lsqfit Documentation

Release 8.1

G. P. Lepage

November 20, 2016

Contents:

CHAPTER

ONE

OVERVIEW AND TUTORIAL

1.1 Introduction

The *lsqfit* module is designed to facilitate least-squares fitting of noisy data by multi-dimensional, nonlinear functions of arbitrarily many parameters, each with a (Bayesian) prior. *lsqfit* makes heavy use of another module, gvar (distributed separately), which provides tools that simplify the analysis of error propagation, and also the creation of complicated multi-dimensional Gaussian distributions. The power of the gvar module, particularly for correlated distributions, is a feature that distinguishes *lsqfit* from standard fitting packages, as demonstrated below.

The following (complete) code illustrates basic usage of *lsqfit*:

```
import numpy as np
import gvar as gv
import lsqfit

y = {                                   # data for the dependent variable
   'data1' : gv.gvar([1.376, 2.010], [[ 0.0047, 0.01], [ 0.01, 0.056]]),
   'data2' : gv.gvar([1.329, 1.582], [[ 0.0047, 0.0067], [0.0067, 0.0136]]),
   'b/a'   : gv.gvar(2.0, 0.5)
   }
x = {                                   # independent variable
   'data1' : np.array([0.1, 1.0]),
   'data2' : np.array([0.1, 0.5])
   }
prior = {}
prior['a'] = gv.gvar(0.5, 0.5)
prior['b'] = gv.gvar(0.5, 0.5))

def fcn(x, p):                          # fit function of x and parameters p
  ans = {}
  for k in ['data1', 'data2']:
     ans[k] = gv.exp(p['a'] + x[k] * p['b'])
  ans['b/a'] = p['b'] / p['a']
  return ans

# do the fit
fit = lsqfit.nonlinear_fit(data=(x, y), prior=prior, fcn=fcn, debug=True)
print(fit.format(maxline=True))         # print standard summary of fit

p = fit.p                               # best-fit values for parameters
outputs = dict(a=p['a'], b=p['b'])
outputs['b/a'] = p['b']/p['a']
inputs = dict(y=y, prior=prior)
print(gv.fmt_values(outputs))               # tabulate outputs
print(gv.fmt_errorbudget(outputs, inputs)) # print error budget for outputs
```

This code fits the function `f(x,a,b)= exp(a+b*x)` (see `fcn(x,p)`) to two sets of data, labeled `data1` and `data2`, by varying parameters `a` and `b` until `f(x['data1'],a,b)` and `f(x['data2'],a,b)` equal `y['data1']` and `y['data2']`, respectively, to within the `y`s' errors. The means and covariance matrices for the `y`s are specified in the `gv.gvar(...)`s used to create them: for example,

```
>>> print(y['data1'])
[1.376(69) 2.01(24)]
>>> print(y['data1'][0].mean, "+-", y['data1'][0].sdev)
1.376 +- 0.068556546004
>>> print(gv.evalcov(y['data1']))    # covariance matrix
[[ 0.0047  0.01  ]
 [ 0.01    0.056 ]]
```

shows the means, standard deviations and covariance matrix for the data in the first data set (0.0685565 is the square root of the 0.0047 in the covariance matrix). The dictionary `prior` gives *a priori* estimates for the two parameters, `a` and `b`: each is assumed to be 0.5±0.5 before fitting. The parameters `p[k]` in the fit function `fcn(x, p)` are stored in a dictionary having the same keys and layout as `prior` (since `prior` specifies the fit parameters for the fitter). In addition, there is an extra piece of input data, `y['b/a']`, which indicates that `b/a` is 2±0.5. The fit function for this data is simply the ratio `b/a` (represented by `p['b']/p['a']` in fit function `fcn(x,p)`). The fit function returns a dictionary having the same keys and layout as the input data `y`.

The output from the code sample above is:

```
Least Square Fit:
  chi2/dof [dof] = 0.17 [5]    Q = 0.97    logGBF = 0.65538

Parameters:
              a   0.253 (32)      [  0.50 (50) ]
              b   0.449 (65)      [  0.50 (50) ]

Fit:
        key         y[key]      f(p)[key]
-----------------------------------------
        b/a      2.00 (50)      1.78 (30)
    data1 0     1.376 (69)     1.347 (46)
          1      2.01 (24)      2.02 (16)
    data2 0     1.329 (69)     1.347 (46)
          1      1.58 (12)     1.612 (82)

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0    (itns/time = 5/0.0)

Values:
                  a: 0.253(32)
                b/a: 1.78(30)
                  b: 0.449(65)

Partial % Errors:
                     a       b/a         b
-----------------------------------------
        y:       12.75     16.72     14.30
    prior:        0.92      1.58      1.88
-----------------------------------------
    total:       12.78     16.80     14.42
```

The best-fit values for `a` and `b` are 0.253(32) and 0.449(65), respectively; and the best-fit result for `b/a` is 1.78(30), which, because of correlations, is slightly more accurate than might be expected from the separate errors for `a` and `b`.

The error budget for each of these three quantities is tabulated at the end and shows that the bulk of the error in each case comes from uncertainties in the `y` data, with only small contributions from uncertainties in the priors `prior`. The fit results corresponding to each piece of input data are also tabulated (`Fit: ...`); the agreement is excellent, as expected given that the `chi**2` per degree of freedom is only 0.17.

Note that the constraint in `y` on `b/a` in this example is much tighter than the constraints on `a` and `b` separately. This suggests a variation on the previous code, where the tight restriction on `b/a` is built into the prior rather than `y`:

```
... as before ...

y = {                       # data for the dependent variable
    'data1' : gv.gvar([1.376, 2.010], [[ 0.0047, 0.01], [ 0.01, 0.056]]),
    'data2' : gv.gvar([1.329, 1.582], [[ 0.0047, 0.0067], [0.0067, 0.0136]])
    }
x = {                       # independent variable
    'data1' : np.array([0.1, 1.0]),
    'data2' : np.array([0.1, 0.5])
    }
prior = {}
prior['a'] = gv.gvar(0.5, 0.5)
prior['b'] = prior['a'] * gv.gvar(2.0, 0.5)

def fcn(x, p):              # fit function of x and parameters p[k]
   ans = {}
   for k in ['data1', 'data2']:
      ans[k] = gv.exp(p['a'] + x[k]*p['b'])
   return ans

... as before ...
```

Here the dependent data `y` no longer has an entry for `b/a`, and neither do results from the fit function; but the prior for `b` is now 2 ± 0.5 times the prior for `a`, thereby introducing a correlation that limits the ratio `b/a` to be 2 ± 0.5 in the fit. This code gives almost identical results to the first one — very slightly less accurate, since there is less input data. We can often move information from the `y` data to the prior or back since both are forms of input information.

There are several things worth noting from this example:

- The input data (`y`) is expressed in terms of Gaussian random variables — quantities with means and a covariance matrix. These are represented by objects of type `gvar.GVar` in the code; module `gvar` has a variety of tools for creating and manipulating Gaussian random variables (also see below).
- The input data is stored in a dictionary (`y`) whose values can be `gvar.GVar`s or arrays of `gvar.GVar`s. The use of a dictionary allows for far greater flexibility than, say, an array. The fit function (`fcn(x, p)`) has to return a dictionary with the same layout as that of `y` (that is, with the same keys and where the value for each key has the same shape as the corresponding value in `y`). *`lsqfit`* allows `y` to be an array instead of a dictionary, which might be preferable for very simple fits (but usually not otherwise).
- The independent data (`x`) can be anything; it is simply passed through the fit code to the fit function `fcn(x,p)`. It can also be omitted altogether, in which case the fit function depends only upon the parameters: `fcn(p)`.
- The fit parameters (`p` in `fcn(x,p)`) are also stored in a dictionary whose values are `gvar.GVar`s or arrays of `gvar.GVar`s. Again this allows for great flexibility. The layout of the parameter dictionary is copied from that of the prior (`prior`). Again `p` can be a single array instead of a dictionary, if that simplifies the code (which is usually not the case).
- The best-fit values of the fit parameters (`fit.p[k]`) are also `gvar.GVar`s and these capture statistical correlations between different parameters that are indicated by the fit. These output parameters can be combined in arithmetic expressions, using standard operators and standard functions, to obtain derived quantities. These operations take account of and track statistical correlations.

- Function `gvar.fmt_errorbudget()` is a useful tool for assessing the origins (`inputs`) of the statistical errors obtained in various final results (`outputs`). It is particularly useful for analyzing the impact of the *a priori* uncertainties encoded in the prior (`prior`).
- Parameter `debug=True` is set in *`lsqfit.nonlinear_fit`*. This is a good idea, particularly in the eary stages of a project, because it causes the code to check for various common errors and give more intelligible error messages than would otherwise arise. This parameter can be dropped once code development is over.
- The priors for the fit parameters specify Gaussian distributions, characterized by the means and standard deviations given `gv.gvar(...)`. Some other distributions become available if argument `extend=True` is included in the call to *`lsqfit.nonlinear_fit`*. The distribution for parameter `a`, for example, can then be switched to a log-normal distribution by replacing `prior['a']=gv.gvar(0.5, 0.5)` with:

```
prior['log(a)'] = gv.log(gv.gvar(0.5,0.5))
```

  in the code. This change would be desirable if we knew *a priori* that parameter `a` is positive since this is guaranteed with a log-normal distribution. Only the prior need be changed (in particular, the fit function `fcn(x,p)` need *not* be changed).

What follows is a tutorial that demonstrates in greater detail how to use these modules in some standard variations on the data fitting problem. As above, code for the examples is specified completely and so can be copied into a file, and run as is. It can also be modified, allowing for experimentation.

Another way to learn about the modules is to examine the case studies that follow this section. Each focuses on a single problem, again with the full code and data to allow for experimentation.

About Printing: The examples in this tutorial use the `print` function as it is used in Python 3. Drop the outermost parenthesis in each `print` statement if using Python 2; or add

```
from __future__ import print_function
```

at the start of your file.

1.2 Gaussian Random Variables and Error Propagation

The inputs and outputs of a nonlinear least squares analysis are probability distributions, and these distributions will be Gaussian provided the input data are sufficiently accurate. *`lsqfit`* assumes this to be the case. (It also provides tests for non-Gaussian behavior, together with methods for dealing with such behavior.) One of the most distinctive features of *`lsqfit`* is that it is built around a class, `gvar.GVar`, of objects that can be used to represent arbitrarily complicated Gaussian distributions — that is, they represent *Gaussian random variables* that specify the means and covariance matrix of the probability distributions. The input data for a fit are represented by a collection of `gvar.GVar`s that specify both the values and possible errors in the input values. The result of a fit is a collection of `gvar.GVar`s specifying the best-fit values for the fit parameters and the estimated uncertainties in those values.

`gvar.GVar`s are defined in the `gvar` module. There are four important things to know about them (see the `gvar` documentation for more details):

1. `gvar.GVar`s are created by `gvar.gvar()`, individually or in groups: for example,

```
>>> import gvar as gv
>>> print(gv.gvar(1.0, 0.1), gv.gvar('1.0 +- 0.2'), gv.gvar('1.0(4)'))
1.00(10) 1.00(20) 1.00(40)
>>> print(gv.gvar([1.0, 1.0, 1.0], [0.1, 0.2, 0.41]))
[1.00(10) 1.00(20) 1.00(41)]
>>> print(gv.gvar(['1.0(1)', '1.0(2)', '1.00(41)']))
[1.00(10) 1.00(20) 1.00(41)]
>>> print(gv.gvar(dict(a='1.0(1)', b=['1.0(2)', '1.0(4)'])))
{'a': 1.00(10),'b': array([1.00(20), 1.00(40)], dtype=object)}
```

gvar uses the compact notation 1.234(22) to represent 1.234±0.022 — the digits in parentheses indicate the uncertainty in the rightmost corresponding digits quoted for the mean value. Very large (or small) numbers use a notation like 1.234(22)e10.

2. gvar.GVars describe not only means and standard deviations, but also statistical correlations between different objects. For example, the gvar.GVars created by

```
>>> import gvar as gv
>>> a, b = gv.gvar([1, 1], [[0.01, 0.01], [0.01, 0.010001]])
>>> print(a, b)
1.00(10) 1.00(10)
```

both have means of 1 and standard deviations equal to or very close to 0.1, but the ratio b/a has a standard deviation that is 100x smaller:

```
>>> print(b / a)
1.0000(10)
```

This is because the covariance matrix specified for a and b when they were created has large, positive off-diagonal elements:

```
>>> print(gv.evalcov([a, b]))          # covariance matrix
[[ 0.01      0.01    ]
 [ 0.01      0.010001]]
```

These off-diagonal elements imply that a and b are strongly correlated, which means that b/a or b-a will have much smaller uncertainties than a or b separately. The correlation coefficient for a and b is 0.99995:

```
>>> print(gv.evalcorr([a, b]))         # correlation matrix
[[ 1.       0.99995]
 [ 0.99995  1.     ]]
```

3. gvar.GVars can be used in arithmetic expressions or as arguments to pure-Python functions. The results are also gvar.GVars. Covariances are propagated through these expressions following the usual rules, (automatically) preserving information about correlations. For example, the gvar.GVars a and b above could have been created using the following code:

```
>>> a = gv.gvar(1, 0.1)
>>> b = a + gv.gvar(0, 0.001)
>>> print(a, b)
1.00(10) 1.00(10)
>>> print(b / a)
1.0000(10)
>>> print(gv.evalcov([a, b]))
[[ 0.01      0.01    ]
 [ 0.01      0.010001]]
```

The correlation is obvious from this code: b is equal to a plus a very small correction. From these variables we can create new variables that are also highly correlated:

```
>>> x = gv.log(1 + a ** 2)
>>> y = b * gv.cosh(a / 2)
>>> print(x, y, y / x)
0.69(10) 1.13(14) 1.627(34)
>>> print gv.evalcov([x, y])
[[ 0.01        0.01388174]
 [ 0.01388174  0.01927153]]
```

The gvar module defines versions of the standard Python functions (sin, cos, ...) that work with gvar.GVars. Most any numeric pure-Python function will work with them as well. Numeric functions

that are compiled in C or other low-level languages generally do not work with `gvar.GVars`; they should be replaced by equivalent pure-Python functions if they are needed for `gvar.GVar`-valued arguments. See the `gvar` documentation for more information.

The fact that correlation information is preserved *automatically* through arbitrarily complicated arithmetic is what makes `gvar.GVars` particularly useful. This is accomplished using *automatic differentiation* to compute the derivatives of any *derived* `gvar.GVar` with respect to the *primary* `gvar.GVars` (those defined using `gvar.gvar()`) from which it was created. As a result, for example, we need not provide derivatives of fit functions for *lsqfit* (which are needed for the fit) since they are computed implicitly by the fitter from the fit function itself. Also it becomes trivial to build correlations into the priors used in fits, and to analyze the propagation of errors through complicated functions of the parameters after the fit.

