-0.19932', '3.90951', '28.46221', '0.
↪84264']
ITER  18 ['25.52956', '29.38325', '12.08679', '-0.19932', '3.90951', '28.46221', '0.
↪84264']
ITER  19 ['25.52956', '29.38325', '12.08679', '-0.19932', '3.90951', '28.46221', '0.
↪84264']
ITER  20 ['20.91135', '29.80809', '28.98798', '-0.25803', '4.54769', '68.26148', '0.
↪28779']
ITER  21 ['20.91135', '29.80809', '28.98798', '-0.25803', '4.54769', '68.26148', '0.
↪28779']
ITER  22 ['20.91135', '29.80809', '28.98798', '-0.25803', '4.54769', '68.26148', '0.
↪28779']
ITER  23 ['20.91135', '29.80809', '28.98798', '-0.25803', '4.54769', '68.26148', '0.
↪28779']
ITER  24 ['20.91135', '29.80809', '28.98798', '-0.25803', '4.54769', '68.26148', '0.
↪28779']
ITER  25 ['20.91135', '29.80809', '28.98798', '-0.25803', '4.54769', '68.26148', '0.
↪28779']
ITER  26 ['29.69932', '8.72234', '28.44164', '-0.29097', '5.00931', '66.97495', '0.
↪41658']
ITER  27 ['29.69932', '8.72234', '28.44164', '-0.29097', '5.00931', '66.97495', '0.
↪41658']
ITER  28 ['29.69932', '8.72234', '28.44164', '-0.29097', '5.00931', '66.97495', '0.
↪41658']
ITER  29 ['29.69932', '8.72234', '28.44165', '-0.29097', '5.00931', '66.97496', '0.
↪41658']
ITER  30 ['29.69932', '8.72234', '28.44164', '-0.29097', '5.00931', '66.97495', '0.
↪41658']
ITER  31 ['29.69932', '8.72234', '28.44164', '-0.29097', '5.00931', '66.97495', '0.
↪41658']
ITER  32 ['46.89714', '8.78075', '7.31686', '-0.28389', '4.21023', '17.22988', '2.
↪55701']
ITER  33 ['46.89714', '8.78075', '7.31686', '-0.28389', '4.21023', '17.22988', '2.
↪55701']
ITER  34 ['46.89714', '8.78075', '7.31686', '-0.28389', '4.21023', '17.22988', '2.
↪55701']
ITER  35 ['46.89714', '8.78075', '7.31686', '-0.28389', '4.21023', '17.22988', '2.
↪55701']
ITER  36 ['46.89714', '8.78075', '7.31686', '-0.28389', '4.21023', '17.22988', '2.
↪55701']
ITER  37 ['46.89714', '8.78075', '7.31686', '-0.28389', '4.21023', '17.22988', '2.
↪55701']
ITER  38 ['-4.07966', '-5.77276', '1.83398', '0.19891', '-0.40733', '4.31869', '-0.
↪88744']
ITER  39 ['42.58021', '9.47161', '3.59328', '-0.28184', '3.21567', '8.46154', '4.
↪72744']
ITER  40 ['42.58021', '9.47161', '3.59328', '-0.28184', '3.21567', '8.46154', '4.
↪72744']
ITER  41 ['42.58021', '9.47161', '3.59328', '-0.28184', '3.21567', '8.46154', '4.
↪72744']
```

(continues on next page)

(continued from previous page)

```
ITER  42 ['42.58021', '9.47161', '3.59328', '-0.28184', '3.21567', '8.46154', '4.
↪72744']
ITER  43 ['42.58021', '9.47161', '3.59328', '-0.28184', '3.21567', '8.46154', '4.
↪72744']
ITER  44 ['42.58021', '9.47161', '3.59328', '-0.28184', '3.21567', '8.46154', '4.
↪72744']
ITER  45 ['30.01832', '6.80220', '2.99461', '-0.12851', '2.34682', '7.05177', '3.
↪99904']
ITER  46 ['30.01832', '6.80220', '2.99461', '-0.12851', '2.34682', '7.05177', '3.
↪99904']
ITER  47 ['30.01832', '6.80220', '2.99461', '-0.12851', '2.34682', '7.05177', '3.
↪99904']
ITER  48 ['30.01832', '6.80220', '2.99461', '-0.12851', '2.34682', '7.05177', '3.
↪99904']
ITER  49 ['30.01832', '6.80220', '2.99461', '-0.12851', '2.34682', '7.05177', '3.
↪99904']
ITER  50 ['30.01832', '6.80220', '2.99461', '-0.12851', '2.34682', '7.05177', '3.
↪99904']
ITER  51 ['17.69507', '8.46657', '1.01659', '-0.23045', '3.83669', '2.39389', '6.
↪94409']
ITER  52 ['17.69507', '8.46657', '1.01659', '-0.23045', '3.83669', '2.39389', '6.
↪94409']
ITER  53 ['17.69507', '8.46657', '1.01659', '-0.23045', '3.83669', '2.39389', '6.
↪94409']
ITER  54 ['17.69507', '8.46657', '1.01659', '-0.23045', '3.83669', '2.39389', '6.
↪94409']
ITER  55 ['17.69507', '8.46657', '1.01659', '-0.23045', '3.83669', '2.39389', '6.
↪94409']
ITER  56 ['17.69507', '8.46657', '1.01659', '-0.23045', '3.83669', '2.39389', '6.
↪94409']
ITER  57 ['22.61349', '7.75137', '1.45137', '-0.21630', '3.56563', '3.41771', '6.
↪21585']
ITER  58 ['22.61349', '7.75137', '1.45137', '-0.21630', '3.56563', '3.41771', '6.
↪21585']
ITER  59 ['22.61349', '7.75137', '1.45137', '-0.21630', '3.56563', '3.41771', '6.
↪21585']
ITER  60 ['22.61349', '7.75137', '1.45137', '-0.21630', '3.56563', '3.41771', '6.
↪21585']
ITER  61 ['22.61349', '7.75137', '1.45137', '-0.21630', '3.56563', '3.41771', '6.
↪21585']
ITER  62 ['22.61349', '7.75137', '1.45137', '-0.21630', '3.56563', '3.41771', '6.
↪21585']
ITER  63 ['24.13916', '7.62179', '1.18261', '-0.20160', '3.34246', '2.78482', '8.
↪14315']
ITER  64 ['24.13916', '7.62179', '1.18261', '-0.20160', '3.34246', '2.78482', '8.
↪14315']
ITER  65 ['24.13916', '7.62179', '1.18261', '-0.20160', '3.34246', '2.78482', '8.
↪14315']
ITER  66 ['24.13916', '7.62179', '1.18260', '-0.20160', '3.34246', '2.78482', '8.
↪14315']
ITER  67 ['24.13916', '7.62179', '1.18261', '-0.20160', '3.34246', '2.78482', '8.
↪14315']
ITER  68 ['24.13916', '7.62179', '1.18261', '-0.20160', '3.34246', '2.78482', '8.
↪14315']
ITER  69 ['24.40272', '7.65357', '1.22777', '-0.20151', '3.32847', '2.89118', '7.
↪92923']
ITER  70 ['24.40272', '7.65357', '1.22777', '-0.20151', '3.32847', '2.89118', '7.
↪92923']
```

(continues on next page)

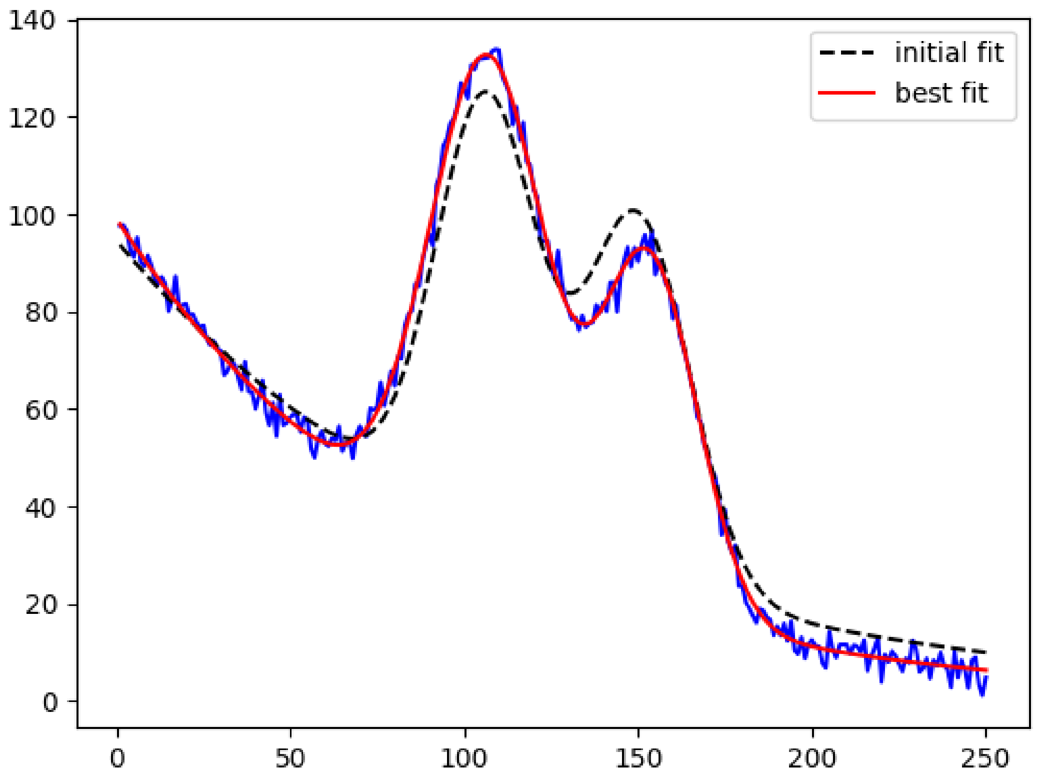
(continued from previous page)

