lmfit documentation

Release 0.1

Matthew Newville

August 03, 2011

CONTENTS

The lmfit Python package provides a simple, flexible interface to non-linear least-squares fitting. LMFIT uses the Levenburg-Marquardt from MINPACK-1 as implemented in scipy.optimize.leastsq. While that function provides the core numerical routine for non-linear least-squares minimization, the lmfit packaage adds a few simple conveniences.

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

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

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

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

In addition, lmfit calculates are reports the estimated uncertainties and correlation between fitted variables.

CHAPTER

ONE

DOWNLOADING AND INSTALLATION

1.1 Prerequisites

The lmfit package requires Python, Numpy, and Scipy. Extensive testing on version compatibility has not yet been done. I have not yet tested with Python 3.

1.2 Downloads

The latest stable version is available from a few sources:

Download Option	Location
Source Kit	lmfit-0.1.tar.gz (CARS) or
Windows Installers	lmfit-0.1.win32-py2.6.exe or lmfit-0.1.win32-py2.7.exe
Development Version	use lmfit github repository

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

```
easy_install -U lmfit
```

1.3 Development Version

To get the latest development version, use:

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

1.4 Installation

Installation from source on any platform is:

```
python setup.py install
```

1.5 License

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

CHAPTER

TWO

NON-LINEAR LEAST-SQUARES FITTING WITH LMFIT-PY

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

2.1 Overview

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

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

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

In a traditional non-linear fit, one must write a function that takes the variable values and calculates the residual $y_i^{\rm meas} - y_i^{\rm model}(\mathbf{v})$, perhaps something like:

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

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

To perform the minimization with scipy, one would do:

```
from scipy.optimize import leastsq

vars = [10.0, 0.2, 3.0, 0.007]

out = leastsq(residual, vars, args=(x, data))
```

Though in python, this is not terribly different from how one would do the same fit in C or Fortran.

2.2 Using Parameters

The challenges in the approach above approach are that

1. The user has to keep track of the order of the variables, and their meaning – vars[2] is the frequency.
2. If the user wants to fix the phase-shift (*not* vary it in the fit), the residual function has to be altered. While reasonable for simple cases, this quickly becomes significant work.
3. There is no way to put bounds on values for the variables, or enforce mathematical relationships between the variables.

LMFIT is designed to void these shortcomings.

The key idea of LMFIT is to use a python dictionary of *Parameters*, which have more attributes than simply their value. The above example would be translated to look like:

```
from lmfit import minimize, Parameter

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

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

params = {'amp': Parameter(value=10),
          'decay': Parameter(value=0.007),
          'phase': Parameter(value=0.2),
          'frequency': Parameter(value=3.0)}

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

So far, this simply looks like it replaced a list of values with a dictionary. But Parameters objects can hold additional attributes to modify the value during the fit. For example, Parameters can be fixed or bounded:

```
params = {'amp': Parameter(value=10, vary=False),
          'decay': Parameter(value=0.007, min=0.0),
          'phase': Parameter(value=0.2),
          'frequency': Parameter(value=3.0, max=10.0)}
```

Now the fit will **not** vary the amplitude parameter, and will also impose a lower bound on the decay factor and an upper bound on the frequency. Importantly, the residual function remains unchanged.

2.3 The `Parameter` class

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

Parameters

- **value** – the numerical value for the parameter
- **vary** (boolean (`True`/`False`)) – whether to vary the parameter or not.
- **min** – lower bound for value (`None` = no lower bound).

- **max** – upper bound for value (`None` = no upper bound).
- **name** (`None` or string – will be overwritten during fit if `None`.) – parameter name
- **expr** (`None` or string) – mathematical expression to use to evaluate value during fit.

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

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

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

`correl`
: a dictionary of the correlation with the other fitted variables in the fit, of the form ‘ {‘decay’: 0.404, ‘phase’: -0.020, ‘frequency’: 0.102}‘

CHAPTER

THREE

OUTPUTS

CHAPTER

FOUR

CONSTRAINTS

4.1 Overview

INDEX