4. Storing `gvar.GVars` in a file for later use is somewhat complicated because one generally wants to hold onto their correlations as well as their mean values and standard deviations. One easy way to do this is to put all of the `gvar.GVars` to be saved into a single dictionary object of type `gvar.BufferDict`, and then to save the `gvar.BufferDict` using Python's `pickle` module: for example, using the variables defined above,

```
>>> import pickle
>>> buffer = gv.BufferDict(a=a, b=b, x=x, y=y)
>>> print(buffer)
{'a': 1.00(10),'b': 1.00(10),'x': 0.69(10),'y': 1.13(14)}
>>> pickle.dump(buffer, open('outputfile.p', 'wb'))
```

This creates a file named `'outputfile.p'` containing the `gvar.GVars`. Loading the file into a Python code later recovers the `gvar.BufferDict` with correlations intact:

```
>>> buffer = pickle.load(open('outputfile.p', 'rb'))
>>> print(buffer)
{'a': 1.00(10),'b': 1.00(10),'x': 0.69(10),'y': 1.13(14)}
>>> print(buffer['y'] / buffer['x'])
1.627(34)
```

`gvar.BufferDicts` were created specifically to handle `gvar.GVars`, although they can be quite useful with other data types as well. The values in a pickled `gvar.BufferDict` can be individual `gvar.GVars` or arbitrary `numpy` arrays of `gvar.GVars`. See the `gvar` docmentation for more information.

There is considerably more information about `gvar.GVars` in the documentation for module `gvar`.

1.3 Basic Fits

A fit analysis typically requires three types of input: 1) fit data `x,y` (or possibly just `y`); 2) a function `y = f(x, p)` relating values of `y` to to values of `x` and a set of fit parameters `p` (if there is no `x`, then `y = f(p)`); and 3) some *a priori* idea about the fit parameters' values. The *a priori* information about a parameter could be fairly imprecise — for example, the parameter is order 1. The point of the fit is to improve our knowledge of the parameter values, beyond our *a priori* impressions, by analyzing the fit data. We now show how to do this using the *lsqfit* module.

For this example, we use fake data generated by a function, `make_data()`, that is described at the end of this section. The function call `x,y = make_data()` generates 11 values for `x`, equal to `5,6,7..10,12,14..20`, and 11 values for `y`, where each `y` is obtained by adding random noise to the value of a function of the corresponding `x`. The function of `x` we use is:

```
sum(a[i] * exp(-E[i]*x)  for i in range(100))
```

where `a[i]=0.4` and `E[i]=0.9*(i+1)`. The result is a set of random `y`s with correlated statistical errors:

```
>>> print(y)
[0.004502(46) 0.001817(19) 0.0007362(79) ... ]
```

```
>>> print(gv.evalcov(y))               # covariance matrix
[[  2.15378088e-09   8.81617947e-10   3.62373566e-10 ... ]
 [  8.81617947e-10   3.61934618e-10   1.49216108e-10 ... ]
 [  3.62373566e-10   1.49216108e-10   6.17104688e-11 ... ]
 ...
]
```

Our goal is to fit this data for `y`, as a function of `x`, and obtain estimates for the parameters `a[i]` and `E[i]`. The correct results are, of course, `a[i]=0.4` and `E[i]=0.9*(i+1)` but we will pretend that we do not know this.

Next we need code for the fit function. We assume that we know that a sum of exponentials is appropriate, and therefore we define the following Python function to represent the relationship between `x` and `y` in our fit:

```
import numpy as np

def f(x, p):          # function used to fit x, y data
    a = p['a']        # array of a[i]s
    E = p['E']        # array of E[i]s
    return sum(ai * np.exp(-Ei * x) for ai, Ei in zip(a, E))
```

The fit parameters, `a[i]` and `E[i]`, are stored in a dictionary, using labels `a` and `E` to access them. These parameters are varied in the fit to find the best-fit values `p=p_fit` for which `f(x, p_fit)` most closely approximates the `y`s in our fit data. The number of exponentials included in the sum is specified implicitly in this function, by the lengths of the `p['a']` and `p['E']` arrays.

Finally we need to define priors that encapsulate our *a priori* knowledge about the fit-parameter values. In practice we almost always have *a priori* knowledge about parameters; it is usually impossible to design a fit function without some sense of the parameter sizes. Given such knowledge it is important (usually essential) to include it in the fit. This is done by designing priors for the fit, which are probability distributions for each parameter that describe the *a priori* uncertainty in that parameter. As discussed in the previous section, we use objects of type `gvar.GVar` to describe (Gaussian) probability distributions. Let's assume that before the fit we suspect that each `a[i]` is of order 0.5±0.4, while `E[i]` is of order (1+i)±0.4. A prior that represents this information is built using the following code:

```
import lsqfit
import gvar as gv

def make_prior(nexp):                 # make priors for fit parameters
    prior = gv.BufferDict()           # prior -- any dictionary works
    prior['a'] = [gv.gvar(0.5, 0.4) for i in range(nexp)]
    prior['E'] = [gv.gvar(i+1, 0.4) for i in range(nexp)]
    return prior
```

where `nexp` is the number of exponential terms that will be used (and therefore the number of `a`s and `E`s). With `nexp=3`, for example, one would then have:

```
>>> print(prior['a'])
[0.50(40) 0.50(40) 0.50(40)]
>>> print(prior['E'])
[1.00(40), 2.00(40), 3.00(40)]
```

We use dictionary-like class `gvar.BufferDict` for the prior because it allows us to save the prior in a file if we wish (using Python's `pickle` module). If saving is unnecessary, `gvar.BufferDict` can be replaced by `dict()` or most any other Python dictionary class.

With fit data, a fit function, and a prior for the fit parameters, we are finally ready to do the fit, which is now easy:

```
fit = lsqfit.nonlinear_fit(data=(x, y), fcn=f, prior=prior)
```

So pulling together the entire code, our complete Python program for making fake data and fitting it is:

```
import lsqfit
import numpy as np
import gvar as gv

def f_exact(x, nexp=100):           # exact f(x)
    return sum(0.4 * np.exp(-0.9*(i+1) * x) for i in range(nexp))

def f(x, p):                        # function used to fit x, y data
    a = p['a']                      # array of a[i]s
    E = p['E']                      # array of E[i]s
    return sum(ai * np.exp(-Ei * x) for ai, Ei in zip(a, E))

def make_data(nexp=100, eps=0.01): # make x, y fit data
    x = np.array([5., 6., 7., 8., 9., 10., 12., 14., 16., 18., 20.])
    cr = gv.gvar(0.0, eps)
    c = [gv.gvar(cr(), eps) for n in range(100)]
    x_xmax = x/max(x)
    noise = 1+ sum(c[n] * x_xmax ** n for n in range(100))
    y = f_exact(x, nexp) * noise
    return x, y

def make_prior(nexp):               # make priors for fit parameters
    prior = gv.BufferDict()         # prior -- any dictionary works
    prior['a'] = [gv.gvar(0.5, 0.4) for i in range(nexp)]
    prior['E'] = [gv.gvar(i+1, 0.4) for i in range(nexp)]
    return prior

def main():
    gv.ranseed([2009, 2010, 2011, 2012]) # initialize random numbers (opt.)
    x, y = make_data()              # make fit data
    p0 = None                       # make larger fits go faster (opt.)
    for nexp in range(1, 11):
        print('************************************ nexp =', nexp)
        prior = make_prior(nexp)
        fit = lsqfit.nonlinear_fit(data=(x, y), fcn=f, prior=prior, p0=p0)
        print(fit)                  # print the fit results
        if nexp > 2:
            E = fit.p['E']          # best-fit parameters
            a = fit.p['a']
            print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
            print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
        if fit.chi2 / fit.dof < 1.:
            p0 = fit.pmean          # starting point for next fit (opt.)
        print()

if __name__ == '__main__':
    main()
```

We are not sure *a priori* how many exponentials are needed to fit our data. Given that there are only eleven `y`s, and these are noisy, there may only be information in the data about the first few terms. Consequently we write our code to try fitting with each of `nexp=1,2,3..10` terms. (The pieces of the code involving `p0` are optional; they make the more complicated fits go about 30 times faster since the output from one fit is used as the starting point for the next fit — see the discussion of the `p0` parameter for *`lsqfit.nonlinear_fit`*.) Running this code produces the following output, which is reproduced here in some detail in order to illustrate a variety of features:

```
************************************ nexp = 1
Least Square Fit:
  chi2/dof [dof] = 9.2e+02 [11]    Q = 0    logGBF = -4830
```

```
Parameters:
            a 0   0.00150 (27)      [  0.50 (40) ]  *
            E 0    1.2467 (89)      [  1.00 (40) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 7/0.0)


************************************ nexp = 2
Least Square Fit:
  chi2/dof [dof] = 4.6 [11]    Q = 5.3e-07    logGBF = 162.76

Parameters:
            a 0    0.4014 (40)      [  0.50 (40) ]
              1    0.4501 (45)      [  0.50 (40) ]
            E 0   0.90158 (50)      [  1.00 (40) ]
              1   1.83335 (71)      [  2.00 (40) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 5/0.0)


************************************ nexp = 3
Least Square Fit:
  chi2/dof [dof] = 0.57 [11]    Q = 0.85    logGBF = 182.44

Parameters:
            a 0    0.4020 (40)      [  0.50 (40) ]
              1    0.4108 (90)      [  0.50 (40) ]
              2      0.59 (35)      [  0.50 (40) ]
            E 0   0.90045 (53)      [  1.00 (40) ]
              1    1.8055 (50)      [  2.00 (40) ]
              2      2.85 (19)      [  3.00 (40) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 30/0.0)

E1/E0 = 2.0050(52)   E2/E0 = 3.17(21)
a1/a0 = 1.022(20)   a2/a0 = 1.47(88)

************************************ nexp = 4
Least Square Fit:
  chi2/dof [dof] = 0.57 [11]    Q = 0.85    logGBF = 182.45

Parameters:
            a 0    0.4020 (40)      [  0.50 (40) ]
              1    0.4110 (90)      [  0.50 (40) ]
              2      0.59 (35)      [  0.50 (40) ]
              3      0.50 (40)      [  0.50 (40) ]
            E 0   0.90045 (53)      [  1.00 (40) ]
              1    1.8055 (50)      [  2.00 (40) ]
              2      2.85 (19)      [  3.00 (40) ]
              3      4.00 (40)      [  4.00 (40) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 3/0.0)

E1/E0 = 2.0051(52)   E2/E0 = 3.17(21)
```

```
a1/a0 = 1.022(20)   a2/a0 = 1.47(88)

************************************ nexp = 5
Least Square Fit:
  chi2/dof [dof] = 0.57 [11]    Q = 0.85    logGBF = 182.45

Parameters:
            a 0    0.4020 (40)      [  0.50 (40) ]
              1    0.4110 (90)      [  0.50 (40) ]
              2      0.59 (35)      [  0.50 (40) ]
              3      0.50 (40)      [  0.50 (40) ]
              4      0.50 (40)      [  0.50 (40) ]
            E 0   0.90045 (53)      [  1.00 (40) ]
              1    1.8055 (50)      [  2.00 (40) ]
              2      2.85 (19)      [  3.00 (40) ]
              3      4.00 (40)      [  4.00 (40) ]
              4      5.00 (40)      [  5.00 (40) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 1/0.0)

E1/E0 = 2.0051(52)   E2/E0 = 3.17(21)
a1/a0 = 1.022(20)   a2/a0 = 1.47(88)
                                    .
                                    .
                                    .
                                    .

************************************ nexp = 10
Least Square Fit:
  chi2/dof [dof] = 0.57 [11]    Q = 0.85    logGBF = 182.45

Parameters:
            a 0    0.4020 (40)      [  0.50 (40) ]
              1    0.4110 (90)      [  0.50 (40) ]
              2      0.59 (35)      [  0.50 (40) ]
              3      0.50 (40)      [  0.50 (40) ]
              4      0.50 (40)      [  0.50 (40) ]
              5      0.50 (40)      [  0.50 (40) ]
              6      0.50 (40)      [  0.50 (40) ]
              7      0.50 (40)      [  0.50 (40) ]
              8      0.50 (40)      [  0.50 (40) ]
              9      0.50 (40)      [  0.50 (40) ]
            E 0   0.90045 (53)      [  1.00 (40) ]
              1    1.8055 (50)      [  2.00 (40) ]
              2      2.85 (19)      [  3.00 (40) ]
              3      4.00 (40)      [  4.00 (40) ]
              4      5.00 (40)      [  5.00 (40) ]
              5      6.00 (40)      [  6.00 (40) ]
              6      7.00 (40)      [  7.00 (40) ]
              7      8.00 (40)      [  8.00 (40) ]
              8      9.00 (40)      [  9.00 (40) ]
              9     10.00 (40)      [ 10.00 (40) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0    (itns/time = 1/0.0)

E1/E0 = 2.0051(52)   E2/E0 = 3.17(21)
```

```
a1/a0 = 1.022(20)    a2/a0 = 1.47(88)
```

There are several things to notice here:

- Clearly two exponentials (`nexp=2`) is not enough. The `chi**2` per degree of freedom (`chi2/dof`) is significantly larger than one. The `chi**2` improves substantially for `nexp=3` exponentials, and there is essentially no change when further exponentials are added.
- The best-fit values for each parameter are listed for each of the fits, together with the prior values (in brackets, on the right). Values for each `a[i]` and `E[i]` are listed in order, starting at the points indicated by the labels `a` and `E`. Asterisks are printed at the end of the line if the mean best-fit value differs from the prior's mean by more than one standard deviation; the number of asterisks, up to a maximum of 5, indicates how many standard deviations the difference is. Differences of one or two standard deviations are not uncommon; larger differences could indicate a problem with the prior or the fit.

  Once the fit converges, the best-fit values for the various parameters agree well — that is to within their errors, approximately — with the exact values, which we know since we are using fake data. For example, `a` and `E` for the first exponential are 0.402(4) and 0.9004(5), respectively, from the fit where the exact answers are 0.4 and 0.9; and we get 0.411(9) and 1.806(5) for the second exponential where the exact values are 0.4 and 1.8.
- Note in the `nexp=4` fit how the means and standard deviations for the parameters governing the fourth (and last) exponential are essentially identical to the values in the corresponding priors: 0.50(40) from the fit for `a` and 4.0(4) for `E`. This tells us that our fit data has no information to add to what we knew *a priori* about these parameters — there isn't enough data and what we have isn't accurate enough.