```
 ITER  71 ['24.40272', '7.65357', '1.22777', '-0.20151', '3.32847', '2.89118', '7.
↪92923']
 ITER  72 ['24.40272', '7.65357', '1.22777', '-0.20151', '3.32847', '2.89118', '7.
↪92923']
 ITER  73 ['24.40272', '7.65357', '1.22777', '-0.20151', '3.32847', '2.89118', '7.
↪92923']
 ITER  74 ['24.40272', '7.65357', '1.22777', '-0.20151', '3.32847', '2.89118', '7.
↪92923']
 ITER  75 ['24.43754', '7.65514', '1.22977', '-0.20146', '3.32621', '2.89589', '7.
↪92763']
 ITER  76 ['24.43754', '7.65514', '1.22977', '-0.20146', '3.32621', '2.89589', '7.
↪92763']
 ITER  77 ['24.43754', '7.65514', '1.22977', '-0.20146', '3.32621', '2.89589', '7.
↪92763']
 ITER  78 ['24.43754', '7.65514', '1.22977', '-0.20146', '3.32621', '2.89589', '7.
↪92763']
 ITER  79 ['24.43754', '7.65514', '1.22977', '-0.20146', '3.32621', '2.89589', '7.
↪92763']
 ITER  80 ['24.43754', '7.65514', '1.22977', '-0.20146', '3.32621', '2.89589', '7.
↪92763']
 ITER  81 ['24.43823', '7.65514', '1.22982', '-0.20146', '3.32615', '2.89600', '7.
↪92754']
 ITER  81 ['24.43823', '7.65514', '1.22982', '-0.20146', '3.32615', '2.89600', '7.
↪92754']
Nfev =  81
[[Model]]
    (Model(gaussian, prefix='peak_') + Model(linear, prefix='bkg_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 81
    # data points      = 401
    # variables        = 5
    chi-square         = 20.7309419
    reduced chi-square = 0.05235086
    Akaike info crit   = -1177.89596
    Bayesian info crit = -1157.92615
[[Variables]]
    peak_amplitude:  24.4382287 +/- 0.16625391 (0.68%) (init = 3)
    peak_center:     7.65513988 +/- 0.00768465 (0.10%) (init = 6)
    peak_sigma:      1.22981974 +/- 0.00834506 (0.68%) (init = 2)
    bkg_slope:      -0.20145591 +/- 0.00207980 (1.03%) (init = 0)
    bkg_intercept:   3.32614932 +/- 0.02701936 (0.81%) (init = 0)
    peak_fwhm:       2.89600411 +/- 0.01965112 (0.68%) == '2.3548200*peak_sigma'
    peak_height:     7.92753837 +/- 0.04386563 (0.55%) == '0.3989423*peak_amplitude/
↪max(1e-15, peak_sigma)'
[[Correlations]] (unreported correlations are < 0.100)
    C(bkg_slope, bkg_intercept)      = -0.857
    C(peak_amplitude, peak_sigma)    =  0.668
    C(peak_amplitude, bkg_intercept) = -0.526
    C(peak_sigma, bkg_intercept)     = -0.352
    C(peak_amplitude, bkg_slope)     =  0.285
    C(peak_sigma, bkg_slope)         =  0.190
    C(peak_center, bkg_slope)        = -0.146
    C(peak_center, bkg_intercept)    =  0.125
```

```
# <examples/doc_with_itercb.py>
import matplotlib.pyplot as plt
from numpy import linspace, random

from lmfit.lineshapes import gaussian
from lmfit.models import GaussianModel, LinearModel


def per_iteration(pars, iteration, resid, *args, **kws):
    print(" ITER ", iteration, ["%.5f" % p for p in pars.values()])


x = linspace(0., 20, 401)
y = gaussian(x, amplitude=24.56, center=7.6543, sigma=1.23)
y = y - .20*x + 3.333 + random.normal(scale=0.23, size=x.size)

mod = GaussianModel(prefix='peak_') + LinearModel(prefix='bkg_')

pars = mod.make_params()
pars['peak_amplitude'].value = 3.0
pars['peak_center'].value = 6.0
pars['peak_sigma'].value = 2.0
pars['bkg_intercept'].value = 0.0
pars['bkg_slope'].value = 0.0

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

plt.plot(x, y, 'b--')

print('Nfev = ', out.nfev)
print(out.fit_report())

plt.plot(x, out.best_fit, 'k-', label='best fit')
plt.legend(loc='best')
plt.show()
# <end examples/doc_with_itercb.py>
```

**Total running time of the script:** ( 0 minutes 0.500 seconds)

## 14.14 doc_builtinmodels_nistgauss2.py



Out:

```
[[Model]]
    ((Model(gaussian, prefix='g1_') + Model(gaussian, prefix='g2_')) +␣
↪Model(exponential, prefix='exp_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 37
    # data points      = 250
    # variables        = 8
    chi-square         = 1247.52821
    reduced chi-square = 5.15507524
    Akaike info crit   = 417.864631
    Bayesian info crit = 446.036318
[[Variables]]
    exp_amplitude:  99.0183270 +/- 0.53748905 (0.54%) (init = 94.53724)
    exp_decay:      90.9508890 +/- 1.10310483 (1.21%) (init = 111.1985)
    g1_amplitude:   4257.77343 +/- 42.3836433 (1.00%) (init = 3189.648)
    g1_center:      107.030956 +/- 0.15006873 (0.14%) (init = 106.5)
    g1_sigma:       16.6725765 +/- 0.16048227 (0.96%) (init = 14.5)
    g1_fwhm:        39.2609166 +/- 0.37790686 (0.96%) == '2.3548200*g1_sigma'
    g1_height:      101.880230 +/- 0.59217233 (0.58%) == '0.3989423*g1_amplitude/
↪max(1e-15, g1_sigma)'
```

(continues on next page)

(continued from previous page)

```
    g2_amplitude:   2493.41733 +/- 36.1696906 (1.45%) (init = 2818.337)
    g2_center:      153.270101 +/- 0.19466905 (0.13%) (init = 150)
    g2_sigma:       13.8069461 +/- 0.18679534 (1.35%) (init = 15)
    g2_fwhm:        32.5128728 +/- 0.43986939 (1.35%) == '2.3548200*g2_sigma'
    g2_height:      72.0455948 +/- 0.61722328 (0.86%) == '0.3989423*g2_amplitude/
↪max(1e-15, g2_sigma)'
[[Correlations]] (unreported correlations are < 0.500)
    C(g1_amplitude, g1_sigma)    =  0.824
    C(g2_amplitude, g2_sigma)    =  0.815
    C(exp_amplitude, exp_decay) = -0.695
    C(g1_sigma, g2_center)       =  0.684
    C(g1_center, g2_amplitude)   = -0.669
    C(g1_center, g2_sigma)       = -0.652
    C(g1_amplitude, g2_center)   =  0.648
    C(g1_center, g2_center)      =  0.621
    C(g1_center, g1_sigma)       =  0.507
    C(exp_decay, g1_amplitude)   = -0.507
```

```python
# <examples/doc_nistgauss2.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.models import ExponentialModel, GaussianModel

dat = np.loadtxt('NIST_Gauss2.dat')
x = dat[:, 1]
y = dat[:, 0]

exp_mod = ExponentialModel(prefix='exp_')
gauss1 = GaussianModel(prefix='g1_')
gauss2 = GaussianModel(prefix='g2_')


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)

pars1 = exp_mod.guess(y[:ix1], x=x[:ix1])
pars2 = gauss1.guess(y[ix1:ix2], x=x[ix1:ix2])
pars3 = gauss2.guess(y[ix2:ix3], x=x[ix2:ix3])

pars = pars1 + pars2 + pars3
mod = gauss1 + gauss2 + exp_mod
```

(continues on next page)

(continued from previous page)

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

print(out.fit_report(min_correl=0.5))

plt.plot(x, y, 'b')
plt.plot(x, out.init_fit, 'k--', label='initial fit')
plt.plot(x, out.best_fit, 'r-', label='best fit')
plt.legend(loc='best')
plt.show()
# <end examples/doc_nistgauss2.py>
```

**Total running time of the script:** ( 0 minutes 0.401 seconds)

## 14.15 doc_fitting_withreport.py

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 83
    # data points      = 1001
    # variables        = 4
    chi-square         = 498.811759
    reduced chi-square = 0.50031270
    Akaike info crit   = -689.222517
    Bayesian info crit = -669.587497
[[Variables]]
    amp:     13.9121945 +/- 0.14120288 (1.01%) (init = 13)
    period:  5.48507045 +/- 0.02666492 (0.49%) (init = 2)
    shift:   0.16203677 +/- 0.01405661 (8.67%) (init = 0)
    decay:   0.03264538 +/- 3.8014e-04 (1.16%) (init = 0.02)
[[Correlations]] (unreported correlations are < 0.100)
    C(period, shift) =  0.797
    C(amp, decay)    =  0.582
    C(amp, shift)    = -0.297
    C(amp, period)   = -0.243
    C(shift, decay)  = -0.182
    C(period, decay) = -0.150
```

