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.

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 buit-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)
print('best_vals: {}'.format(best_vals))
```

```
best_vals: [2.40484385 0.20537821 1.37159225]
```

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 automaticaly 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=5, amp=200, wid=1)
```

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=6.5, amp=100, wid=2.0)
```

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

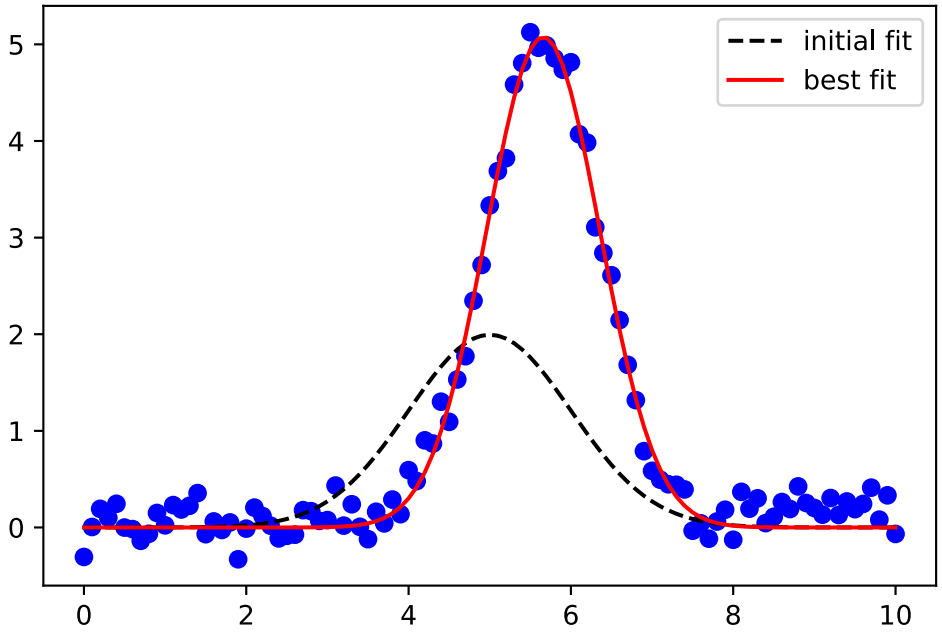
```
# <examples/doc_model_gaussian.py>
import matplotlib.pyplot as plt
from numpy import exp, loadtxt, pi, sqrt

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.

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*)

Model class.

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** (*str, optional*) – How to handle NaN and missing values in data. Must be one of 'raise' (default), 'propagate', or 'omit'. See Note 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']
```

`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*, ***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 of same size as `data`, optional) – Weights to use for the calculation of the fit residual (default is None).
- **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).
- **nan_policy** (*str*, *optional, one of 'raise' (default), 'propagate', or 'omit'.*) – What to do when encountering NaNs when fitting Model.
- **fit_kws** (*dict*, *optional*) – Options to pass to the minimizer being used.
- **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.
- ****kwargs** (*optional*) – Arguments to pass to the model function, possibly overriding params.

Returns:

Return type: 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)
```

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*.

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: params

Return type: Parameters

Notes

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

Raises: NotImplementedError –

`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

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()` for the named parameter.

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.

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.

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.

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()
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.

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.

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`.

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):
    ...
```

instead of using:

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

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

Initializing values with `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:

```
y1 = mod.eval(x=x, a=7.0, b=-2.0)
out = mod.fit(x=x, pars, 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()`.