  This situation remains true of further terms as they are added in the `nexp=5` and later fits. This is why the fit results stop changing once we have `nexp=3` exponentials. There is no point in including further exponentials, beyond the need to verify that the fit has indeed converged.
- The last fit includes `nexp=10` exponentials and therefore has 20 parameters. This is in a fit to 11 `ys`. Old-fashioned fits, without priors, are impossible when the number of parameters exceeds the number of data points. That is clearly not the case here, where the number of terms and parameters can be made arbitrarily large, eventually (after `nexp=3` terms) with no effect at all on the results.

  The reason is that the prior that we include for each new parameter is, in effect, a new piece of data (the mean and standard deviation of the *a priori* expectation for that parameter); it leads to a new term in the `chi**2` function. We are fitting both the data and our *a priori* expectations for the parameters. So in the `nexp=10` fit, for example, we actually have 31 pieces of data to fit: the 11 `ys` plus the 20 prior values for the 20 parameters.

  The effective number of degrees of freedom (`dof` in the output above) is the number of pieces of data minus the number of fit parameters, or 31-20=11 in this last case. With priors for every parameter, the number of degrees of freedom is always equal to the number of `ys`, irrespective of how many fit parameters there are.
- The Gaussian Bayes Factor (whose logarithm is `logGBF` in the output) is a measure of the likelihood that the actual data being fit could have come from a theory with the prior and fit function used in the fit. The larger this number, the more likely it is that prior/fit-function and data could be related. Here it grows dramatically from the first fit (`nexp=1`) but then stops changing after `nexp=3`. The implication is that this data is much more likely to have come from a theory with `nexp>=3` than with `nexp=1` (which we know to be the actual case).
- In the code, results for each fit are captured in a Python object `fit`, which is of type *`lsqfit.nonlinear_fit`*. A summary of the fit information is obtained by printing `fit`. Also the best-fit results for each fit parameter can be accessed through `fit.p`, as is done here to calculate various ratios of parameters.

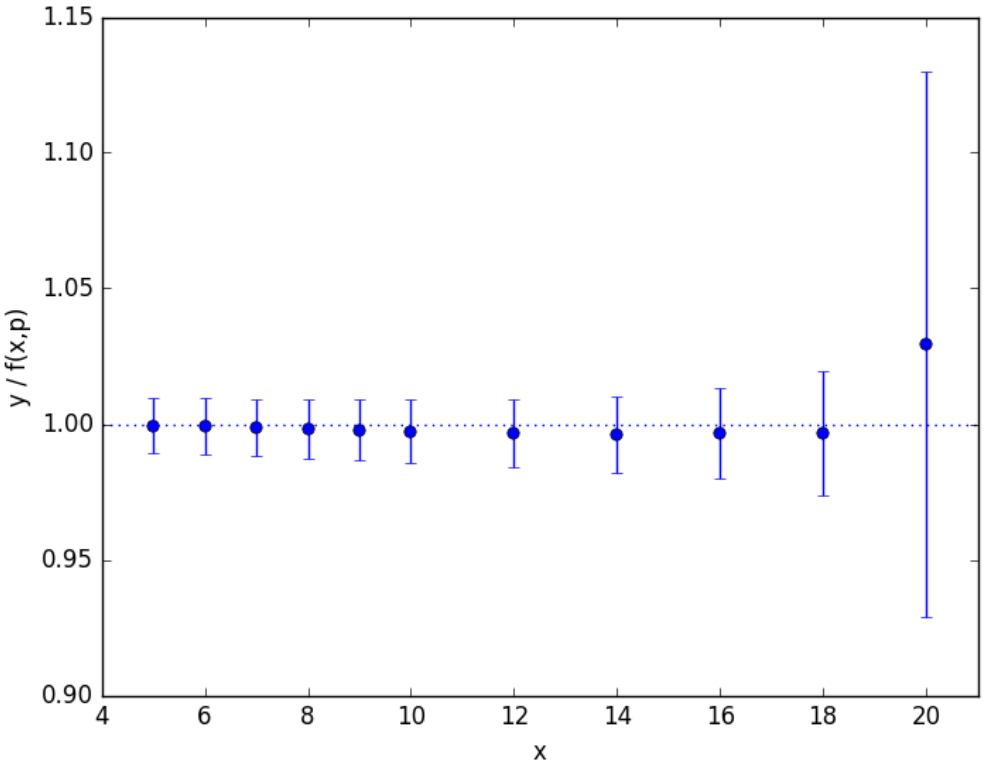
  The errors in these last calculations automatically account for any correlations in the statistical errors for different parameters. This is evident in the ratio `a1/a0`, which would be 1.022(25) if there was no statistical correlation between our estimates for `a1` and `a0`, but in fact is 1.022(20) in this fit. The modest (positive) correlation is clear from the correlation matrix:

```
>>> print(gv.evalcorr(a[:2]))
[[ 1.          0.48404447]
 [ 0.48404447  1.        ]]
```

Finally we inspect the fit's quality point by point. The input data are compared with results from the fit function, evaluated with the best-fit parameters, in the following table (obtained in the code by printing the output from `fit.format(maxline=True)`):

```
Fit:
     x[k]               y[k]           f(x[k],p)
-----------------------------------------------
        5      0.004502 (46)       0.004505 (46)
        6      0.001817 (19)       0.001819 (19)
        7     0.0007362 (79)      0.0007371 (78)
        8     0.0002987 (33)      0.0002992 (32)
        9     0.0001213 (14)      0.0001216 (13)
       10    0.00004926 (57)     0.00004939 (56)
       12       8.13(10)e-06        8.156(96)e-06
       14      1.342(19)e-06       1.347(17)e-06
       16      2.217(37)e-07       2.224(29)e-07
       18      3.661(85)e-08       3.674(50)e-08
       20       6.24(61)e-09        6.067(88)e-09
```

The fit is excellent over the entire six orders of magnitude. This information is presented again in the following plot, which shows the ratio `y/f(x,p)`, as a function of `x`, using the best-fit parameters `p`. The correct result for this ratio, of course, is one. The smooth variation in the data — smooth compared with the size of the statistical-error bars — is an indication of the statistical correlations between individual `y`s.



This particular plot was made using the `matplotlib` module, with the following code added to the end of `main()` (outside the loop):

```
import matplotlib.pyplot as plt
ratio = y / f(x, fit.pmean)
plt.xlim(4, 21)
plt.xlabel('x')
plt.ylabel('y / f(x,p)')
plt.errorbar(x=x, y=gv.mean(ratio), yerr=gv.sdev(ratio), fmt='ob')
plt.plot([4.0, 21.0], [1.0, 1.0], 'b:')
plt.show()
```

Making Fake Data: Function `make_data()` creates a list of `x` values, evaluates the underlying function, `f_exact(x)`, for those values, and then adds random noise to the results to create the `y` array of fit data: `y = f_exact(x) * noise` where

```
noise = 1 + sum_n=0..99 c[n] * (x/x_max) ** n
```

Here the `c[n]` are random coefficients generated using the following code:

```
cr = gv.gvar(0.0, eps)
c = [gv.gvar(cr(), eps) for n in range(100)]
```

Gaussian variable `cr` represents a Gaussian distribution with mean 0.0 and width 0.01, which we use here as a random number generator: `cr()` is a number drawn randomly from the distribution represented by `cr`:

```
>>> print(cr)
0.000(10)
>>> print(cr())
0.00452180208286
>>> print(cr())
-0.00731564589737
```

We use `cr()` to generate mean values for the Gaussian distributions represented by the `c[n]`s, each of which has width 0.01. The resulting `y`s fluctuate around the corresponding values of `f_exact(x)`:

```
>>> print(y-f_exact(x))
[0.0011(27) 0.00029(80) ... ]
```

The Gaussian variables `y[i]` together with the numbers `x[i]` comprise our fake data.

1.4 Chained Fits

The priors in a fit represent knowledge that we have about the parameters before we do the fit. This knowledge might come from theoretical considerations or experiment. Or it might come from another fit. Imagine that we want to add new information to that extracted from the fit in the previous section. For example, we might learn from some other source that the ratio of amplitudes `a[1]/a[0]` equals 1±1e-5. The challenge is to combine this new information with information extracted from the fit above without rerunning that fit. (We assume it is not possible to rerun the first fit, because, say, the input data for that fit has been lost or is unavailable.)

We can combine the new data with the old fit results by creating a new fit using the best-fit parameters, `fit.p`, from the old fit as the priors for the new fit. To try this out, we add the following code onto the end of the `main()` subroutine in the previous section, keeping `nexp<=4` (since that is all that is needed):

```
def ratio(p):                       # new fit function
    a = p['a']
    return a[1] / a[0]

prior = fit.p                       # prior = best-fit parameters from 1st fit
data = gv.gvar(1, 1e-5)             # new data for the ratio
```

```
newfit = lsqfit.nonlinear_fit(data=data, fcn=ratio, prior=prior)
print(newfit)
E = newfit.p['E']
a = newfit.p['a']
print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
```

The result of the new fit (to one piece of new data) is:

```
Least Square Fit:
  chi2/dof [dof] = 1.3 [1]    Q = 0.25    logGBF = 2.3618

Parameters:
            a 0    0.4018 (40)       [  0.4020 (40) ]
              1    0.4018 (40)       [  0.4110 (90) ]  *
              2      0.37 (29)       [    0.59 (35) ]
              3      0.49 (40)       [    0.50 (40) ]
            E 0   0.90030 (51)       [ 0.90045 (53) ]
              1    1.8000 (12)       [  1.8055 (50) ]  *
              2      2.69 (13)       [    2.85 (19) ]
              3      4.01 (40)       [    4.00 (40) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 3/0.0)

E1/E0 = 1.9993(11)   E2/E0 = 2.99(14)
a1/a0 = 1.000000(10)   a2/a0 = 0.91(73)
```

Parameters `a[0]` and `E[0]` are essentially unchanged by the new information, but `a[i]` and `E[i]` are more precise for `i=1` and 2, as is `a[1]/a[0]`, of course.

It might seem odd that `E[1]`, for example, is changed at all, since the fit function, `ratio(p)`, makes no mention of it. This is not surprising, however, since `ratio(p)` does depend upon `a[1]`, and `a[1]` is strongly correlated with `E[1]` through the prior. It is important to include all parameters from the first fit as parameters in the new fit in order to capture the impact of the new information on parameters correlated with `a[1]/a[0]`.

It would have been easy to change the fit code in the previous section to incorporate the new information about `a[1]/a[0]`. The approach presented here is numerically equivalent to that approach insofar as the `chi**2` function for the original fit can be well approximated by a quadratic function in the fit parameters. This is, of course, a fundamental assumption underlying the use of *`lsqfit`* in the first place.

Obviously, we can include further fits in order to incorporate more data. The prior for each new fit is the best-fit output (`fit.p`) from the previous fit. The output from the chain's final fit is the cummulative result of all of these fits.

Finally note that this particular problem can be done much more simply using a weighted average (*`lsqfit.wavg()`*). Adding the following code onto the end of the `main()` subroutine in the previous section

```
fit.p['a1/a0'] = fit.p['a'][1] / fit.p['a'][0]
new_data = {'a1/a0' : gv.gvar(1,1e-5)}
new_p = lsqfit.wavg([fit.p, new_data])

print('chi2/dof = %.2f\n' % new_p.chi2 / new_p.dof)
print('E:', new_p['E'][:4])
print('a:', new_p['a'][:4])
print('a1/a0:', new_p['a1/a0'])
```

gives the following output:

```
chi2/dof = 1.30

E: [0.90030(51) 1.8000(12) 2.69(13) 4.01(40)]
a: [0.4018(40) 0.4018(41) 0.36(29) 0.49(40)]
a1/a0: 1.000000(10)
```

Here we do a weighted average of `a[1]/a[0]` from the original fit (`fit.p['a1/a0']`) with our new piece of data (`new_data['a1/a0']`). The dictionary `new_p` returned by *`lsqfit.wavg()`* has an entry for every key in either `fit.p` or `new_data`. The weighted average for `a[1]/a[0]` is in `new_data['a1/a0']`. New values for the fit parameters, that take account of the new data, are stored in `new_p['E']` and `new_p['a']`. The `E[i]` and `a[i]` estimates differ from their values in `fit.p` since those parameters are correlated with `a[1]/a[0]`. Consequently when the ratio is shifted by new data, the `E[i]` and `a[i]` are shifted as well. The final results in `new_p` are almost identical to what we obtained above; this is because the errors are sufficiently small that the ratio `a[1]/a[0]` is Gaussian.

1.5 Bayesian Integrals

The least-squares fit analyses above implicitly assume that every probability distribution relevant to the fits is well approximated by a Gaussian distribution. The key distribution in a fit is the probability density function for the parameters, which is proportional to `exp(-chi**2/2)` in a full Bayesian analysis. (Here the `chi**2` function is treated as a function of the parameters and has contributions from both the data and the priors.) Bayesian estimates for the fit parameters are the expectation values of those parameters calculated with this probability density function. These will agree with the best-fit results of our least-squares fits provided `chi**2` is well approximated by its quadratic expansion in the parameters — that is, insofar as `exp(-chi**2/2)` is well approximated by the Gaussian distribution in the parameters specified by their best-fit means and covariance matrix (in `fit.p`). *`lsqfit.BayesIntegrator`* can be used to evaluate Bayesian expectation values when the Gaussian approximation breaks down.

`lsqfit.BayesIntegrator` uses the `vegas` module for adaptive multi-dimensional integration to evaluate expectation values by integrating arbitrary functions of the parameters, mutliplied by the exact probability density function, over the entire parameter space. (Module `vegas` must be installed for *`lsqfit.BayesIntegrator`*.) To illustrate its use we will redo the fit in our discussion of *Basic Fits* using Bayesian integration. To keep things simple we will focus just on the parameters `a[0]` and `E[0]` that describe the leading exponential.