```
# <examples/doc_fitting_withreport.py>
from numpy import exp, linspace, pi, random, sign, sin

from lmfit import Parameters, fit_report, minimize

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)
```

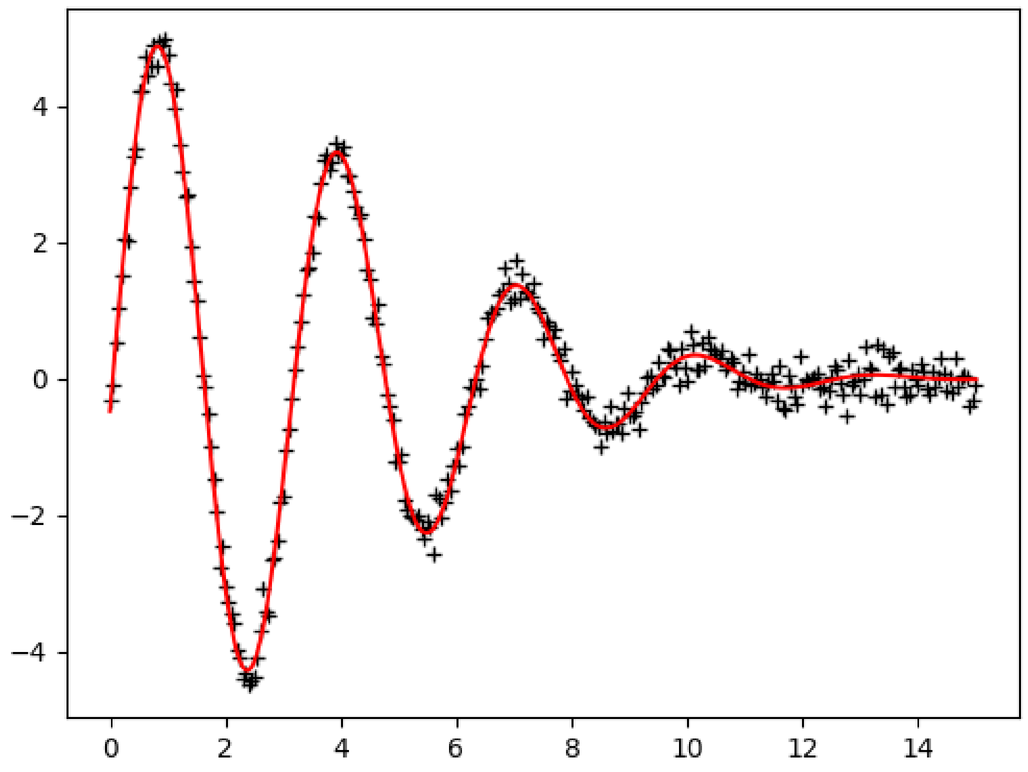
(continues on next page)

(continued from previous page)

```


def residual(pars, x, data=None):
    """Model a decaying sine wave and subtract data."""
    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


random.seed(0)
x = linspace(0.0, 250., 1001)
noise = random.normal(scale=0.7215, size=x.size)
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})

print(fit_report(out))
# <end examples/doc_fitting_withreport.py>
```

**Total running time of the script:** ( 0 minutes 0.029 seconds)

## 14.16 doc_parameters_valuesdict.py



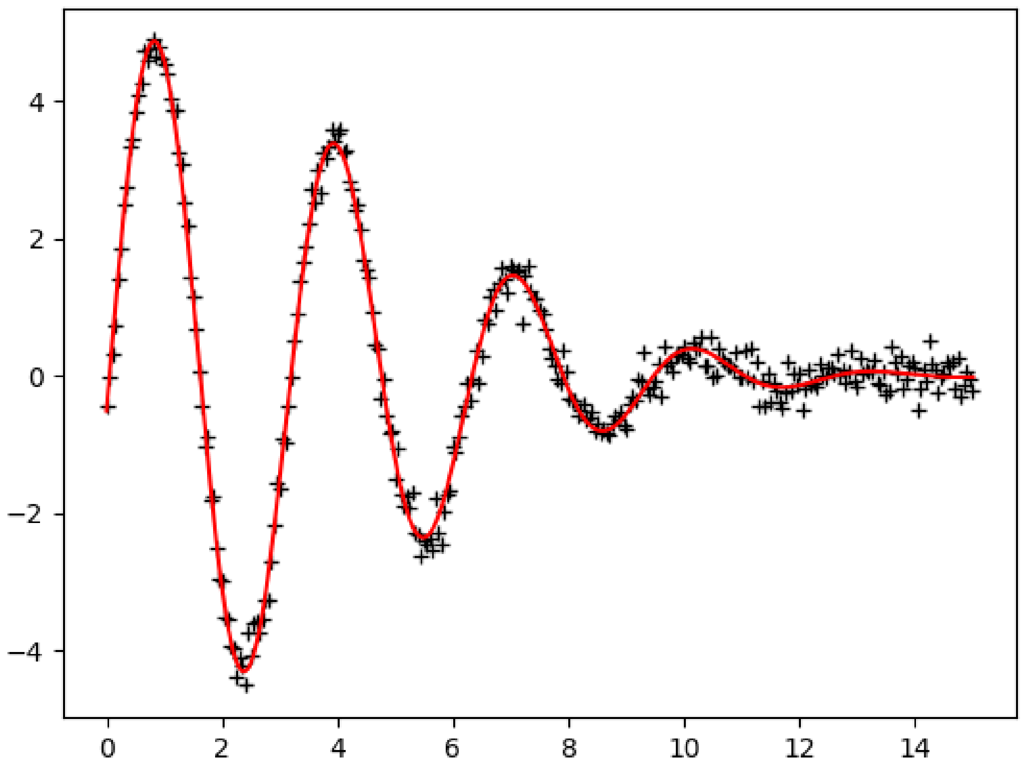
Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 63
    # data points      = 301
    # variables        = 4
    chi-square         = 10.9764657
    reduced chi-square = 0.03695780
    Akaike info crit   = -988.718387
    Bayesian info crit = -973.889945
[[Variables]]
    amp:    4.96174550 +/- 0.03830181 (0.77%) (init = 10)
    decay:  0.02571887 +/- 4.5357e-04 (1.76%) (init = 0.1)
    shift: -0.09714733 +/- 0.00991436 (10.21%) (init = 0)
    omega:  1.99750219 +/- 0.00321063 (0.16%) (init = 3)
[[Correlations]] (unreported correlations are < 0.100)
    C(shift, omega) = -0.785
    C(amp, decay)   =  0.584
    C(amp, shift)   = -0.120
```

```
# <examples/doc_parameters_valuesdict.py>
import numpy as np

from lmfit import Minimizer, Parameters, report_fit

# create data to be fitted
x = np.linspace(0, 15, 301)
data = (5.0 * np.sin(2.0*x - 0.1) * np.exp(-x*x*0.025) +
        np.random.normal(size=x.size, scale=0.2))


# define objective function: returns the array to be minimized
def fcn2min(params, x, data):
    """Model a decaying sine wave and subtract data."""
    v = params.valuesdict()

    model = v['amp'] * np.sin(x * v['omega'] + v['shift']) * np.exp(-x*x*v['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 the default leastsq algorithm
minner = Minimizer(fcn2min, params, fcn_args=(x, data))
result = minner.minimize()

# calculate final result
final = data + result.residual

# write error report
report_fit(result)

# try to plot results
try:
    import matplotlib.pyplot as plt
    plt.plot(x, data, 'k+')
    plt.plot(x, final, 'r')
    plt.show()
except ImportError:
    pass
# <end of examples/doc_parameters_valuesdict.py>
```