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')
```

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 instance*) – 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:

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.

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*, ***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 (*str, optional, one of 'raise' (default), 'propagate', or 'omit'.*) – 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.
- **fit_kws (*optional*) – Keyword arguments to send to minimization routine.

`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: out – 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** (*str, optional, one of 'raise' (default), 'propagate', or 'omit'.*) – 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: **text** – Multi-line text of fit report.

Return type: str

See also

`fit_report()`

`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 nicely 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

Example

```
>>> 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')
```

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.

`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, if available.

The method will produce a matplotlib figure 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 on to the plot function for data points.
- **fit_kws** (*dict, optional*) – Keyword arguments passed on to the plot function for fitted curve.
- **init_kws** (*dict, optional*) – Keyword arguments passed on 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 there is one being created.
- **show_init** (*bool, optional*) – Whether to show the initial conditions for the fit (default is False).
- **parse_complex** (*str, optional*) – How to reduce complex data for plotting. Options are one of `['real', 'imag', 'abs', 'angle']`, which correspond to the numpy functions of the same name (default is 'abs').

Returns:

Return type: A tuple with matplotlib's Figure and GridSpec objects.

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.

See also

ModelResult.plot_fit()

Plot the fit results using matplotlib.

ModelResult.plot_residuals()

Plot the fit residuals using matplotlib.

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 on to the plot function for data points.
- **fit_kws** (*dict, optional*) – Keyword arguments passed on to the plot function for fitted curve.

- **init_kws** (*dict, optional*) – Keyword arguments passed on to the plot function for the initial conditions of the fit.
- **ax_kws** (*dict, optional*) – Keyword arguments for a new axis, if there is one being created.
- **show_init** (*bool, optional*) – Whether to show the initial conditions for the fit (default is False).
- **parse_complex** (*str, optional*) – How to reduce complex data for plotting. Options are one of `['real', 'imag', 'abs', 'angle']`, which correspond to the numpy functions of the same name (default is 'abs').

Returns:

Return type: matplotlib.axes.Axes

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.

See also

`ModelResult.plot_residuals()`

Plot the fit residuals using matplotlib.

`ModelResult.plot()`

Plot the fit results and residuals using matplotlib.

`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 on to the plot function for data points.
- **fit_kws** (*dict*, *optional*) – Keyword arguments passed on to the plot function for fitted curve.
- **ax_kws** (*dict*, *optional*) – Keyword arguments for a new axis, if there is one being created.
- **parse_complex** (*str*, *optional*) – How to reduce complex data for plotting. Options are one of `['real', 'imag', 'abs', 'angle']`, which correspond to the numpy functions of the same name (default is 'abs').

Returns:

Return type: matplotlib.axes.Axes

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.

See also

`ModelResult.plot_fit()`

Plot the fit results using matplotlib.

`ModelResult.plot()`

Plot the fit results and residuals using matplotlib.

`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()`.

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 have 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.

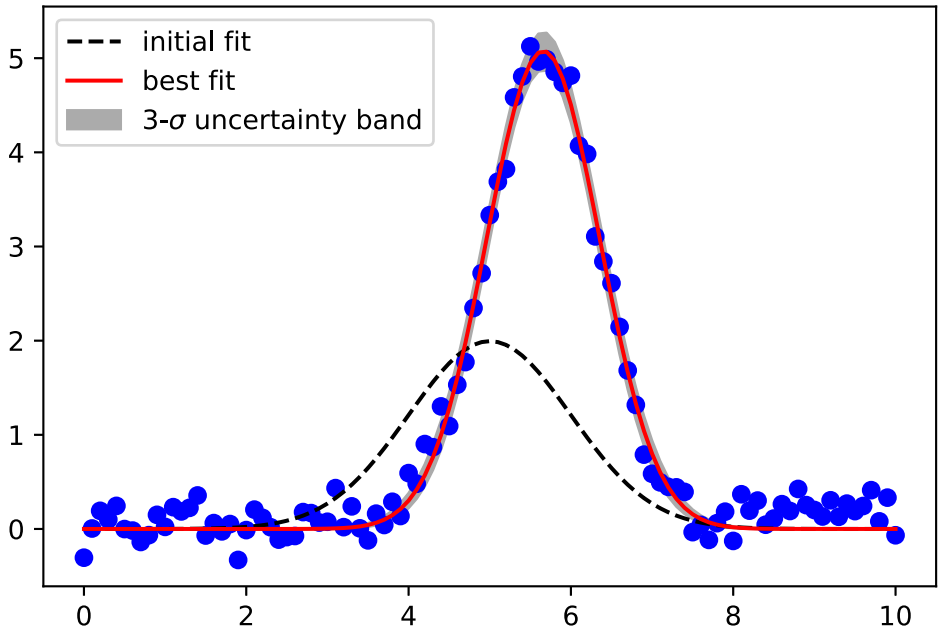
Calculating uncertainties in the model function

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 σ.

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-σ bands for the best-fit Gaussian, and produce the figure below.



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 instance*) – 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:

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>
```