The following code, added to the end of the `main()` function in *Basic Fits* (outside the loop), evaluates the Bayesian expectation value of function `g(p)` of the fit parameters:

```
# fit is from the last least-squares fit
expval = lsqfit.BayesIntegrator(fit, limit=10.)

# adapt integrator to PDF from fit
expval(neval=10000, nitn=10)

# calculate expectation value of function g(p)
E0_hist = gv.PDFHistogram(fit.p['E'][0])

def g(p):
    parameters = [p['a'][0], p['E'][0]]
    return dict(
        mean=parameters,
        outer=np.outer(parameters, parameters),
        hist=E0_hist.count(p['E'][0]),
        )

results = expval(g, neval=10000, nitn=10, adapt=False)
```

```
# analyze results and compare with least-squares fit results
print(results.summary())
mean = results['mean']
cov = results['outer'] - np.outer(results['mean'], results['mean'])

print('Results from Bayesian Integration:')
print('a0: mean =', mean[0], '  sdev =', cov[0,0]**0.5)
print('E0: mean =', mean[1], '  sdev =', cov[1,1]**0.5)
print(
    'covariance from Bayesian integral =',
    np.array2string(cov, prefix=36 * ' ')
    )
print()

print('Results from Least-Squares Fit:')
print('a0: mean =', fit.p['a'][0].mean, '  sdev =', fit.p['a'][0].sdev)
print('E0: mean =', fit.p['E'][0].mean, '  sdev =', fit.p['E'][0].sdev)
print(
    'covariance from least-squares fit =',
    np.array2string(
        gv.evalcov([fit.p['a'][0], fit.p['E'][0]]),
        prefix=36 * ' ', precision=3
        )
    )

# make histogram of E[0] probabilty
plt = E0_hist.make_plot(results['hist'])
plt.xlabel('$E_0$')
plt.show()
```

Here `expval` is an integrator that is used to evaluate expectation values of arbitrary functions of the fit parameters. `BayesIntegrator` uses output (`fit`) from a least-squares fit to design a `vegas` integrator for calculating expectation values. The integrator uses an iterative Monte Carlo algorithm that adapts to the probability density function after each iteration. See the `vegas` documentation for much more information.

We first call the integrator without a function. This allows it to adapt to the probability density function from the fit without the extra overhead of evaluating some function of the parameters. The integrator uses `nitn=10` iterations of the `vegas` algorithm, with at most `neval=10000` evaluations of the probability density function for each iteration.

We then use the optimized integrator to evaluate the expectation value of function `g(p)` (turning adaptation off with `adapt=False`). The expectation value of `g(p)` is returned in dictionary `results`, where `results[k]` is the expectation value of `g(p)[k]` for each key used in the dictionary returned by `g(p)`. Thus `results[mean]`, for example, contains the expectation value of `[a[0], E[0]]` (*i.e.*, the mean value). We also compute the expectation value of outer product of `[a[0], E[0]]` with itself, so we can compute the covariance matrix for `a[0]` and `E[0]`. Finally we accumulate histogram data, using `PDFHistogram` from the `gvar` module, for `E[0]`.

The output from this codes is (for `nexp=4`):

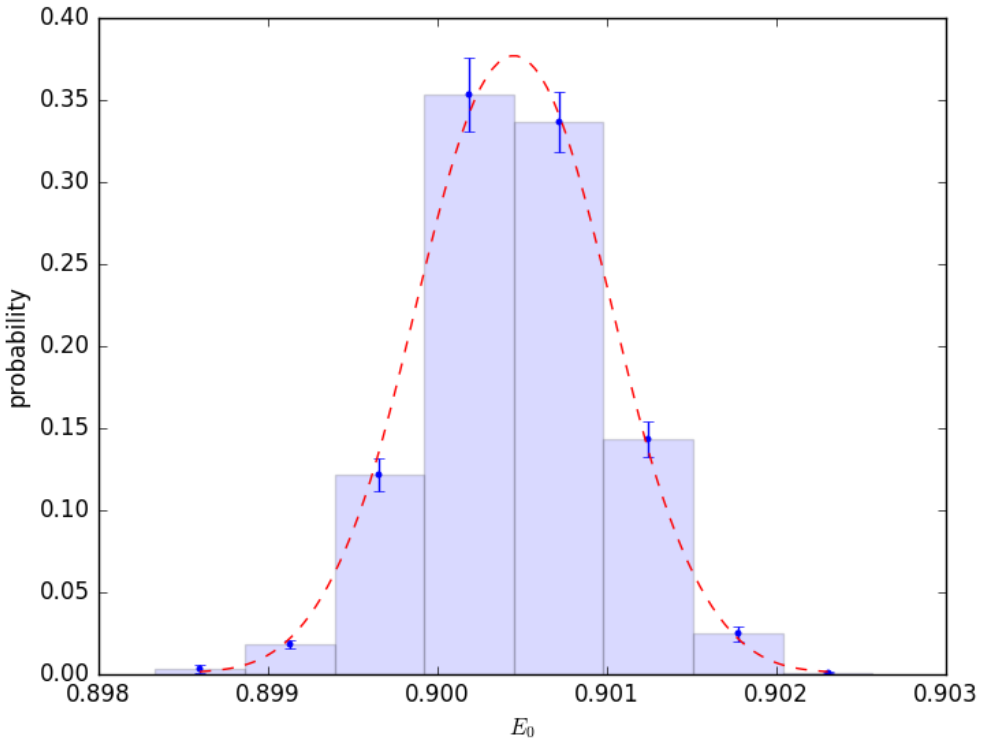
```
itn   integral        average         chi2/dof        Q
-------------------------------------------------------
  1   0.275(38)       0.275(38)           0.00     1.00
  2   0.273(27)       0.274(23)           0.60     0.90
  3   0.232(22)       0.260(17)           0.63     0.95
  4   0.232(15)       0.253(13)           0.74     0.91
  5   0.274(55)       0.257(15)           0.81     0.87
  6   0.287(43)       0.262(15)           0.84     0.86
  7   0.283(47)       0.265(14)           0.90     0.76
  8   0.257(27)       0.264(13)           0.85     0.89
  9   0.205(13)       0.258(12)           0.85     0.90
```

```
 10    0.259(23)       0.258(11)           0.86      0.90

Results from Bayesian Integration:
a0: mean = 0.40210(16)    sdev = 0.00401(11)
E0: mean = 0.900460(19)    sdev = 0.000528(31)
covariance from Bayesian integral = [[0.00001610(87) -1.5(5.8)e-08]
                                     [-1.5(5.8)e-08 2.79(32)e-07]]

Results from Least-Squares Fit:
a0: mean = 0.40197397247    sdev = 0.00401779882449
E0: mean = 0.900453307997    sdev = 0.000529523389062
covariance from least-squares fit = [[  1.614e-05   2.166e-08]
                                     [  2.166e-08   2.804e-07]]
```

The table shows results from each of the `nitn=10` `vegas` iterations used to evaluate the expectation values. Estimates for the integral of the probability density function are listed for each iteration. (Results from the integrator are approximate, with error estimates.) These are consistent with each other and with the (more accurate) overall average. The table shows that the probability density function is normalized to 0.258(11) rather than to unity. This indicates that the distribution is not quite Gaussian, although 0.258 is fairly close to 1 for an 8-dimensional distribution (the 1/8-th root of 0.258 is 0.84).

The agreement is excellent between the mean values and covariances calculated using Bayesian integrals, and those directly from the least-squares fit. This suggests that the Gaussian approximation is reliable for this problem. The Bayesian integral, over all 8 parameters, takes much more time (15–20 seconds on a 2014 laptop) than the least-squares fit (a small fraction of a second), so the latter is much more efficient when the Gaussian approximation is sufficiently accurate.

As a further check on the Gaussian approximation we also accumulated data (`results['hist']`) on the probability distribution of `E[0]` values, integrating out the other 7 parameters. The plot produced by the code above shows that posterior probability distribution for `E[0]` (bars) agrees well with that implied by `fit.p['E'][0]` (dashed line):



The Bayesian integrals are relatively simple in this example. More complicated problems can require much more

computer time to evaluate the integrals, with hundreds of thousands or millions of integrand evaluations per iteration (`neval`). This is particularly true as the number of parameters increases. `BayesIntegrator` uses information from the least-squares fit to simplify the integration for `vegas` by optimizing the integration variables used, but integrals over tens of variables are intrinsically challenging. `BayesIntegrator` can be used with MPI to run such integrals on multiple processors, for a considerable speed-up.

1.6 `x` has Error Bars

We now consider variations on our basic fit analysis (described in *Basic Fits*). The first variation concerns what to do when the independent variables, the `x`s, have errors, as well as the `y`s. This is easily handled by turning the `x`s into fit parameters, and otherwise dispensing with independent variables.

To illustrate this, we modify the basic analysis code above. First we need to add errors to the `x`s, which we do by changing `make_data` so that each `x` has a random value within about $\pm 0.001\%$ of its original value and an error:

```
def make_data(nexp=100, eps=0.01): # make x, y fit data
    x = np.array([5., 6., 7., 8., 9., 10., 12., 14., 16., 18., 20.])
    cr = gv.gvar(0.0, eps)
    c = [gv.gvar(cr(), eps) for n in range(100)]
    x_xmax = x/max(x)
    noise = 1+ sum(c[n] * x_xmax ** n for n in range(100))
    y = f_exact(x, nexp) * noise            # noisy y[i]s
    xfac = gv.gvar(1.0, 0.00001)    # Gaussian distrib'n: 1±0.001%
    x = np.array([xi * gv.gvar(xfac(), xfac.sdev) for xi in x]) # noisy x[i]s
    return x, y
```

Here `gvar.GVar` object `xfac` is used as a random number generator: each time it is called, `xfac()` is a different random number from the distribution with mean `xfac.mean` and standard deviation `xfac.sdev` (that is, 1 ± 0.00001). The main program is modified so that the (now random) `x` array is treated as a fit parameter. The prior for each `x` is, obviously, specified by the mean and standard deviation of that `x`, which is read directly out of the array of `x`s produced by `make_data()`:

```
def make_prior(nexp, x):            # make priors for fit parameters
    prior = gv.BufferDict()         # prior -- any dictionary works
    prior['a'] = [gv.gvar(0.5, 0.5) for i in range(nexp)]
    prior['E'] = [gv.gvar(i+1, 0.5) for i in range(nexp)]
    prior['x'] = x                  # x now an array of parameters
    return prior

def main():
    gv.ranseed([2009, 2010, 2011, 2012]) # initialize random numbers (opt.)
    x, y = make_data()              # make fit data
    p0 = None                       # make larger fits go faster (opt.)
    for nexp in range(1, 10):
        print('************************************ nexp =', nexp)
        prior = make_prior(nexp, x)
        fit = lsqfit.nonlinear_fit(data=y, fcn=f, prior=prior, p0=p0)
        print(fit)                  # print the fit results
        if nexp > 2:
            E = fit.p['E']              # best-fit parameters
            a = fit.p['a']
            print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
            print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
        if fit.chi2/fit.dof<1.:
            p0 = fit.pmean          # starting point for next fit (opt.)
        print()
```

The fit data now consists of just the `y` array (`data=y`), and the fit function loses its `x` argument and gets its `x` values from the fit parameters `p` instead:

```
def f(p):
    a = p['a']
    E = p['E']
    x = p['x']
    return sum(ai*exp(-Ei*x) for ai, Ei in zip(a, E))
```

Running the new code gives, for `nexp=4` terms:

```
************************************ nexp = 4
Least Square Fit:
  chi2/dof [dof] = 0.97 [11]    Q = 0.47    logGBF = 167.6

Parameters:
            a 0      0.3982 (41)        [     0.50 (40) ]
              1       0.423 (94)        [     0.50 (40) ]
              2        0.50 (40)        [     0.50 (40) ]
              3        0.50 (40)        [     0.50 (40) ]
            E 0     0.90028 (68)        [     1.00 (40) ]
              1       1.810 (57)        [     2.00 (40) ]
              2        3.01 (40)        [     3.00 (40) ]
              3        4.00 (40)        [     4.00 (40) ]
            x 0    5.000046 (50)        [ 5.000045 (50) ]
              1    5.999982 (56)        [ 5.999985 (60) ]
              2    6.999966 (47)        [ 6.999990 (70) ]
              3    7.999968 (52)        [ 7.999926 (80) ]
              4    8.999973 (52)        [ 8.999860 (90) ]  *
              5    9.999976 (63)        [ 10.00018 (10) ]  **
              6   11.999985 (84)        [ 11.99994 (12) ]
              7    14.00002 (11)        [ 13.99995 (14) ]
              8    16.00010 (15)        [ 16.00014 (16) ]
              9    18.00018 (18)        [ 18.00018 (18) ]
             10    19.99957 (20)        [ 19.99957 (20) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 2/0.0)

E1/E0 = 2.011(62)   E2/E0 = 3.34(44)
a1/a0 = 1.06(23)   a2/a0 = 1.2(1.0)
```

This looks quite a bit like what we obtained before, except that now there are 11 more parameters, one for each `x`, and also now all results are a good deal less accurate. Note that one result from this analysis is new values for the `x`s. In some cases (*e.g.*, `x[4]` and `x[5]`), the errors on the `x` values have been reduced — by information in the fit data.

1.7 Correlated Parameters; Gaussian Bayes Factor

`gvar.GVar` objects are very useful for handling more complicated priors, including situations where we know *a priori* of correlations between parameters. Returning to the *Basic Fits* example above, imagine a situation where we still have a ± 0.4 uncertainty about the value of any individual `E[i]`, but we know *a priori* that the separations between adjacent `E[i]`s is 0.9 ± 0.01. We want to build the correlation between adjacent `E[i]`s into our prior.

We do this by introducing a `gvar.GVar` object `de[i]` for each separate difference `E[i]-E[i-1]`, with `de[0]` being `E[0]`:

```
de = [gvar(0.9, 0.01) for i in range(nexp)]
de[0] = gvar(1, 0.5)     # different distribution for E[0]
```

Then `de[0]` specifies the probability distribution for `E[0]`, `de[0]+de[1]` the distribution for `E[1]`, `de[0]+de[1]+de[2]` the distribution for `E[2]`, and so on. This can be implemented (slightly inefficiently) in a single line of Python:

```
E = [sum(de[:i+1]) for i in range(nexp)]
```

For `nexp=3`, this implies that

```
>>> print(E)
[1.00(40) 1.90(40) 2.80(40)]
>>> print(E[1] - E[0], E[2] - E[1])
0.900(10) 0.900(10)
```

which shows that each `E[i]` separately has an uncertainty of ± 0.4 (approximately) but that differences are specified to within ± 0.01.