**Total running time of the script:** ( 0 minutes 0.303 seconds)

## 14.17 doc_parameters_basic.py



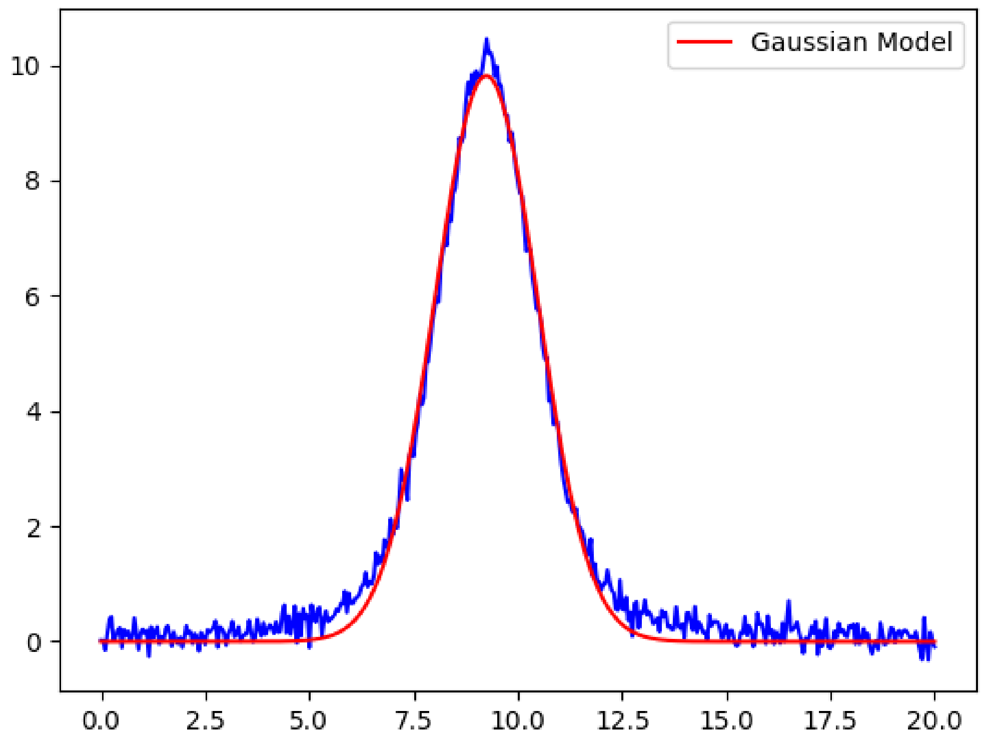
Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 64
    # data points      = 301
    # variables        = 4
    chi-square         = 11.0484618
    reduced chi-square = 0.03720021
    Akaike info crit   = -986.750534
    Bayesian info crit = -971.922093
[[Variables]]
    amp:    4.96078856 +/- 0.03791272 (0.76%) (init = 10)
    decay:  0.02437789 +/- 4.2572e-04 (1.75%) (init = 0.1)
    shift: -0.10363212 +/- 0.00981677 (9.47%) (init = 0)
    omega:  2.00019266 +/- 0.00309578 (0.15%) (init = 3)
[[Correlations]] (unreported correlations are < 0.100)
    C(shift, omega) = -0.785
    C(amp, decay)   =  0.584
    C(amp, shift)   = -0.117
```

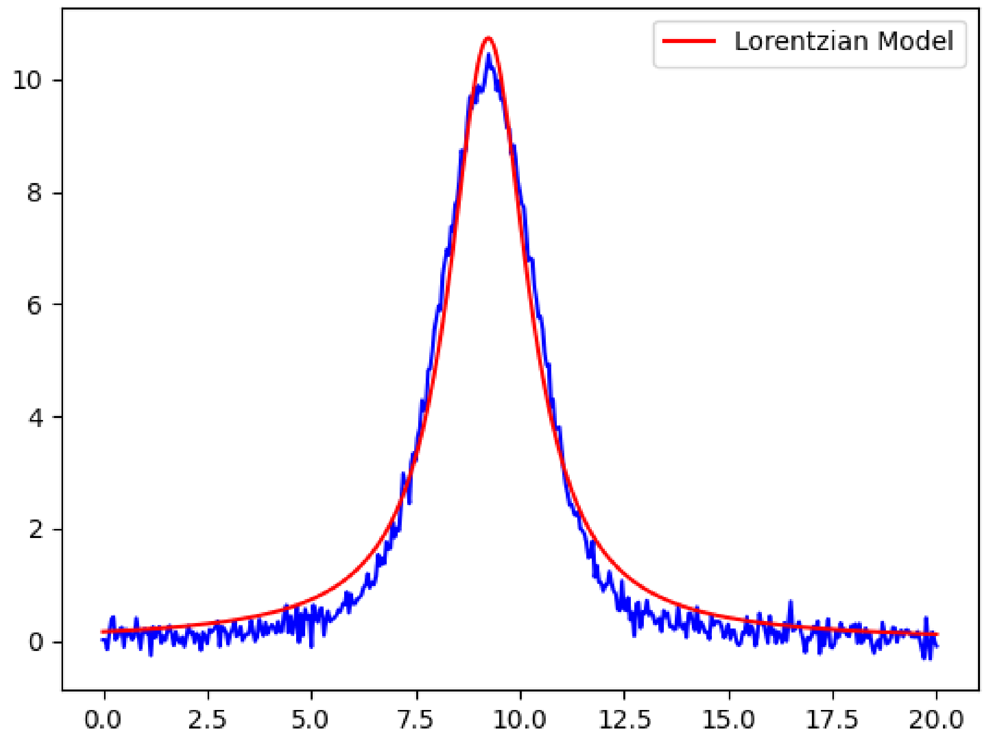
```
# <examples/doc_parameters_basic.py>
import numpy as np

from lmfit import Minimizer, Parameters, report_fit

# create data to be fitted
x = np.linspace(0, 15, 301)
data = (5.0 * np.sin(2.0*x - 0.1) * np.exp(-x*x*0.025) +
        np.random.normal(size=x.size, scale=0.2))


# define objective function: returns the array to be minimized
def fcn2min(params, x, data):
    """Model a decaying sine wave and subtract data."""
    amp = params['amp']
    shift = params['shift']
    omega = params['omega']
    decay = params['decay']
    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 the default leastsq algorithm
minner = Minimizer(fcn2min, params, fcn_args=(x, data))
result = minner.minimize()

# calculate final result
final = data + result.residual

# write error report
report_fit(result)

# try to plot results
try:
    import matplotlib.pyplot as plt
    plt.plot(x, data, 'k+')
    plt.plot(x, final, 'r')
    plt.show()
except ImportError:
    pass
# <end of examples/doc_parameters_basic.py>
```

**Total running time of the script:** ( 0 minutes 0.233 seconds)

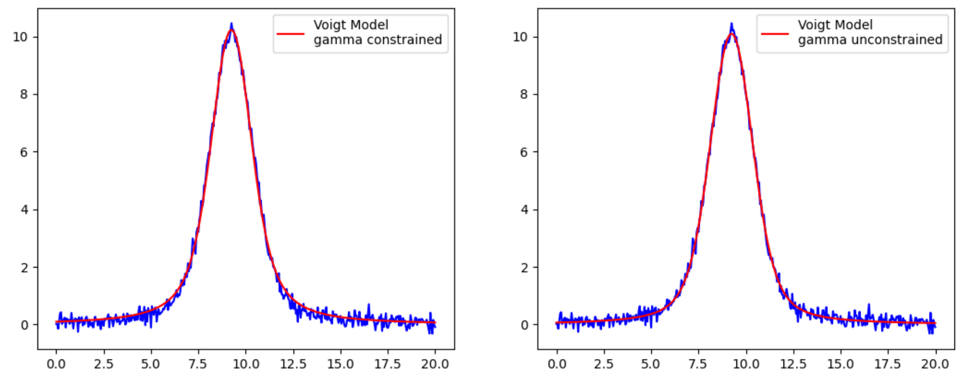
## 14.18 doc_builtinmodels_peakmodels.py



-



•



•

Out:

```
[[Model]]
    Model(gaussian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 25
    # data points      = 401
    # variables        = 3
    chi-square         = 29.9943157
    reduced chi-square = 0.07536260
```

(continues on next page)

(continued from previous page)

```
    Akaike info crit   = -1033.77437
    Bayesian info crit = -1021.79248
[[Variables]]
    amplitude:  30.3135620 +/- 0.15712686 (0.52%) (init = 43.62238)
    center:     9.24277047 +/- 0.00737496 (0.08%) (init = 9.25)
    sigma:      1.23218359 +/- 0.00737496 (0.60%) (init = 1.35)
    fwhm:       2.90157056 +/- 0.01736670 (0.60%) == '2.3548200*sigma'
    height:     9.81457817 +/- 0.05087283 (0.52%) == '0.3989423*amplitude/max(1e-15,␣
↪sigma)'
[[Correlations]] (unreported correlations are < 0.250)
    C(amplitude, sigma) =  0.577
[[Model]]
    Model(lorentzian)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 21
    # data points      = 401
    # variables        = 3
    chi-square         = 53.7535387
    reduced chi-square = 0.13505914
    Akaike info crit   = -799.830322
    Bayesian info crit = -787.848438
[[Variables]]
    amplitude:  38.9727645 +/- 0.31386183 (0.81%) (init = 54.52798)
    center:     9.24438944 +/- 0.00927619 (0.10%) (init = 9.25)
    sigma:      1.15483925 +/- 0.01315659 (1.14%) (init = 1.35)
    fwhm:       2.30967850 +/- 0.02631318 (1.14%) == '2.0000000*sigma'
    height:     10.7421156 +/- 0.08633945 (0.80%) == '0.3183099*amplitude/max(1e-15,␣
↪sigma)'
[[Correlations]] (unreported correlations are < 0.250)
    C(amplitude, sigma) =  0.709
[[Model]]
    Model(voigt)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 21
    # data points      = 401
    # variables        = 3
    chi-square         = 14.5448627
    reduced chi-square = 0.03654488
    Akaike info crit   = -1324.00615
    Bayesian info crit = -1312.02427
[[Variables]]
    amplitude:  35.7554146 +/- 0.13861321 (0.39%) (init = 65.43358)
    center:     9.24411150 +/- 0.00505482 (0.05%) (init = 9.25)
    sigma:      0.73015627 +/- 0.00368460 (0.50%) (init = 0.8775)
    gamma:      0.73015627 +/- 0.00368460 (0.50%) == 'sigma'
    fwhm:       2.62950494 +/- 0.00806900 (0.31%) == '1.0692*gamma+sqrt(0.
↪8664*gamma**2+5.545083*sigma**2)'
    height:     10.2203969 +/- 0.03009415 (0.29%) == '(amplitude/(max(1e-15,␣
↪sigma*sqrt(2*pi))))*wofz((1j*gamma)/(max(1e-15, sigma*sqrt(2)))).real'
[[Correlations]] (unreported correlations are < 0.250)
    C(amplitude, sigma) =  0.651
```