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:

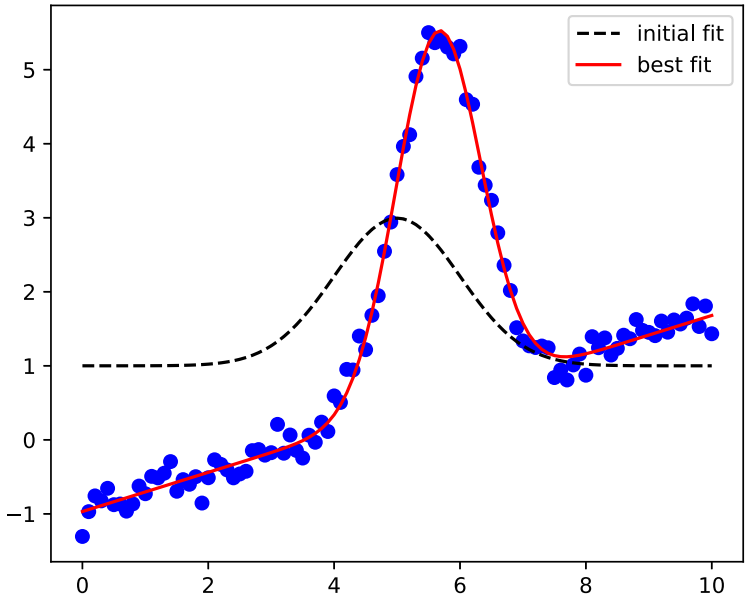
```
# <examples/doc_model_two_components.py>
import matplotlib.pyplot as plt
from numpy import exp, loadtxt, pi, sqrt

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>
```

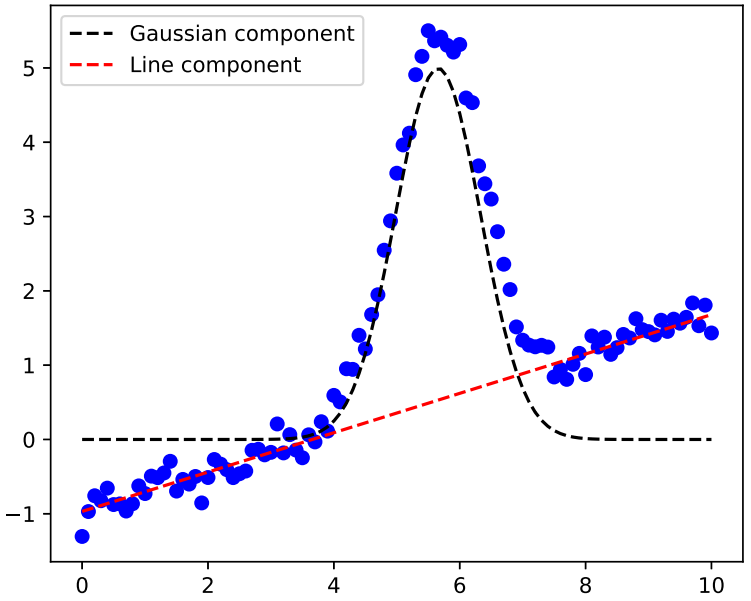
which prints out the results:

```
[[Model]]
    (Model(gaussian) + Model(line))
[[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 = np.linspace(-5, 25, 3001)
predicted = mod.eval(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 a binary operator (*op*) into a CompositeModel.

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

1. 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
```