In the code, we need only change the definition of the prior in order to introduce these correlations:

```
def make_prior(nexp):                # make priors for fit parameters
    prior = gv.BufferDict()          # prior -- any dictionary works
    prior['a'] = [gv.gvar(0.5, 0.4) for i in range(nexp)]
    de = [gv.gvar(0.9, 0.01) for i in range(nexp)]
    de[0] = gv.gvar(1, 0.4)
    prior['E'] = [sum(de[:i + 1]) for i in range(nexp)]
    return prior
```

Running the code as before, but now with the correlated prior in place, we obtain the following fit with `nexp=5` terms:

```
************************************ nexp = 5
Least Square Fit:
  chi2/dof [dof] = 0.56 [11]    Q = 0.87    logGBF = 186.47

Parameters:
            a 0    0.4019 (40)      [  0.50 (40) ]
              1    0.4057 (71)      [  0.50 (40) ]
              2     0.370 (48)      [  0.50 (40) ]
              3      0.49 (40)      [  0.50 (40) ]
              4      0.50 (40)      [  0.50 (40) ]
            E 0   0.90041 (53)      [  1.00 (40) ]
              1    1.8029 (41)      [  1.90 (40) ]
              2     2.703 (11)      [  2.80 (40) ]
              3     3.603 (15)      [  3.70 (40) ]
              4     4.503 (18)      [  4.60 (40) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 2/0.0)

E1/E0 = 2.0023(42)   E2/E0 = 3.002(12)
a1/a0 = 1.009(14)   a2/a0 = 0.92(12)
```

The results are similar to before for the leading parameters, but substantially more accurate for parameters describing the second and later exponential terms, as might be expected given our enhanced knowledge about the differences between `E[i]`s. The output energy differences are particularly accurate: they range from `E[1]-E[0] = 0.9025(39)`, which is almost three times more precise than the prior, to `E[6]-E[5] = 0.900(10)`, which is just what was put into the fit through the prior (the fit data adds no new information). The correlated prior allows us to merge our *a priori* information about the energy differences with the new information carried by the fit data `x, y`.

Note that the Gaussian Bayes Factor (see `logGBF` in the output) is larger with the correlated prior (`logGBF = 186.5`) than it was for the uncorrelated prior (`logGBF = 182.4`). Had we been uncertain as to which prior was more appropriate, this difference says that the data prefers the correlated prior. (More precisely, it says that we would be `exp(186.5-182.4) = 60` times more likely to get our `x,y` data from a theory with the correlated prior than from one with the uncorrelated prior.) This difference is significant despite the fact that the `chi**2`s in the two cases are almost the same. `chi**2` tests goodness of fit, but there are usually more ways than one to get a good fit. Some are more plausible than others, and the Bayes factor helps sort out which.

The Gaussian Bayes Factor is an approximation to the *Bayes Factor* which is valid in the limit where all distributions can be approximated by Gaussians. The Bayes Factor is the probability (density) that the fit data would be generated randomly from the fit function and priors (the *model*) used in the fit. Ratios of Bayes Factors from fits with different models tell us about the relative likelihood of the different models given the data. (Actually the ratio gives the ratio of probabilities for obtaining the data from the models, as opposed to the probabilities for the models given the data. The latter ratio is the same as the former, however, provided the two models are equally likely *a priori* (from Bayes Theorem).)

1.8 Tuning Priors and the Empirical Bayes Criterion

Given two choices of prior for a parameter, the one that results in a larger Gaussian Bayes Factor after fitting (see `logGBF` in fit output or `fit.logGBF`) is the one preferred by the data. We can use this fact to tune a prior or set of priors in situations where we are uncertain about the correct *a priori* value: we vary the widths and/or central values of the priors of interest to maximize `logGBF`. This leads to complete nonsense if it is applied to all the priors, but it is useful for tuning (or testing) limited subsets of the priors when other information is unavailable. In effect we are using the data to get a feel for what is a reasonable prior. This procedure for setting priors is called the *Empirical Bayes* method.

This method is implemented in a driver program

```
fit, z = lsqfit.empbayes_fit(z0, fitargs)
```

which varies `numpy` array `z`, starting at `z0`, to maximize `fit.logGBF` where

```
fit = lsqfit.nonlinear_fit(**fitargs(z)).
```

Function `fitargs(z)` returns a dictionary containing the arguments for `nonlinear_fit()`. These arguments, and the prior in particular, are varied as some function of `z`. The optimal fit (that is, the one for which `fit.logGBF` is maximum) and `z` are returned.

To illustrate, consider tuning the widths of the priors for the amplitudes, `prior['a']`, in the example from the previous section. This is done by adding the following code to the end of `main()` subroutine:

```
def fitargs(z, nexp=nexp, prior=prior, f=f, data=(x, y), p0=p0):
    z = np.exp(z)
    prior['a'] = [gv.gvar(0.5, 0.5 * z[0]) for i in range(nexp)]
    return dict(prior=prior, data=data, fcn=f, p0=p0)
##
z0 = [0.0]
fit, z = empbayes_fit(z0, fitargs, tol=1e-3)
print(fit)                  # print the optimized fit results
E = fit.p['E']              # best-fit parameters
a = fit.p['a']
print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
print("prior['a'] =", fit.prior['a'][0])
print()
```

Function `fitargs` generates a dictionary containing the arguments for *`lsqfit.nonlinear_fit`*. These are identical to what we have been using except that the width of the priors in `prior['a']` is adjusted according to parameter `z`. Function *`lsqfit.empbayes_fit()`* does fits for different values of `z` and selects the `z` that maximizes `fit.logGBF`. It returns the corresponding fit and the value of `z`.

This code generates the following output when `nexp=4`:

```
Least Square Fit:
  chi2/dof [dof] = 0.81 [11]    Q = 0.63    logGBF = 188.78

Parameters:
            a 0    0.4023 (40)      [  0.50 (11) ]
              1    0.4040 (67)      [  0.50 (11) ]
              2     0.388 (42)      [  0.50 (11) ]  *
              3      0.50 (11)      [  0.50 (11) ]
            E 0   0.90036 (52)      [  1.00 (50) ]
              1    1.8014 (38)      [  1.90 (50) ]
              2     2.703 (11)      [  2.80 (50) ]
              3     3.603 (15)      [  3.70 (50) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 1/0.0)

E1/E0 = 2.0008(39)   E2/E0 = 3.002(12)
a1/a0 = 1.004(13)   a2/a0 = 0.96(10)
```

Reducing the width of the `prior['a']`s from 0.5 to 0.1 increased `logGBF` from 186.5 to 188.8. The error for `a2/a0` is 20% smaller, but the other results are less affected — suggesting that the details of `prior['a']` are not all that important, which is confirmed by the error budgets generated in the next section. It is not surprising, of course, that the optimal width is 0.1 since the mean values for the `fit.p['a']`s are clustered around 0.4, which is 0.1 below the mean value of the priors `prior['a']`.

The Bayes factor, `exp(fit.logGBF)`, is useful for deciding about fit functions as well as priors. Consider the following two fits of the sort discussed in the previous section, one using just two terms in the fit function and one using three terms:

```
************************************* nexp = 2
Least Square Fit:
  chi2/dof [dof] = 0.58 [11]    Q = 0.84    logGBF = 187.74

Parameters:
            a 0    0.4018 (40)      [  0.50 (50) ]
              1    0.4017 (40)      [  0.50 (50) ]
            E 0   0.90032 (51)      [  1.00 (50) ]
              1   1.80017 (73)      [  1.90 (50) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 5/0.0)


************************************* nexp = 3
Least Square Fit:
  chi2/dof [dof] = 0.64 [11]    Q = 0.79    logGBF = 185.09

Parameters:
            a 0    0.4019 (40)      [  0.50 (50) ]
              1    0.4051 (70)      [  0.50 (50) ]
              2    -0.029 (49)      [  0.50 (50) ]  *
            E 0   0.90040 (53)      [  1.00 (50) ]
```

```
              1    1.8025 (41)       [  1.90 (50) ]
              2     2.702 (11)       [  2.80 (50) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 4/0.0)
```

Measured by their `chi**2`s, the two fits are almost equally good. The Bayes factor for the first fit, however, is larger than that for the second fit. It says that the probability that our fit data comes from an underlying theory with exactly two terms is `exp(187.74 - 185.09) = 14` times larger than the probability that it comes from a theory with three terms. In fact, the data comes from a theory with only two terms since it was generated using the same code as in the previous section but with `x, y = make_data(2)` instead of `x, y = make_data()` in the main program.

1.9 Partial Errors and Error Budgets

We frequently want to know how much of the uncertainty in a fit result is due to a particular input uncertainty or subset of input uncertainties (from the input data and/or from the priors). We refer to such errors as "partial errors" (or partial standard deviations) since each is only part of the total uncertainty in the fit result. The collection of such partial errors, each associated with a different input error, is called an "error budget" for the fit result. The partial errors from all sources of input error reproduce the total fit error when they are added in quadrature.

Given the `fit` object (an *`lsqfit.nonlinear_fit`* object) from the example in the section on *Correlated Parameters; Gaussian Bayes Factor*, for example, we can extract such information using `gvar.GVar.partialsdev()` — for example:

```
>>> E = fit.p['E']
>>> a = fit.p['a']
>>> print(E[1] / E[0])
2.0023(42)
>>> print((E[1] / E[0]).partialsdev(fit.prior['E']))
0.00166261727498
>>> print((E[1] / E[0]).partialsdev(fit.prior['a']))
0.000472922525865
>>> print((E[1] / E[0]).partialsdev(y))
0.00387452586852
```

This shows that the total uncertainty in `E[1]/E[0]`, 0.0042, is the sum in quadrature of a contribution 0.0017 due to the priors specified by `prior['E']`, 0.0005 due to `prior['a']`, and 0.0039 from the statistical errors in the input data `y`.

There are two utility functions for tabulating results and error budgets. They require dictionaries of output results and inputs, and use the keys from the dictionaries to label columns and rows, respectively, in an error-budget table:

```
outputs = {
          'E1/E0':E[1] / E[0], 'E2/E0':E[2] / E[0],
          'a1/a0':a[1] / a[0], 'a2/a0':a[2] / a[0],
          }
inputs = {'E':fit.prior['E'], 'a':fit.prior['a'], 'y':y}
print(fit.fmt_values(outputs))
print(fit.fmt_errorbudget(outputs, inputs))
```

This gives the following output:

```
Values:
              E2/E0: 3.002(12)
              E1/E0: 2.0023(42)
              a2/a0: 0.92(12)
              a1/a0: 1.009(14)
```

```
Partial % Errors:
               E2/E0     E1/E0     a2/a0     a1/a0
-------------------------------------------------
        a:      0.01      0.02      4.56      0.19
        y:      0.13      0.19     10.95      1.27
        E:      0.38      0.08      5.68      0.55
-------------------------------------------------
    total:      0.40      0.21     13.15      1.40
```

This table shows, for example, that the 0.40% uncertainty in `E2/E0` comes from a 0.01% contribution due to `prior['a']`, a 0.13% contribution due to due to statistical errors in the fit data `y`, and a 0.38% contribution due to `prior['E']`, where, again, the total error is the sum in quadrature of the partial errors. This suggests that reducing the statistical errors in the input `y` data would reduce the error in `E2/E0` only slightly. On the other hand, more accurate `y` data should significantly reduce the errors in `E1/E0` and `a1/a0`, where `y` is the dominant source of uncertainty. In fact a four-fold reduction in the `y` errors reduces the `E1/E0` error to 0.06% (from 0.21%) while leaving the `E2/E0` error at 0.38%.

1.10 `y` has No Error Bars

Occasionally there are fit problems where values for the dependent variable `y` are known exactly (to machine precision). This poses a problem for least-squares fitting since the `chi**2` function is infinite when standard deviations are zero. How does one assign errors to exact `y`s in order to define a `chi**2` function that can be usefully minimized?

It is almost always the case in physical applications of this sort that the fit function has in principle an infinite number of parameters. It is, of course, impossible to extract information about infinitely many parameters from a finite number of `y`s. In practice, however, we generally care about only a few of the parameters in the fit function. (If this isn't the case, give up.) The goal for a least-squares fit is to figure out what a finite number of exact `y`s can tell us about the parameters we want to know.

The key idea here is to use priors to model the part of the fit function that we don't care about, and to remove that part of the function from the analysis by subtracting or dividing it out from the input data. To illustrate, consider again the example described in the section on *Correlated Parameters; Gaussian Bayes Factor*. Let us imagine that we know the exact values for `y` for each of `x=1, 1.2, 1.4...2.6, 2.8`. We are fitting this data with a sum of exponentials `a[i]*exp(-E[i]*x)` where now we will assume that *a priori* we know that: `E[0]=1.0(5)`, `E[i+1]-E[i]=0.9(2)`, and `a[i]=0.5(5)`. Suppose that our goal is to find good estimates for `E[0]` and `a[0]`.

We know that for some set of parameters

```
y = sum_i=0..inf  a[i]*exp(-E[i]*x)
```

for each `x`-`y` pair in our fit data. Given that `a[0]` and `E[0]` are all we want to know, we might imagine defining a new, modified dependent variable `ymod`, equal to just `a[0]*exp(-E[0]*x)`:

```
ymod = y - sum_i=1..inf a[i]*exp(-E[i]*x)
```

We know everything on the right-hand side of this equation: we have exact values for `y` and we have *a priori* estimates for the `a[i]` and `E[i]` with `i>0`. So given means and standard deviations for every `i>0` parameter, and the exact `y`, we can determine a mean and standard deviation for `ymod`. The strategy then is to compute the corresponding `ymod` for every `y` and `x` pair, and then fit `ymod` versus `x` to the *single* exponential `a[0]*exp(-E[0]*t)`. That fit will give values for `a[0]` and `E[0]` that reflect the uncertainties in `ymod`, which in turn originate in uncertainties in our knowledge about the parameters for the `i>0` exponentials.