```
# <examples/doc_builtinmodels_peakmodels.py>
import matplotlib.pyplot as plt
from numpy import loadtxt

from lmfit.models import GaussianModel, LorentzianModel, VoigtModel

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


# Gaussian model
mod = GaussianModel()
pars = mod.guess(y, x=x)
out = mod.fit(y, pars, x=x)

print(out.fit_report(min_correl=0.25))

plt.plot(x, y, 'b-')
plt.plot(x, out.best_fit, 'r-', label='Gaussian Model')
plt.legend(loc='best')
plt.show()


# Lorentzian model
mod = LorentzianModel()
pars = mod.guess(y, x=x)
out = mod.fit(y, pars, x=x)

print(out.fit_report(min_correl=0.25))

plt.figure()
plt.plot(x, y, 'b-')
plt.plot(x, out.best_fit, 'r-', label='Lorentzian Model')
plt.legend(loc='best')
plt.show()


# Voigt model
mod = VoigtModel()
pars = mod.guess(y, x=x)
out = mod.fit(y, pars, x=x)

print(out.fit_report(min_correl=0.25))

fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))

axes[0].plot(x, y, 'b-')
axes[0].plot(x, out.best_fit, 'r-', label='Voigt Model\ngamma constrained')
axes[0].legend(loc='best')

# free gamma parameter
pars['gamma'].set(value=0.7, vary=True, expr='')
out_gamma = mod.fit(y, pars, x=x)
axes[1].plot(x, y, 'b-')
axes[1].plot(x, out_gamma.best_fit, 'r-', label='Voigt Model\ngamma unconstrained')
axes[1].legend(loc='best')
```

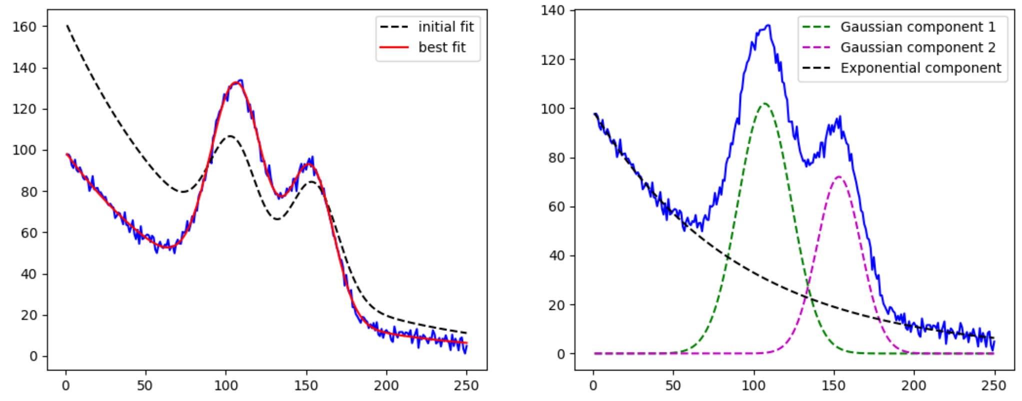
(continues on next page)

(continued from previous page)

```
plt.show()
# <end examples/doc_builtinmodels_peakmodels.py>
```

**Total running time of the script:** ( 0 minutes 1.150 seconds)

## 14.19 doc_builtinmodels_nistgauss.py



Out:

```
[[Model]]
    ((Model(gaussian, prefix='g1_') + Model(gaussian, prefix='g2_')) +␣
↪Model(exponential, prefix='exp_'))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 46
    # data points      = 250
    # variables        = 8
    chi-square         = 1247.52821
    reduced chi-square = 5.15507524
    Akaike info crit   = 417.864631
    Bayesian info crit = 446.036318
[[Variables]]
    exp_amplitude:  99.0183283 +/- 0.53748735 (0.54%) (init = 162.2102)
    exp_decay:      90.9508860 +/- 1.10310509 (1.21%) (init = 93.24905)
    g1_amplitude:   4257.77319 +/- 42.3833645 (1.00%) (init = 2000)
    g1_center:      107.030954 +/- 0.15006786 (0.14%) (init = 105)
    g1_sigma:       16.6725753 +/- 0.16048161 (0.96%) (init = 15)
    g1_fwhm:        39.2609138 +/- 0.37790530 (0.96%) == '2.3548200*g1_sigma'
    g1_height:      101.880231 +/- 0.59217099 (0.58%) == '0.3989423*g1_amplitude/
↪max(1e-15, g1_sigma)'
    g2_amplitude:   2493.41771 +/- 36.1694731 (1.45%) (init = 2000)
    g2_center:      153.270101 +/- 0.19466743 (0.13%) (init = 155)
    g2_sigma:       13.8069484 +/- 0.18679415 (1.35%) (init = 15)
    g2_fwhm:        32.5128783 +/- 0.43986659 (1.35%) == '2.3548200*g2_sigma'
    g2_height:      72.0455934 +/- 0.61722094 (0.86%) == '0.3989423*g2_amplitude/
↪max(1e-15, g2_sigma)'
```

(continues on next page)

(continued from previous page)

```
[[Correlations]] (unreported correlations are < 0.500)
    C(g1_amplitude, g1_sigma)    =  0.824
    C(g2_amplitude, g2_sigma)    =  0.815
    C(exp_amplitude, exp_decay) = -0.695
    C(g1_sigma, g2_center)       =  0.684
    C(g1_center, g2_amplitude)  = -0.669
    C(g1_center, g2_sigma)       = -0.652
    C(g1_amplitude, g2_center)  =  0.648
    C(g1_center, g2_center)      =  0.621
    C(g1_center, g1_sigma)       =  0.507
    C(exp_decay, g1_amplitude)  = -0.507
```

```
# <examples/doc_builtinmodels_nistgauss.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.models import ExponentialModel, GaussianModel

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

gauss2 = GaussianModel(prefix='g2_')
pars.update(gauss2.make_params())

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

mod = gauss1 + gauss2 + exp_mod

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

print(out.fit_report(min_correl=0.5))

fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))
axes[0].plot(x, y, 'b')
axes[0].plot(x, init, 'k--', label='initial fit')
axes[0].plot(x, out.best_fit, 'r-', label='best fit')
axes[0].legend(loc='best')
```

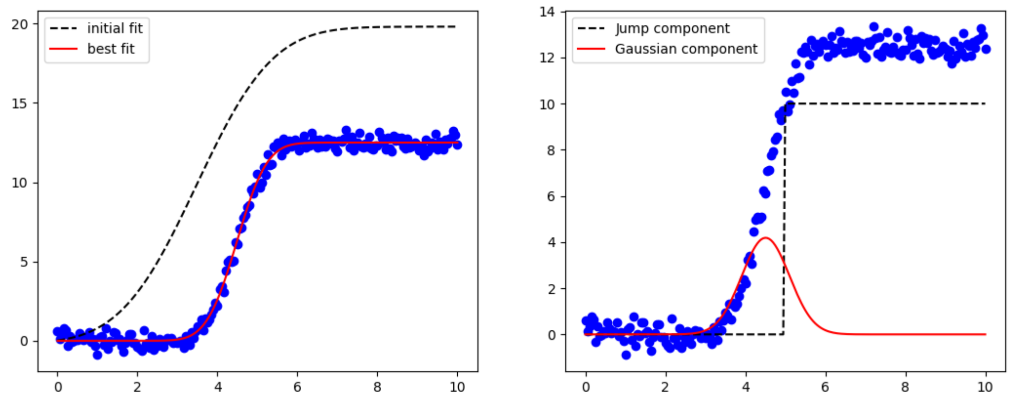
(continues on next page)

(continued from previous page)

```
comps = out.eval_components(x=x)
axes[1].plot(x, y, 'b')
axes[1].plot(x, comps['g1_'], 'g--', label='Gaussian component 1')
axes[1].plot(x, comps['g2_'], 'm--', label='Gaussian component 2')
axes[1].plot(x, comps['exp_'], 'k--', label='Exponential component')
axes[1].legend(loc='best')

plt.show()
# <end examples/doc_builtinmodels_nistgauss.py>
```

**Total running time of the script:** ( 0 minutes 0.649 seconds)

## 14.20 doc_model_composite.py



Out:

```
[[Model]]
    (Model(jump) <function convolve at 0x1243149d0> Model(gaussian))
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 25
    # data points      = 201
    # variables        = 3
    chi-square         = 24.7562335
    reduced chi-square = 0.12503148
    Akaike info crit   = -414.939746
    Bayesian info crit = -405.029832
[[Variables]]
    mid:         5 (fixed)
    amplitude:  0.62508459 +/- 0.00189732 (0.30%) (init = 1)
    center:     4.50853671 +/- 0.00973231 (0.22%) (init = 3.5)
    sigma:      0.59576118 +/- 0.01348582 (2.26%) (init = 1.5)
[[Correlations]] (unreported correlations are < 0.100)
    C(amplitude, center) =  0.329
    C(amplitude, sigma)  =  0.268
```

```python
# <examples/doc_model_composite.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit import CompositeModel, Model
from lmfit.lineshapes import gaussian, step

# create data from broadened step
x = np.linspace(0, 10, 201)
y = step(x, amplitude=12.5, center=4.5, sigma=0.88, form='erf')
np.random.seed(0)
y = y + np.random.normal(scale=0.35, size=x.size)


def jump(x, mid):
    """Heaviside step function."""
    o = np.zeros(x.size)
    imid = max(np.where(x <= mid)[0])
    o[imid:] = 1.0
    return o


def convolve(arr, kernel):
    """Simple convolution of two arrays."""
    npts = min(arr.size, kernel.size)
    pad = np.ones(npts)
    tmp = np.concatenate((pad*arr[0], arr, pad*arr[-1]))
    out = np.convolve(tmp, kernel, mode='valid')
    noff = int((len(out) - npts) / 2)
    return out[noff:noff+npts]