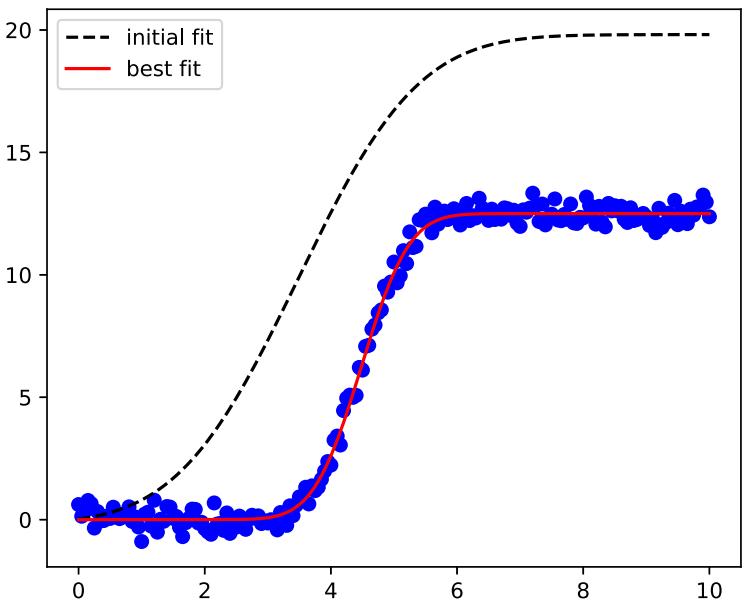
```
x = np.linspace(0, 10, 201)
y = step(x, amplitude=12.5, center=4.5, sigma=0.88, form='erf')
np.random.seed(0)
y = y + np.random.normal(scale=0.35, size=x.size)


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

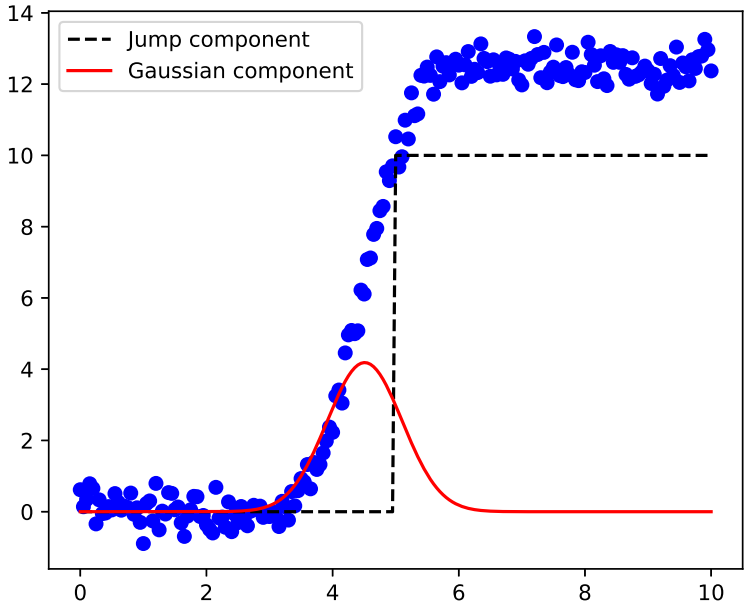
```
axes[1].plot(x, y, 'bo')
axes[1].plot(x, 10*comps['jump'], 'k--', label='Jump component')
axes[1].plot(x, 10*comps['gaussian'], 'r-', label='Gaussian component')
axes[1].legend(loc='best')

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

which prints out the results:

```
[[Model]]
    (Model(jump) <function convolve at 0x1a245b3400> 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.