It turns out to be quite simple to implement such a strategy using `gvar.GVar`s. We convert our code by first modifying the main program so that it provides prior information to a subroutine that computes `ymod`. We will vary the

number of terms `nexp` that are kept in the fit, putting the rest into `ymod` as above (up to a maximum of 20 terms, which is close enough to infinity):

```
def main():
    gv.ranseed([2009, 2010, 2011, 2012])  # initialize random numbers (opt.)
    max_prior = make_prior(20)          # maximum sized prior
    p0 = None                           # make larger fits go faster (opt.)
    for nexp in range(1, 7):
        print('************************************ nexp =', nexp)
        fit_prior = gv.BufferDict()     # part of max_pior used in fit
        ymod_prior = gv.BufferDict()    # part of max_prior absorbed in ymod
        for k in max_prior:
            fit_prior[k] = max_prior[k][:nexp]
            ymod_prior[k] = max_prior[k][nexp:]
        x, y = make_data(ymod_prior)    # make fit data
        fit = lsqfit.nonlinear_fit(data=(x, y), fcn=f, prior=fit_prior, p0=p0)
        print(fit.format(maxline=True))          # print the fit results
        print()
        if fit.chi2/fit.dof<1.:
            p0 = fit.pmean              # starting point for next fit (opt.)
```

We put all of our *a priori* knowledge about parameters into prior `max_prior` and then pull out the part we need for the fit — that is, the first `nexp` terms. The remaining part of `max_prior` is used to correct the exact data, which comes from a new `make_data`:

```
def make_data(ymod_prior):          # make x, y fit data
    x = np.arange(1., 10 * 0.2 +  1., 0.2)
    ymod = f_exact(x) - f(x, ymod_prior)
    return x, ymod
```

Running the new code produces the following output, where again `nexp` is the number of exponentials kept in the fit (and `20-nexp` is the number pushed into the modified dependent variable `ymod`):

```
************************************ nexp = 1
Least Square Fit:
  chi2/dof [dof] = 0.051 [10]    Q = 1    logGBF = 97.499

Parameters:
            a 0    0.4009 (14)       [  0.50 (50) ]
            E 0   0.90033 (62)       [  1.00 (50) ]

Fit:
     x[k]           y[k]        f(x[k],p)
----------------------------------------
        1      0.15 (11)     0.16292 (47)
      1.2     0.128 (74)     0.13607 (38)
      1.4     0.110 (52)     0.11365 (30)
      1.6     0.093 (37)     0.09492 (24)
      1.8     0.078 (26)     0.07928 (19)
        2     0.066 (18)     0.06622 (15)
      2.2     0.055 (13)     0.05531 (12)
      2.4    0.0462 (93)    0.046192 (94)
      2.6    0.0387 (66)    0.038581 (74)
      2.8    0.0323 (47)    0.032223 (58)

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0*    (itns/time = 5/0.0)
```

```
************************************ nexp = 2
Least Square Fit:
  chi2/dof [dof] = 0.053 [10]    Q = 1    logGBF = 99.041

Parameters:
            a 0    0.4002 (13)      [  0.50 (50) ]
              1     0.405 (36)      [  0.50 (50) ]
            E 0   0.90006 (55)      [  1.00 (50) ]
              1     1.803 (30)      [  1.90 (54) ]

Fit:
     x[k]            y[k]        f(x[k],p)
----------------------------------------
        1      0.223 (45)      0.2293 (44)
      1.2      0.179 (26)      0.1823 (28)
      1.4      0.145 (15)      0.1459 (18)
      1.6     0.1168 (90)      0.1174 (12)
      1.8     0.0947 (53)     0.09492 (74)
        2     0.0770 (32)     0.07711 (47)
      2.2     0.0628 (19)     0.06289 (30)
      2.4     0.0515 (11)     0.05148 (19)
      2.6    0.04226 (67)     0.04226 (12)
      2.8    0.03479 (40)    0.034784 (72)

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0*    (itns/time = 3/0.0)


************************************ nexp = 3
Least Square Fit:
  chi2/dof [dof] = 0.057 [10]    Q = 1    logGBF = 99.844

Parameters:
            a 0   0.39998 (93)      [  0.50 (50) ]
              1     0.399 (35)      [  0.50 (50) ]
              2     0.401 (99)      [  0.50 (50) ]
            E 0   0.89999 (36)      [  1.00 (50) ]
              1     1.799 (26)      [  1.90 (54) ]
              2      2.70 (20)      [  2.80 (57) ]

Fit:
     x[k]             y[k]        f(x[k],p)
-----------------------------------------
        1       0.253 (19)      0.2557 (54)
      1.2      0.1968 (91)      0.1977 (28)
      1.4      0.1545 (45)      0.1548 (15)
      1.6      0.1224 (22)     0.12256 (75)
      1.8      0.0979 (11)     0.09793 (39)
        2     0.07885 (54)     0.07886 (20)
      2.2     0.06391 (27)     0.06391 (10)
      2.4     0.05206 (13)    0.052065 (51)
      2.6    0.042602 (67)    0.042601 (26)
      2.8    0.034983 (33)    0.034982 (13)

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0*    (itns/time = 4/0.0)
```

```
************************************ nexp = 4
Least Square Fit:
  chi2/dof [dof] = 0.057 [10]    Q = 1    logGBF = 99.842

Parameters:
            a 0   0.39995 (77)       [  0.50 (50) ]
              1     0.399 (32)       [  0.50 (50) ]
              2      0.40 (10)       [  0.50 (50) ]
              3      0.40 (15)       [  0.50 (50) ]
            E 0   0.89998 (30)       [  1.00 (50) ]
              1     1.799 (23)       [  1.90 (54) ]
              2      2.70 (19)       [  2.80 (57) ]
              3      3.61 (28)       [  3.70 (61) ]

Fit:
     x[k]              y[k]         f(x[k],p)
-----------------------------------------------
        1       0.2656 (78)        0.2666 (22)
      1.2       0.2027 (32)       0.20297 (97)
      1.4       0.1573 (13)       0.15737 (42)
      1.6      0.12378 (54)       0.12381 (18)
      1.8      0.09853 (22)      0.098540 (79)
        2     0.079153 (93)      0.079155 (34)
      2.2     0.064051 (39)      0.064051 (15)
      2.4     0.052134 (16)     0.0521344 (64)
      2.6    0.0426348 (67)     0.0426347 (29)
      2.8    0.0349985 (28)     0.0349985 (13)

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0*    (itns/time = 4/0.0)


E1/E0 = 1.999(25)   E2/E0 = 3.00(21)
a1/a0 = 0.997(77)   a2/a0 = 1.01(26)
```

Here we use `fit.format(maxline=True)` to print out a table of `x` and `y` (actually `ymod`) values, together with the value of the fit function using the best-fit parameters. There are several things to notice:

- Were we really only interested in `a[0]` and `E[0]`, a single-exponential fit would have been adequate. This is because we are in effect doing a 20-exponential fit even in that case, by including all but the first term as corrections to `y`. The answers given by the first fit are correct (we know the exact values since we are using fake data).

  The ability to push uninteresting parameters into a `ymod` can be highly useful in practice since it is usually much cheaper to incorporate those fit parameters into `ymod` than it is to include them as fit parameters — fits with smaller numbers of parameters are usually a lot faster.

- The `chi**2` and best-fit parameter means and standard deviations are almost unchanged by shifting terms from `ymod` back into the fit function, as `nexp` increases. The final results for `a[0]` and `E[0]`, for example, are nearly identical in the `nexp=1` and `nexp=4` fits.

  In fact it is straightforward to prove that best-fit parameter means and standard deviations, as well as `chi**2`, should be exactly the same in such situations provided the fit function is linear in all fit parameters. Here the fit function is approximately linear, given our small standard deviations, and so results are only approximately independent of `nexp`.

- The uncertainty in `ymod` for a particular `x` decreases as `nexp` increases and as `x` increases. Also the `nexp` independence of the fit results depends upon capturing all of the correlations in the correction to `y`. This is why `gvar.GVar`s are useful since they make the implementation of those correlations trivial.

- Although we motivated this example by the need to deal with `ys` having no errors, it is straightforward to apply the same ideas to a situation where the `ys` have errors. Again one might want to do so since fitting uninteresting fit parameters is generally more costly than absorbing them into the `y` (which then has a modified mean and standard deviation).

1.11 SVD Cuts and Roundoff Error

All of the fits discussed above have (default) SVD cuts of 1e-15. This has little impact in most of the problems, but makes a big difference in the problem discussed in the previous section. Had we run that fit, for example, with an SVD cut of 1e-19, instead of 1e-15, we would have obtained the following output:

```
Least Square Fit:
  chi2/dof [dof] = 0.057 [10]    Q = 1    logGBF = 99.847

Parameters:
             a 0   0.39994 (77)       [  0.50 (50) ]
               1     0.398 (32)       [  0.50 (50) ]
               2      0.40 (10)       [  0.50 (50) ]
               3      0.40 (15)       [  0.50 (50) ]
             E 0   0.89997 (30)       [  1.00 (50) ]
               1     1.799 (23)       [  1.90 (54) ]
               2      2.70 (19)       [  2.80 (57) ]
               3      3.61 (28)       [  3.70 (61) ]

Fit:
     x[k]              y[k]        f(x[k],p)
-----------------------------------------------
        1       0.2656 (78)      0.2666 (57)
      1.2       0.2027 (32)      0.2030 (23)
      1.4       0.1573 (13)     0.15737 (92)
      1.6      0.12378 (54)     0.12381 (35)
      1.8      0.09853 (22)     0.09854 (12)
        2     0.079153 (93)    0.079155 (37)
      2.2     0.064051 (39)    0.064051 (18)
      2.4     0.052134 (16)    0.052134 (20)
      2.6    0.0426348 (67)    0.042635 (19)
      2.8    0.0349985 (28)    0.034998 (17)

Settings:
  svdcut/n = 1e-19/0    reltol/abstol = 0.0001/0    (itns/time = 5/0.0)


E1/E0 = 2.00(49)   E2/E0 = 3.0(3.8)
a1/a0 = 1.0(1.5)   a2/a0 = 1.0(3.3)
```

The standard deviations quoted for `E1/E0`, *etc.* are much too large compared with the standard deviations shown for the individual parameters, and much larger than what we obtained in the previous section. This is due to roundoff error. The standard deviations quoted for the parameters are computed differently from the standard deviations in `fit.p` (which was used to calculate `E1/E0`). The former come directly from the curvature of the `chi**2` function at its minimum; the latter are related back to the standard deviations of the input data and priors used in the fit. The two should agree, but they will not agree if the covariance matrix for the input `y` data is too ill-conditioned.

The inverse of the `y-prior` covariance matrix is used in the `chi**2` function that is minimized by *lsqfit.nonlinear_fit*. Given the finite precision of computer hardware, it is impossible to compute this inverse accurately if the matrix is singular or almost singular, and in such situations the reliability of the fit results is in question. The eigenvalues of the covariance matrix in this example (for `nexp=6`) indicate that this is the case:

they range from `7.2e-5` down to `4.2e-26`, covering 21 orders of magnitude. This is likely too large a range to be handled with the 16–18 digits of precision available in normal double precision computation. The smallest eigenvalues and their eigenvectors are likely to be quite inaccurate, as is any method for computing the inverse matrix.

One solution to this common problem in least-squares fitting is to introduce an SVD cut, here called `svdcut`:

```
fit = nonlinear_fit(data=(x, ymod), fcn=f, prior=prior, p0=p0, svdcut=1e-15)
```

This regulates the singularity of the covariance matrix by, in effect, replacing its smallest eigenvalues with a larger, minimum eigenvalue. The cost is less precision in the final results since we are decreasing the precision of the input `y` data. This is a conservative move, but numerical stability is worth the tradeoff. The listing shows that 2 eigenvalues are modified when `svdcut=1e-15` (see entry for `svdcut/n`); no eigenvalues are changed when `svdcut=1e-19`.

The SVD cut is actually applied to the correlation matrix, which is the covariance matrix rescaled by standard deviations so that all diagonal elements equal 1. Working with the correlation matrix rather than the covariance matrix helps mitigate problems caused by large scale differences between different variables. Eigenvalues of the correlation matrix that are smaller than a minimum eigenvalue, equal to `svdcut` times the largest eigenvalue, are replaced by the minimum eigenvalue, while leaving their eigenvectors unchanged. This defines a new, less singular correlation matrix from which a new, less singular covariance matrix is constructed. Larger values of `svdcut` affect larger numbers of eigenmodes and increase errors in the final results.

The error budget is different in the example above. There is no contribution from the original `y` data since it is exact. So all statistical uncertainty comes from the priors in `max_prior`, and from the SVD cut, which contributes since it modifies the effective variances of several eigenmodes of the correlation matrix. The SVD contribution to the error can be obtained from `fit.svdcorrection`, so the full error budget is constructed by the following code,

```
outputs = {'E1/E0':E[1] / E[0], 'E2/E0':E[2] / E[0],
           'a1/a0':a[1] / a[0], 'a2/a0':a[2] / a[0]}
inputs = {'E':max_prior['E'], 'a':max_prior['a'], 'svd':fit.svdcorrection}
print(fit.fmt_values(outputs))
print(fit.fmt_errorbudget(outputs, inputs))
```

which gives:

```
Values:
              E2/E0: 3.00(21)
              E1/E0: 1.999(25)
              a2/a0: 1.01(26)
              a1/a0: 0.997(77)

Partial % Errors:
               E2/E0     E1/E0     a2/a0     a1/a0
-------------------------------------------------
        a:      3.84      0.72     12.11      4.46
      svd:      0.29      0.10      0.13      0.55
        E:      5.86      0.99     22.26      6.30
-------------------------------------------------
    total:      7.01      1.23     25.35      7.74
```

Here the contribution from the SVD cut is almost negligible, which might not be the case in other applications.

The SVD cut is applied separately to each block diagonal sub-matrix of the correlation matrix. This means, among other things, that errors for uncorrelated data are unaffected by the SVD cut. Applying an SVD cut of 1e-4, for example, to the following singular covariance matrix,

```
[[  1.0   1.0   0.0  ]
 [  1.0   1.0   0.0  ]
 [  0.0   0.0   1e-20]],
```

gives a new, non-singular matrix

```
[[  1.0001    0.9999    0.0   ]
 [  0.9999    1.0001    0.0   ]
 [  0.0       0.0       1e-20]]
```

where only the upper right sub-matrix is different.

lsqfit.nonlinear_fit uses a default value for `svdcut` of 1e-15. This default can be overridden, as shown above, but for many problems it is a good choice. Roundoff errors become more accute, however, when there are strong correlations between different parts of the fit data or prior. Then much larger `svdcuts` may be needed.

The SVD cut is applied to both the data and the prior. It is possible to apply SVD cuts to either of these separately using `gvar.svd()` before the fit: for example,

```
ymod = gv.svd(ymod, svdcut=1e-10)
prior = gv.svd(prior, svdcut=1e-12)
fit = nonlinear_fit(data=(x, ymod), fcn=f, prior=prior, p0=p0, svdcut=None)
```

applies different SVD cuts to the prior and data.

Note that taking `svdcut=-1e-15`, with a minus sign, causes the problematic modes to be dropped. This is a more conventional implementation of SVD cuts, but here it results in much less precision than using `svdcut=1e-15` (giving, for example, 1.993(69) for `E1/E0`, which is almost three times less precise). Dropping modes is equivalent to setting the corresponding variances to infinity, which is (obviously) much more conservative and less realistic than setting them equal to the SVD-cutoff variance.

The method *lsqfit.nonlinear_fit.check_roundoff()* can be used to check for roundoff errors by adding the line `fit.check_roundoff()` after the fit. It generates a warning if roundoff looks to be a problem. This check is done automatically if `debug=True` is added to the argument list of *lsqfit.nonlinear_fit*.

1.12 Bootstrap Error Analysis

Our analysis above assumes that every probability distribution relevant to the fit is approximately Gaussian. For example, we characterize the input data for `y` by a mean and a covariance matrix obtained from averaging many random samples of `y`. For large sample sizes it is almost certainly true that the average values follow a Gaussian distribution, but in practical applications the sample size could be too small. The *statistical bootstrap* is an analysis tool for dealing with such situations.