# create Composite Model using the custom convolution operator
mod = CompositeModel(Model(jump), Model(gaussian), convolve)
pars = mod.make_params(amplitude=1, center=3.5, sigma=1.5, mid=5.0)

# 'mid' and 'center' should be completely correlated, and 'mid' is
# used as an integer index, so a very poor fit variable:
pars['mid'].vary = False

# fit this model to data array y
result = mod.fit(y, params=pars, x=x)

print(result.fit_report())

# generate components
comps = result.eval_components(x=x)

# plot results
fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))

axes[0].plot(x, y, 'bo')
axes[0].plot(x, result.init_fit, 'k--', label='initial fit')
axes[0].plot(x, result.best_fit, 'r-', label='best fit')
```

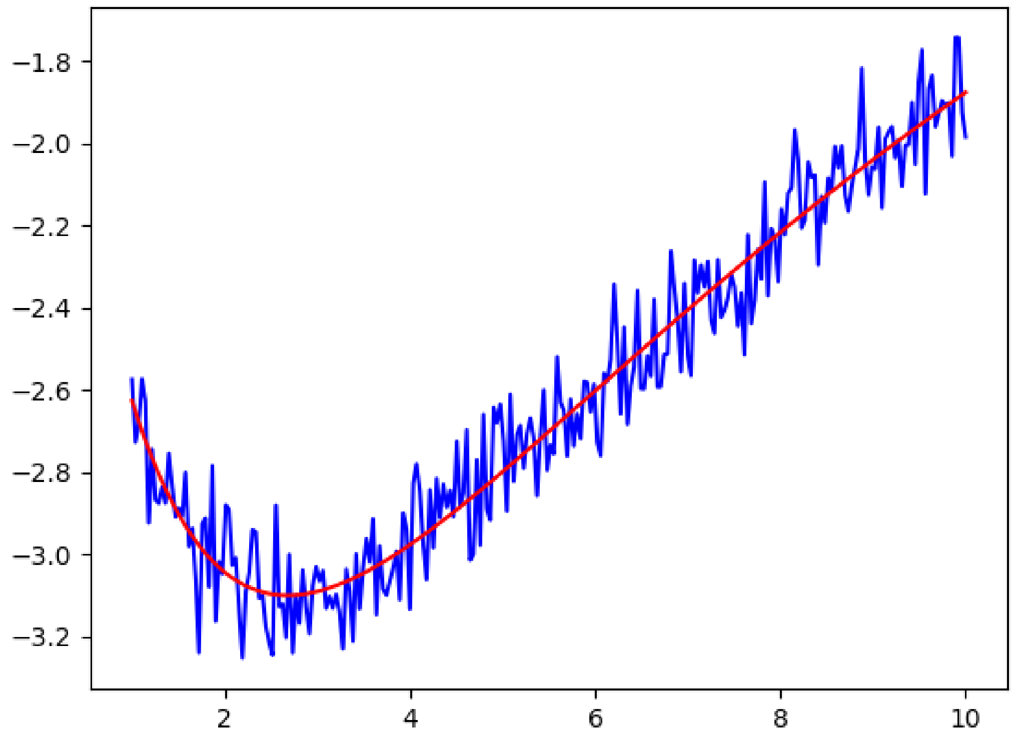
(continues on next page)

(continued from previous page)

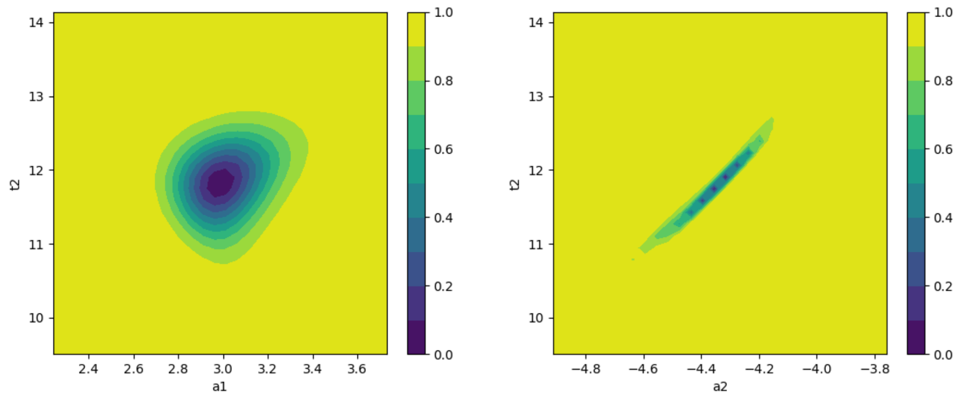
```
axes[0].legend(loc='best')

axes[1].plot(x, y, 'bo')
axes[1].plot(x, 10*comps['jump'], 'k--', label='Jump component')
axes[1].plot(x, 10*comps['gaussian'], 'r-', label='Gaussian component')
axes[1].legend(loc='best')

plt.show()
# <end examples/doc_model_composite.py>
```

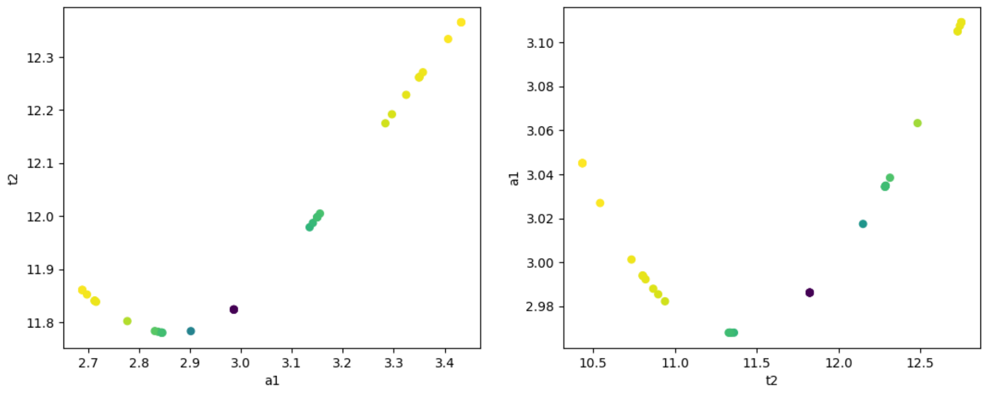
**Total running time of the script:** ( 0 minutes 0.497 seconds)

## 14.21 doc_confidence_advanced.py



•

- 

- 

Out:

```
[[Variables]]
    a1:  2.98622120 +/- 0.14867187 (4.98%) (init = 2.986237)
    a2: -4.33526327 +/- 0.11527506 (2.66%) (init = -4.335256)
    t1:  1.30994233 +/- 0.13121177 (10.02%) (init = 1.309932)
    t2:  11.8240351 +/- 0.46316470 (3.92%) (init = 11.82408)
[[Correlations]] (unreported correlations are < 0.500)
    C(a2, t2) =  0.987
    C(a2, t1) = -0.925
    C(t1, t2) = -0.881
    C(a1, t1) = -0.599
       95.45%    68.27%    _BEST_    68.27%    95.45%
 a1:  -0.27286  -0.14165   2.98622  +0.16353  +0.36343
 a2:  -0.30444  -0.13219  -4.33526  +0.10688  +0.19683
 t1:  -0.23392  -0.12494   1.30994  +0.14660  +0.32369
 t2:  -1.01943  -0.48820  11.82404  +0.46041  +0.90441
```

```
# <examples/doc_confidence_advanced.py>
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import lmfit

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(x.size)

p = lmfit.Parameters()
p.add_many(('a1', 4.), ('a2', 4.), ('t1', 3.), ('t2', 3.))


def residual(p):
    return p['a1']*np.exp(-x/p['t1']) + p['a2']*np.exp(-(x-0.1)/p['t2']) - y


# create Minimizer
mini = lmfit.Minimizer(residual, p, nan_policy='propagate')

# first solve with Nelder-Mead algorithm
out1 = mini.minimize(method='Nelder')

# then solve with Levenberg-Marquardt using the
# Nelder-Mead solution as a starting point
out2 = mini.minimize(method='leastsq', params=out1.params)

lmfit.report_fit(out2.params, min_correl=0.5)

ci, trace = lmfit.conf_interval(mini, out2, sigmas=[1, 2], trace=True)
lmfit.printfuncs.report_ci(ci)

# plot data and best fit
plt.figure()
plt.plot(x, y, 'b')
plt.plot(x, residual(out2.params) + y, 'r-')

# plot confidence intervals (a1 vs t2 and a2 vs t2)
fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))
cx, cy, grid = lmfit.conf_interval2d(mini, out2, 'a1', 't2', 30, 30)
ctp = axes[0].contourf(cx, cy, grid, np.linspace(0, 1, 11))
fig.colorbar(ctp, ax=axes[0])
axes[0].set_xlabel('a1')
axes[0].set_ylabel('t2')

cx, cy, grid = lmfit.conf_interval2d(mini, out2, 'a2', 't2', 30, 30)
ctp = axes[1].contourf(cx, cy, grid, np.linspace(0, 1, 11))
fig.colorbar(ctp, ax=axes[1])
axes[1].set_xlabel('a2')
axes[1].set_ylabel('t2')