The strategy is to: 1) make a large number of "bootstrap copies" of the original input data that differ from each other by random amounts characteristic of the underlying randomness in the original data; 2) repeat the entire fit analysis for each bootstrap copy of the data, extracting fit results from each; and 3) use the variation of the fit results from bootstrap copy to bootstrap copy to determine an approximate probability distribution (possibly non-Gaussian) for the each result.

Consider the code from the previous section, where we might reasonably want another check on the error estimates for our results. That code can be modified to include a bootstrap analysis by adding the following to the end of the `main()` subroutine:

```
Nbs = 40                                    # number of bootstrap copies
outputs = {'E1/E0':[], 'E2/E0':[], 'a1/a0':[], 'a2/a0':[]}   # results
for bsfit in fit.bootstrap_iter(n=Nbs):
    E = bsfit.pmean['E']                    # best-fit parameter values
    a = bsfit.pmean['a']                    #   (ignore errors)
    outputs['E1/E0'].append(E[1] / E[0])      # accumulate results
    outputs['E2/E0'].append(E[2] / E[0])
    outputs['a1/a0'].append(a[1] / a[0])
    outputs['a2/a0'].append(a[2] / a[0])
    outputs['E1'].append(E[1])
```

```
    outputs['a1'].append(a[1])
# extract "means" and "standard deviations" from the bootstrap output;
# print using .fmt() to create compact representation of GVars
outputs = gv.dataset.avg_data(outputs, bstrap=True)
print('Bootstrap results:')
print('E1/E0 =', outputs['E1/E0'].fmt(), '  E2/E1 =', outputs['E2/E0'].fmt())
print('a1/a0 =', outputs['a1/a0'].fmt(), '  a2/a0 =', outputs['a2/a0'].fmt())
print('E1 =', outputs['E1'].fmt(), '  a1 =', outputs['a1'].fmt())
```

The results are consistent with the results obtained directly from the fit (when using `svdcut=1e-15`):

```
Bootstrap results:
E1/E0 = 1.999(22)   E2/E1 = 2.96(22)
a1/a0 = 0.995(68)   a2/a0 = 1.00(38)
E1 = 1.799(20)   a1 = 0.398(28)
```

In particular, the bootstrap analysis confirms our previous error estimates (to within 10-30%, since `Nbs=40`). When `Nbs` is small, it is often safer to use the median instead of the mean as the estimator, which is what `gv.dataset.avg_data` does here since flag `bstrap` is set to `True`.

1.13 Testing Fits with Simulated Data

Ideally we would test a fitting protocol by doing fits of data similar to our actual fit but where we know the correct values for the fit parameters ahead of the fit. The *`lsqfit.nonlinear_fit`* iterator `simulated_fit_iter` creates any number of such simulations of the original fit. Returning again to the fits in the section on *Correlated Parameters; Gaussian Bayes Factor*, we can add three fit simulations to the end of the `main` program:

```
def main():
    gv.ranseed([2009, 2010, 2011, 2012]) # initialize random numbers (opt.)
    x, y = make_data()              # make fit data
    p0 = None                       # make larger fits go faster (opt.)
    for nexp in range(1, 5):
        print('************************************ nexp =', nexp)
        prior = make_prior(nexp)
        fit = lsqfit.nonlinear_fit(data=(x, y), fcn=f, prior=prior, p0=p0)
        print(fit)                  # print the fit results
        if nexp > 2:
            E = fit.p['E']              # best-fit parameters
            a = fit.p['a']
            print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
            print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
        if fit.chi2 / fit.dof < 1.:
            p0 = fit.pmean          # starting point for next fit (opt.)
        print()

    # 3 fit simulations based upon last fit
    for sfit in fit.simulated_fit_iter(3):
        print(sfit)
        sE = sfit.p['E']             # best-fit parameters (simulation)
        sa = sfit.p['a']
        E = sfit.pexact['E']         # correct results for parameters
        a = sfit.pexact['a']
        print('E1/E0 =', sE[1] / sE[0], '  E2/E0 =', sE[2] / sE[0])
        print('a1/a0 =', sa[1] / sa[0], '  a2/a0 =', sa[2] / sa[0])
        print('\nSimulated Fit Values - Exact Values:')
        print(
```

```
            'E1/E0:', (sE[1] / sE[0]) - (E[1] / E[0]),
            '  E2/E0:', (sE[2] / sE[0]) - (E[2] / E[0])
            )
        print(
            'a1/a0:', (sa[1] / sa[0]) - (a[1] / a[0]),
            '  a2/a0:', (sa[2] / sa[0]) - (a[2] / a[0])
            )

        # compute chi**2 comparing selected fit results to exact results
        sim_results = [sE[0], sE[1], sa[0], sa[1]]
        exact_results = [E[0], E[1], a[0], a[1]]
        chi2 = gv.chi2(sim_results, exact_results)
        print(
            '\nParameter chi2/dof [dof] = %.2f' % (chi2 / gv.chi2.dof),
            '[%d]' % gv.chi2.dof,
            '  Q = %.1f' % gv.chi2.Q
            )
```

The fit data for each of the three simulations is the same as the original fit data except that the means have been adjusted (randomly) so the correct values for the fit parameters are in each case equal to `pexact=fit.pmean`. Simulation fit results will typically differ from the correct values by an amount of order a standard deviation. With sufficiently accurate data, the results from a large number of simulations will be distributed in Gaussians centered on the correct values (`pexact`), with widths that equal the standard deviations given by the fit (`fit.psdev`). (With less accurate data, the distributions may become non-Gaussian, and the interpretation of fit results more complicated.)

In the present example, the output from the three simulations is:

```
************************************ simulation
Least Square Fit:
  chi2/dof [dof] = 0.53 [11]    Q = 0.88    logGBF = 185.72

Parameters:
            a 0    0.4076 (40)      [  0.50 (50) ]
              1    0.4166 (73)      [  0.50 (50) ]
              2     0.336 (50)      [  0.50 (50) ]
              3      0.48 (50)      [  0.50 (50) ]
            E 0   0.90130 (52)      [  1.00 (50) ]
              1    1.8074 (41)      [  1.90 (50) ]
              2     2.707 (11)      [  2.80 (50) ]
              3     3.607 (15)      [  3.70 (50) ]

Settings:
  svdcut/n = None/0    reltol/abstol = 0.0001/0*    (itns/time = 3/0.0)

E1/E0 = 2.0053(42)   E2/E0 = 3.004(12)
a1/a0 = 1.022(14)   a2/a0 = 0.82(12)

Simulated Fit Values - Exact Values:
E1/E0: 0.0029(42)   E2/E0: 0.002(12)
a1/a0: 0.013(14)   a2/a0: -0.10(12)

Parameter chi2/dof [dof] = 1.27 [4]   Q = 0.3

************************************ simulation
Least Square Fit:
  chi2/dof [dof] = 0.26 [11]    Q = 0.99    logGBF = 187.22

Parameters:
```

```
            a 0    0.4038 (40)      [  0.50 (50) ]
              1    0.4129 (72)      [  0.50 (50) ]
              2     0.332 (50)      [  0.50 (50) ]
              3      0.48 (50)      [  0.50 (50) ]
            E 0   0.90029 (52)      [  1.00 (50) ]
              1    1.8067 (41)      [  1.90 (50) ]
              2     2.707 (11)      [  2.80 (50) ]
              3     3.607 (15)      [  3.70 (50) ]

Settings:
  svdcut/n = None/0    reltol/abstol = 0.0001/0*    (itns/time = 3/0.0)

E1/E0 = 2.0068(42)   E2/E0 = 3.006(12)
a1/a0 = 1.023(14)   a2/a0 = 0.82(12)

Simulated Fit Values - Exact Values:
E1/E0: 0.0044(42)   E2/E0: 0.005(12)
a1/a0: 0.013(14)   a2/a0: -0.10(12)

Parameter chi2/dof [dof] = 0.56 [4]   Q = 0.7

************************************ simulation
Least Square Fit:
  chi2/dof [dof] = 0.6 [11]    Q = 0.83    logGBF = 185.3

Parameters:
            a 0    0.4053 (40)      [  0.50 (50) ]
              1    0.4051 (70)      [  0.50 (50) ]
              2     0.406 (48)      [  0.50 (50) ]
              3      0.50 (50)      [  0.50 (50) ]
            E 0   0.89959 (52)      [  1.00 (50) ]
              1    1.7995 (41)      [  1.90 (50) ]
              2     2.700 (11)      [  2.80 (50) ]
              3     3.600 (15)      [  3.70 (50) ]

Settings:
  svdcut/n = None/0    reltol/abstol = 0.0001/0*    (itns/time = 3/0.0)

E1/E0 = 2.0004(42)   E2/E0 = 3.001(12)
a1/a0 = 1.000(14)   a2/a0 = 1.00(12)

Simulated Fit Values - Exact Values:
E1/E0: -0.0020(42)   E2/E0: -0.001(12)
a1/a0: -0.010(14)   a2/a0: 0.08(12)

Parameter chi2/dof [dof] = 0.97 [4]   Q = 0.4
```

The simulations show that the fit values usually agree with the correct values to within a standard deviation or so (the correct results here are the mean values from the last fit discussed in *Correlated Parameters; Gaussian Bayes Factor*). Furthermore the error estimates for each parameter from the original fit are reproduced by the simulations. We also compute the `chi**2` for the difference between the leading fit parameters and the exact values. This checks parameter values, standard deviations, and correlations. The results are reasonable for four degrees of freedom. Here the first simulation shows results that are off by a third of a standard deviation on average, but this is not so unusual — the `Q=0.1` indicates that it happens 10% of the time.

More thorough testing is possible: for example, one could run many simulations (100?) to verify that the distribution of (simulation) fit results is Gaussian, centered around `pexact`. This is overkill in most situations, however. The three simulations above are enough to reassure us that the original fit estimates, including errors, are reliable.

1.14 Positive Parameters; Non-Gaussian Priors

The priors for *lsqfit.nonlinear_fit* are all Gaussian. There are situations, however, where other distributions would be desirable. One such case is where a parameter is known to be positive, but is close to zero in value ("close" being defined relative to the *a priori* uncertainty). For such cases we would like to use non-Gaussian priors that force positivity — for example, priors that impose log-normal or exponential distributions on the parameter. Ideally the decision to use such a distribution is made on a parameter- by-parameter basis, when creating the priors, and has no impact on the definition of the fit function itself.

lsqfit.nonlinear_fit supports log-normal distributions when `extend=True` is set in its argument list. This argument only affects fits that use dictionaries for their parameters. The prior for a parameter `'c'` is switched from a Gaussian distribution to a log-normal distribtuion by replacing parameter `'c'` in the fit prior with a prior for its logarithm, using the key `'log(c)'`. This causes *lsqfit.nonlinear_fit* to use the logarithm as the fit parameter (with its Gaussian prior). Parameter dictionaries produced by *lsqfit.nonlinear_fit* will have entries for both `'c'` and `'log(c)'`, so only the prior need be changed to switch distributions. In particular the fit function can be expressed directly in terms of `'c'` so that it is independent of the distribution chosen for the `'c'` prior.

To illustrate consider a simple problem where an experimental quantity `y` is known to be positive, but experimental errors mean that measured values can often be negative:

```
import gvar as gv
import lsqfit

y = gv.gvar([
   '-0.17(20)', '-0.03(20)', '-0.39(20)', '0.10(20)', '-0.03(20)',
   '0.06(20)', '-0.23(20)', '-0.23(20)', '-0.15(20)', '-0.01(20)',
   '-0.12(20)', '0.05(20)', '-0.09(20)', '-0.36(20)', '0.09(20)',
   '-0.07(20)', '-0.31(20)', '0.12(20)', '0.11(20)', '0.13(20)'
   ])
```

We want to know the average value `a` of the `y`s and so could use the following fitting code:

```
prior = gv.BufferDict()
prior['a'] = gv.gvar(0.02, 0.02)                # a = avg value of y's

def fcn(p, N=len(y)):
   return N * [p['a']]

fit = lsqfit.nonlinear_fit(prior=prior, data=y, fcn=fcn)
print(fit)
print('a =', fit.p['a'].fmt())
```

where we are assuming *a priori* information that suggests the average is around 0.02. The output from this code is:

```
Least Square Fit:
  chi2/dof [dof] = 0.84 [20]    Q = 0.67    logGBF = 5.3431

Parameters:
              a   0.004 (18)      [ 0.020 (20) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0    (itns/time = 2/0.0)

a = 0.004(18)
```

This is not such a useful result since much of the one-sigma range for `a` is negative, and yet we know that `a` must be postive.

A better analysis is to use a log-normal distribution for `a`:

```
prior = gv.BufferDict()
prior['log(a)'] = gv.log(gv.gvar(0.02, 0.02)) # loga not a

def fcn(p, N=len(y)):
   return N * [p['a']]

fit = lsqfit.nonlinear_fit(prior=prior, data=y, fcn=fcn, extend=True)
print(fit)
print('a =', fit.p['a'].fmt())                  # exp(log(a))
```

The fit parameter is now `log(a)` rather than `a` itself, but the code is unchanged except for the definition of the prior and the addition of `extend=True` to the *lsqfit.nonlinear_fit* arguments. In particular the fit function is identical to what we used in the first case since parameter dictionary `p` has entries for both `'a'` and `'log(a)'`.

The result from this fit is

```
Least Square Fit:
  chi2/dof [dof] = 0.85 [20]    Q = 0.65    logGBF = 5.252

Parameters:
         log(a)   -4.44 (97)     [ -3.9 (1.0) ]
-----------------------------------------------
             a    0.012 (11)     [ 0.020 (20) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 12/0.0)

a = 0.012(11)
```

which is more compelling. Parameters listed above the dashed line in the parameter table are the actual parameters used in the fit; those listed below the dashed line are derived from those above the line. The "correct" value for `a` here is 0.015 (given the method used to generate the `ys`).

Setting `extend=True` in *lsqfit.nonlinear_fit* also allows parameters to be replaced by their square roots as fit parameters, or by the inverse error function. The latter option is useful here because it allows us to define a prior distribution for parameter `a` that is uniform between 0 and 0.04:

```
prior = gv.BufferDict()
prior['erfinv(50*a - 1)'] = gv.gvar('0(1)') / gv.sqrt(2)

def fcn(p, N=len(y)):
   a = (1 + p['50*a-1']) / 50.
   return N * [a]

fit = lsqfit.nonlinear_fit(prior=prior, data=y, fcn=fcn, extend=True)
print(fit)
print('a =', ((1+fit.p['a']) / 50).fmt())
```

In general, setting a prior `prior['erfinv(w)']` equal to (0±1)/sqrt(2) means that the prior probability for variable `w` is constant between -1 and 1, and zero elsewhere. Here `w=50*a-1`, so that the prior distribution for `a` is uniform between 0 and 0.04, and zero elsewhere. This again guarantees a positive parameter.