# plot dependence between two parameters
fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))
cx1, cy1, prob = trace['a1']['a1'], trace['a1']['t2'], trace['a1']['prob']
cx2, cy2, prob2 = trace['t2']['t2'], trace['t2']['a1'], trace['t2']['prob']

axes[0].scatter(cx1, cy1, c=prob, s=30)
axes[0].set_xlabel('a1')
axes[0].set_ylabel('t2')
```

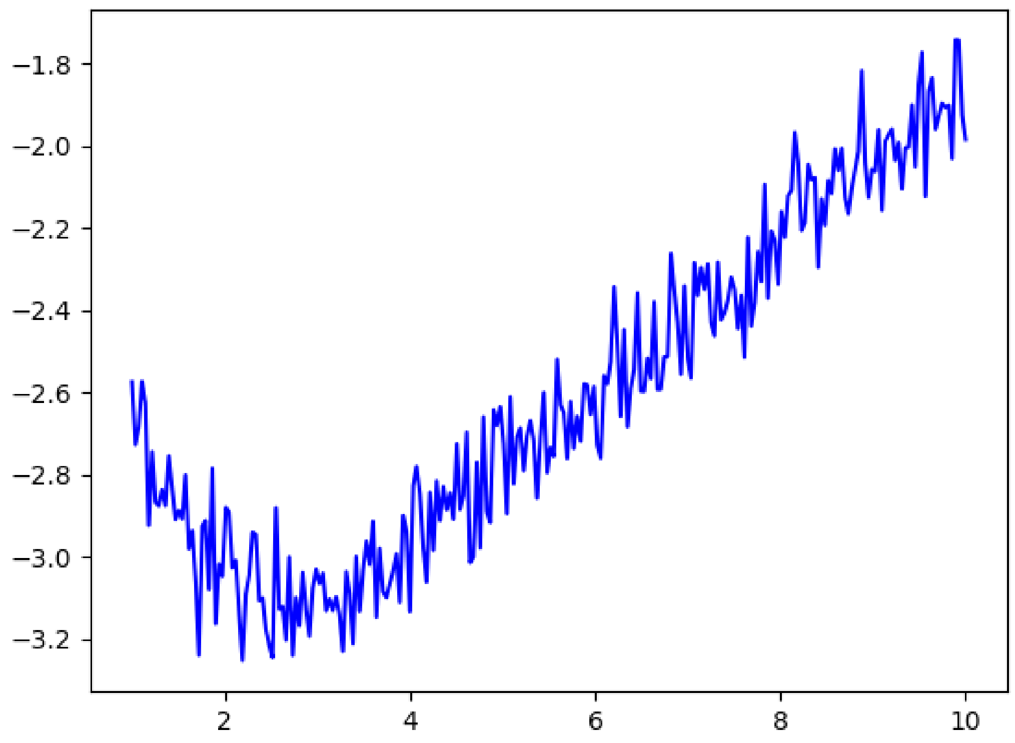
(continues on next page)

(continued from previous page)

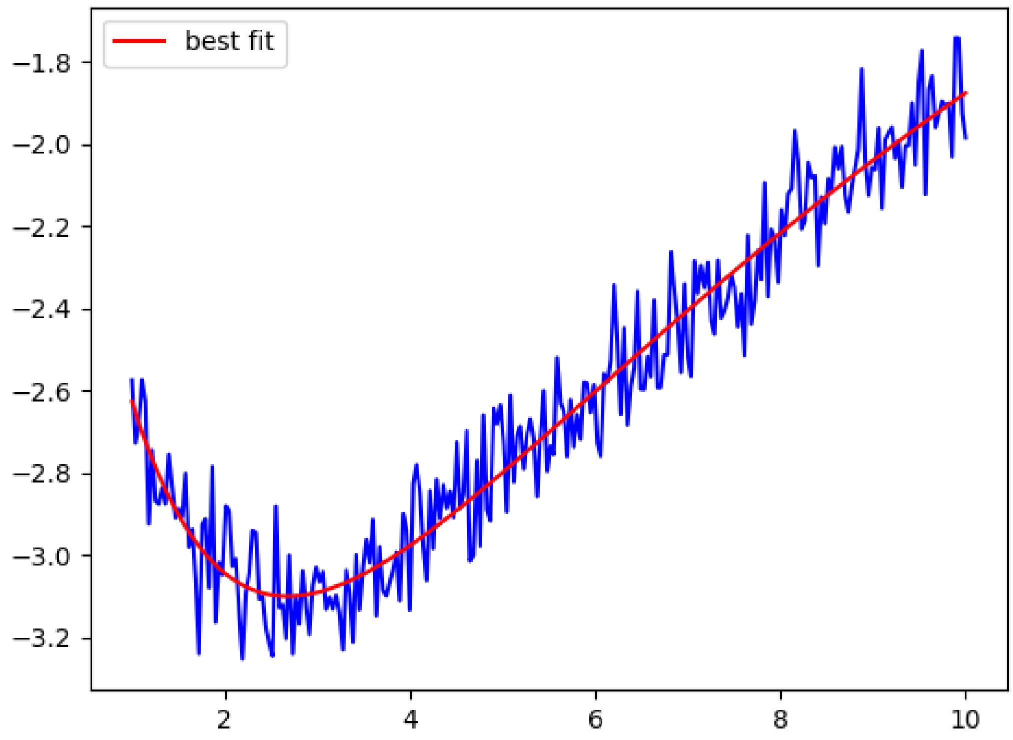
```
axes[1].scatter(cx2, cy2, c=prob2, s=30)
axes[1].set_xlabel('t2')
axes[1].set_ylabel('a1')

plt.show()
# <end examples/doc_confidence_advanced.py>
```

**Total running time of the script:** ( 0 minutes 12.095 seconds)

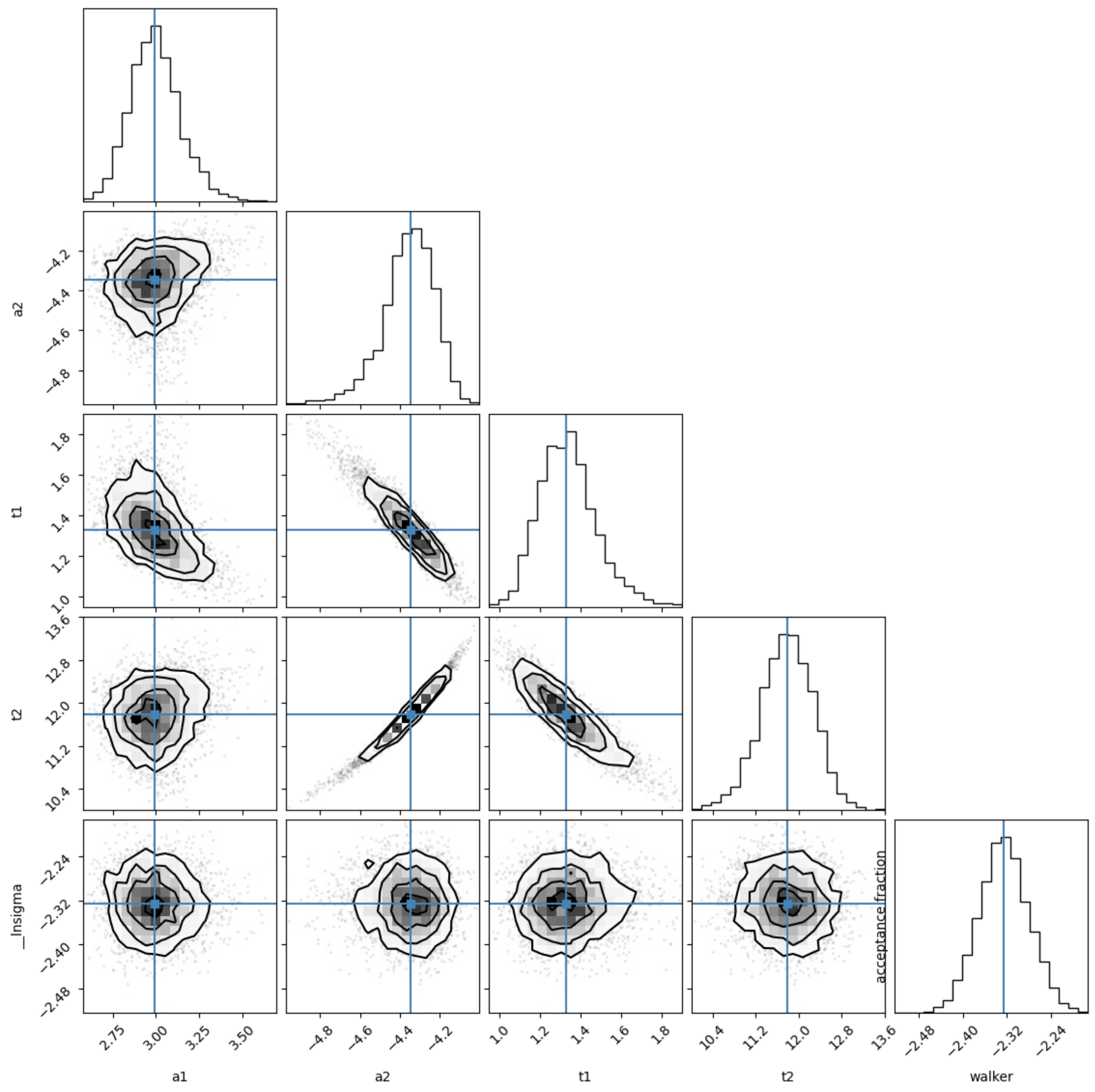
## 14.22 doc_fitting_emcee.py



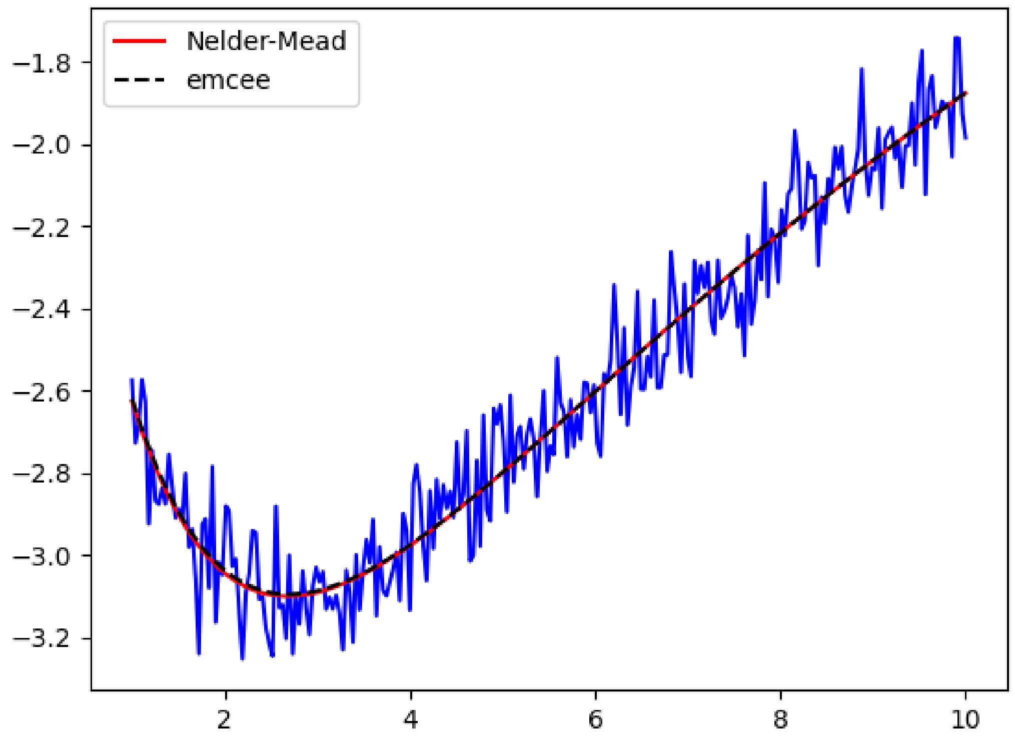
•


best fit
−1.8
−2.0
−2.2
−2.4
−2.6
−2.8
−3.0
−3.2
2
4
6
8
10

•


a2
t1
t2
__lnsigma
acceptance fraction
a1
a2
t1
t2
walker

•



•

Out:

```
[[Variables]]
    a1:  2.98623689 +/- 0.15010519 (5.03%) (init = 4)
    a2: -4.33525597 +/- 0.11765824 (2.71%) (init = 4)
    t1:  1.30993186 +/- 0.13449656 (10.27%) (init = 3)
    t2:  11.8240752 +/- 0.47172610 (3.99%) (init = 3)
[[Correlations]] (unreported correlations are < 0.500)
    C(a2, t2) =  0.988
    C(a2, t1) = -0.928
    C(t1, t2) = -0.885
    C(a1, t1) = -0.609
The chain is shorter than 50 times the integrated autocorrelation time for 5␣
↪parameter(s). Use this estimate with caution and run a longer chain!
N/50 = 20;
tau: [42.15955322 47.347426   48.71211873 46.7985718  40.89881208]

median of posterior probability distribution
--------------------------------------------
[[Variables]]
    a1:         2.98945718 +/- 0.14033921 (4.69%) (init = 2.986237)
    a2:        -4.34687243 +/- 0.12131092 (2.79%) (init = -4.335256)
    t1:         1.32883916 +/- 0.13766047 (10.36%) (init = 1.309932)
    t2:         11.7836194 +/- 0.47719763 (4.05%) (init = 11.82408)
    __lnsigma: -2.32559226 +/- 0.04542650 (1.95%) (init = -2.302585)
[[Correlations]] (unreported correlations are < 0.100)
    C(a2, t2) =  0.981
```

(continues on next page)

(continued from previous page)

```
    C(a2, t1) = -0.938
    C(t1, t2) = -0.894
    C(a1, t1) = -0.508
    C(a1, a2) =  0.214
    C(a1, t2) =  0.178

Maximum Likelihood Estimation from emcee
-------------------------------------------------
Parameter  MLE Value   Median Value   Uncertainty
  a1         2.93839     2.98946       0.14034
  a2        -4.35274    -4.34687       0.12131
  t1         1.34310     1.32884       0.13766
  t2        11.78782    11.78362       0.47720

Error Estimates from emcee
------------------------------------------------------
Parameter  -2sigma  -1sigma   median  +1sigma  +2sigma
  a1       -0.2656  -0.1362   2.9895   0.1445   0.3141
  a2       -0.3209  -0.1309  -4.3469   0.1118   0.1985
  t1       -0.2377  -0.1305   1.3288   0.1448   0.3278
  t2       -1.0677  -0.4807  11.7836   0.4739   0.8990
```

```python
# <examples/doc_fitting_emcee.py>
import numpy as np

import lmfit

try:
    import matplotlib.pyplot as plt
    HASPYLAB = True
except ImportError:
    HASPYLAB = False

try:
    import corner
    HASCORNER = True
except ImportError:
    HASCORNER = False

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(x.size))
if HASPYLAB:
    plt.plot(x, y, 'b')
    plt.show()

p = lmfit.Parameters()
p.add_many(('a1', 4), ('a2', 4), ('t1', 3), ('t2', 3., True))
```

(continues on next page)

(continued from previous page)

```python
def residual(p):
    v = p.valuesdict()
    return v['a1']*np.exp(-x/v['t1']) + v['a2']*np.exp(-(x-0.1) / v['t2']) - y


mi = lmfit.minimize(residual, p, method='nelder', nan_policy='omit')
lmfit.printfuncs.report_fit(mi.params, min_correl=0.5)
if HASPYLAB:
    plt.figure()
    plt.plot(x, y, 'b')
    plt.plot(x, residual(mi.params) + y, 'r', label='best fit')
    plt.legend(loc='best')
    plt.show()

# Place bounds on the ln(sigma) parameter that emcee will automatically add
# to estimate the true uncertainty in the data since is_weighted=False
mi.params.add('__lnsigma', value=np.log(0.1), min=np.log(0.001), max=np.log(2))

res = lmfit.minimize(residual, method='emcee', nan_policy='omit', burn=300,
                     steps=1000, thin=20, params=mi.params, is_weighted=False,
                     progress=False)

if HASPYLAB and HASCORNER:
    emcee_corner = corner.corner(res.flatchain, labels=res.var_names,
                                 truths=list(res.params.valuesdict().values()))
    plt.show()

if HASPYLAB:
    plt.plot(res.acceptance_fraction)
    plt.xlabel('walker')
    plt.ylabel('acceptance fraction')
    plt.show()

if hasattr(res, "acor"):
    print("Autocorrelation time for the parameters:")
    print("----------------------------------------")
    for i, par in enumerate(p):
        print(par, res.acor[i])

print("\nmedian of posterior probability distribution")
print('--------------------------------------------')
lmfit.report_fit(res.params)


# find the maximum likelihood solution
highest_prob = np.argmax(res.lnprob)
hp_loc = np.unravel_index(highest_prob, res.lnprob.shape)
mle_soln = res.chain[hp_loc]
for i, par in enumerate(p):
    p[par].value = mle_soln[i]

print('\nMaximum Likelihood Estimation from emcee       ')
print('-------------------------------------------------')
print('Parameter  MLE Value   Median Value   Uncertainty')
fmt = '  {:5s}  {:11.5f} {:11.5f}   {:11.5f}'.format
for name, param in p.items():
    print(fmt(name, param.value, res.params[name].value,
```

(continues on next page)

(continued from previous page)

```
              res.params[name].stderr))

if HASPYLAB:
    plt.figure()
    plt.plot(x, y, 'b')
    plt.plot(x, residual(mi.params) + y, 'r', label='Nelder-Mead')
    plt.plot(x, residual(res.params) + y, 'k--', label='emcee')
    plt.legend()
    plt.show()

print('\nError Estimates from emcee    ')
print('------------------------------------------------------')
print('Parameter  -2sigma  -1sigma   median  +1sigma  +2sigma ')

for name in p.keys():
    quantiles = np.percentile(res.flatchain[name],
                              [2.275, 15.865, 50, 84.135, 97.275])
    median = quantiles[2]
    err_m2 = quantiles[0] - median
    err_m1 = quantiles[1] - median
    err_p1 = quantiles[3] - median
    err_p2 = quantiles[4] - median
    fmt = '  {:5s}   {:8.4f} {:8.4f} {:8.4f} {:8.4f} {:8.4f}'.format
    print(fmt(name, err_m2, err_m1, median, err_p1, err_p2))
```

**Total running time of the script:** ( 0 minutes 27.869 seconds)

# PYTHON MODULE INDEX

# INDEX