The result from this last fit is:

```
Least Square Fit:
  chi2/dof [dof] = 0.85 [20]    Q = 0.65    logGBF = 5.2385

Parameters:
  erfinv(50a-1)   -0.42 (68)     [  0.00 (71) ]
```

```
-----------------------------------------------
          50a-1   -0.44 (64)     [  0.00 (80) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 9/0.0)

a = 0.011(13)
```

This fit implies that `a=0.011(13)` which is almost identical to the result obtained from the log-normal distribution.

Other distributions can be defined using `lsqfit.add_parameter_distribution()`. For example,

```
import lsqfit
import gvar as gv

def invf(x):
  return 0.02 + 0.02 * gv.tanh(x)

def f(x):                                   # not used
  return gv.arctanh((x - 0.02) / 0.02)

lsqfit.add_parameter_distribution('f', invf)
prior = gv.BufferDict()
prior['f(a)'] = gv.gvar('0.00(75)')

... as before ...
```

does a fit with Gaussian parameter `f(a)`, which forces `a` to lie between 0 and 0.04. This fit gives `a=0.012(12)`, which again agrees well with log-normal fit. The prior 0 ± 0.75 for `f(a)` is chosen to make the prior probability distribution for parameter `a` almost flat across most (80%) of the interval 0.2 ± 0.2.

1.15 Debugging and Troubleshooting

It is a very good idea to set parameter `debug=True` in *`lsqfit.nonlinear_fit`*, at least in the early stages of a project. This causes the code to look for common mistakes and report on them with more intelligible error messages. The code also then checks for significant roundoff errors in the matrix inversion of the covariance matrice.

A common mistake is a mismatch between the format of the data and the format of what comes back from the fit function. Another mistake is when a fit function `fcn(p)` returns results containing `gvar.GVars` when the parameters `p` are all just numbers (or arrays of numbers). The only way a `gvar.GVar` should get into a fit function is through the parameters; if a fit function requires an extra `gvar.GVar`, that `gvar.GVar` should be turned into a parameter by adding it to the prior.

Error messages that come from inside the *gsl* routines used by *`lsqfit.nonlinear_fit`* are sometimes less than useful. They are usually due to errors in one of the inputs to the fit (that is, the fit data, the prior, or the fit function). Again setting `debug=True` may catch the errors before they land in *gsl*.

Occasionally *`lsqfit.nonlinear_fit`* appears to go crazy, with gigantic `chi**2`s (*e.g.*, `1e78`). This could be because there is a genuine zero-eigenvalue mode in the covariance matrix of the data or prior. Such a zero mode makes it impossible to invert the covariance matrix when evaluating `chi**2`. One fix is to include SVD cuts in the fit by setting, for example, `svdcut=1e-8` in the call to *`lsqfit.nonlinear_fit`*. These cuts will exclude exact or nearly exact zero modes, while leaving important modes mostly unaffected.

Even if the SVD cuts work in such a case, the question remains as to why one of the covariance matrices has a zero mode. A common cause is if the same `gvar.GVar` was used for more than one prior. For example, one might think that

```
>>> import gvar as gv
>>> z = gv.gvar(1, 1)
>>> prior = gv.BufferDict(a=z, b=z)
```

creates a prior 1 ± 1 for each of parameter `a` and parameter `b`. Indeed each parameter separately is of order 1 ± 1, but in a fit the two parameters would be forced equal to each other because their priors are both set equal to the same `gvar.GVar`, `z`:

```
>>> print(prior['a'], prior['b'])
1.0(1.0) 1.0(1.0)
>>> print(prior['a']-prior['b'])
0(0)
```

That is, while parameters `a` and `b` fluctuate over a range of 1 ± 1, they fluctuate together, in exact lock-step. The covariance matrix for `a` and `b` must therefore be singular, with a zero mode corresponding to the combination `a-b`; it is all 1s in this case:

```
>>> import numpy as np
>>> cov = gv.evalcov(prior.flat)     # prior's covariance matrix
>>> print(np.linalg.det(cov))        # determinant is zero
0.0
```

This zero mode upsets `nonlinear_fit()`. If `a` and `b` are meant to fluctuate together then an SVD cut as above will give correct results (with `a` and `b` being forced equal to several decimal places, depending upon the cut). Of course, simply replacing `b` by `a` in the fit function would be even better. If, on the other hand, `a` and `b` were not meant to fluctuate together, the prior should be redefined:

```
>>> prior = gv.BufferDict(a=gv.gvar(1, 1), b=gv.gvar(1, 1))
```

where now each parameter has its own `gvar.GVar`. A slightly more succinct way of writing this line is:

```
>>> prior = gv.gvar(gv.BufferDict(a='1(1)', b='1(1)'))
```

CHAPTER

TWO

CASE STUDY: SIMPLE EXTRAPOLATION

In this case study, we examine a simple extrapolation problem. We show first how *not* to solve this problem. A better solution follows, together with a discussion of priors and Bayes factors. Finally a very simple, alternative solution, using marginalization, is described.

2.1 The Problem

Consider a problem where we have five pieces of uncorrelated data for a function `y(x)`:

```
x[i]        y(x[i])
----------------------
0.1         0.5351 (54)
0.3         0.6762 (67)
0.5         0.9227 (91)
0.7         1.3803(131)
0.95        4.0145(399)
```

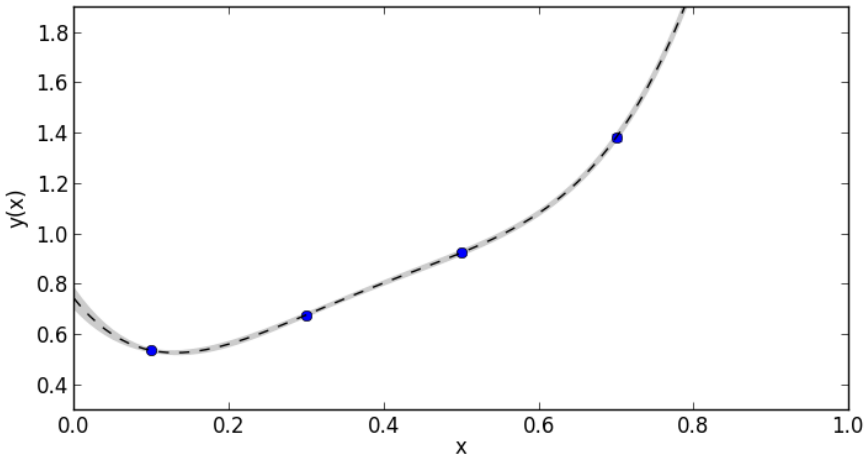
We know that `y(x)` has a Taylor expansion in `x`:

```
y(x) = sum_n=0..inf p[n] x**n
```

The challenge is to extract a reliable estimate for `y(0)=p[0]` from the data — that is, the challenge is to fit the data and use the fit to extrapolate the data to `x=0`.

2.2 A Bad Solution

One approach that is certainly wrong is to fit the data with a power series expansion for `y(x)` that is truncated after five terms (`n<=4`) — there are only five pieces of data and such a fit would have five parameters. This approach gives the following fit, where the gray band shows the 1-sigma uncertainty in the fit function evaluated with the best-fit parameters:



This fit was generated using the following code:

```
import numpy as np
import gvar as gv
import lsqfit

# fit data
y = gv.gvar([
   '0.5351(54)', '0.6762(67)', '0.9227(91)', '1.3803(131)', '4.0145(399)'
   ])
x = np.array([0.1, 0.3, 0.5, 0.7, 0.95])

# fit function
def f(x, p):
   return sum(pn * x ** n for n, pn in enumerate(p))

p0 = np.ones(5.)              # starting value for chi**2 minimization
fit = lsqfit.nonlinear_fit(data=(x, y), p0=p0, fcn=f)
print(fit.format(maxline=True))
```

Note that here the function `gv.gvar` converts the strings `'0.5351(54)'`, *etc.* into `gvar.GVars`. Running the code gives the following output:

```
Least Square Fit (no prior):
  chi2/dof [dof] = 1.2e-26 [0]    Q = 0    logGBF = None

Parameters:
              0    0.742 (39)      [   1 +- inf ]
              1    -3.86 (59)      [   1 +- inf ]
              2    21.5 (2.4)      [   1 +- inf ]
              3   -39.1 (3.7)      [   1 +- inf ]
              4    25.8 (1.9)      [   1 +- inf ]

Fit:
     x[k]           y[k]       f(x[k],p)
-----------------------------------------
      0.1    0.5351 (54)     0.5351 (54)
      0.3    0.6762 (67)     0.6762 (67)
      0.5    0.9227 (91)     0.9227 (91)
      0.7     1.380 (13)      1.380 (13)
     0.95     4.014 (40)      4.014 (40)
```

```
Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0    (itns/time = 2/0.0)
```

This is a "perfect" fit in that the fit function agrees exactly with the data; the `chi**2` for the fit is zero. The 5-parameter fit gives a fairly precise answer for `p[0]` (`0.74(4)`), but the curve looks oddly stiff. Also some of the best-fit values for the coefficients are quite large (*e.g.*, `p[3]= -39(4)`), perhaps unreasonably large.

2.3 A Better Solution — Priors

The problem with a 5-parameter fit is that there is no reason to neglect terms in the expansion of `y(x)` with `n>4`. Whether or not extra terms are important depends entirely on how large we expect the coefficients `p[n]` for `n>4` to be. The extrapolation problem is impossible without some idea of the size of these parameters; we need extra information.

In this case that extra information is obviously connected to questions of convergence of the Taylor expansion we are using to model `y(x)`. Let's assume we know, from previous work, that the `p[n]` are of order one. Then we would need to keep at least 91 terms in the Taylor expansion if we wanted the terms we dropped to be small compared with the 1% data errors at `x=0.95`. So a possible fitting function would be:

```
y(x; N) = sum_n=0..N p[n] x**n
```

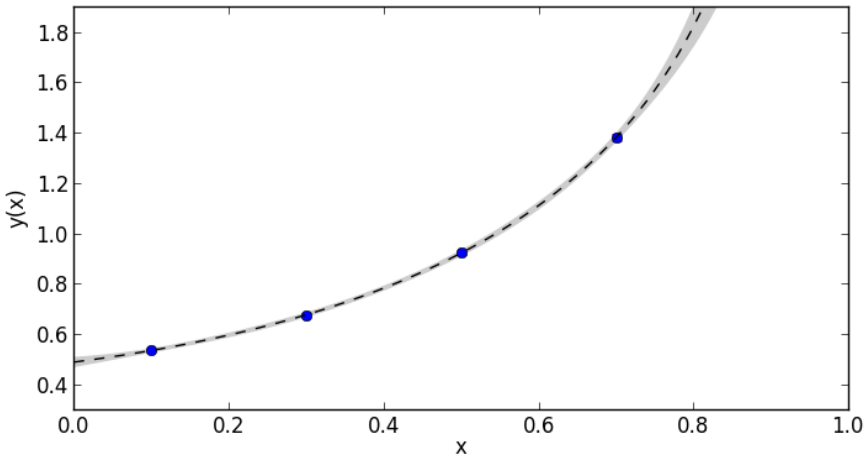
with `N=90`.

Fitting a 91-parameter formula to five pieces of data is also impossible. Here, however, we have extra (*prior*) information: each coefficient is order one, which we make specific by saying that they equal 0 ± 1. We include these *a priori* estimates for the parameters as extra data that must be fit, together with our original data. So we are actually fitting 91+5 pieces of data with 91 parameters.

The prior information is introduced into the fit as a *prior*:

```
import numpy as np
import gvar as gv
import lsqfit

# fit data
y = gv.gvar([
   '0.5351(54)', '0.6762(67)', '0.9227(91)', '1.3803(131)', '4.0145(399)'
   ])
x = np.array([0.1, 0.3, 0.5, 0.7, 0.95])

# fit function
def f(x, p):
   return sum(pn * x ** n for n, pn in enumerate(p))

# 91-parameter prior for the fit
prior = gv.gvar(91 * ['0(1)'])

fit = lsqfit.nonlinear_fit(data=(x, y), prior=prior, fcn=f)
print(fit.format(maxline=True))
```

Note that a starting value `p0` is not needed when a prior is specified. This code also gives an excellent fit, with a `chi**2` per degree of freedom of `0.35` (note that the data point at `x=0.95` is off the chart, but agrees with the fit to within its 1% errors):



The fit code output is:

```
Least Square Fit:
  chi2/dof [dof] = 0.35 [5]    Q = 0.88    logGBF = -0.45508

Parameters:
              0      0.489 (17)      [  0.0 (1.0) ]
              1       0.40 (20)      [  0.0 (1.0) ]
              2       0.60 (64)      [  0.0 (1.0) ]
              3       0.44 (80)      [  0.0 (1.0) ]
              4       0.28 (87)      [  0.0 (1.0) ]
              5       0.19 (87)      [  0.0 (1.0) ]
              6       0.16 (90)      [  0.0 (1.0) ]
              7       0.16 (93)      [  0.0 (1.0) ]
              8       0.17 (95)      [  0.0 (1.0) ]
              9       0.18 (96)      [  0.0 (1.0) ]
             10       0.19 (97)      [  0.0 (1.0) ]
             11       0.19 (97)      [  0.0 (1.0) ]
             12       0.19 (97)      [  0.0 (1.0) ]
             13       0.19 (97)      [  0.0 (1.0) ]
             14       0.18 (97)      [  0.0 (1.0) ]
             15       0.18 (97)      [  0.0 (1.0) ]
                .
                .
                .
             88   0.004 (1.000)      [  0.0 (1.0) ]
             89   0.004 (1.000)      [  0.0 (1.0) ]
             90   0.004 (1.000)      [  0.0 (1.0) ]

Fit:
     x[k]           y[k]      f(x[k],p)
----------------------------------------
      0.1    0.5351 (54)    0.5349 (54)
      0.3    0.6762 (67)    0.6768 (65)
      0.5    0.9227 (91)    0.9219 (87)
      0.7     1.380 (13)     1.381 (13)
     0.95     4.014 (40)     4.014 (40)

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0    (itns/time = 2/0.0)
```

This is a much more plausible fit than than the 5-parameter fit, and gives an extrapolated value of `p[0]=0.489(17)`. The original data points were created using a Taylor expansion with random coefficients, but with `p[0]` set equal to `0.5`. So this fit to the five data points (plus 91 *a priori* values for the `p[n]` with `n<91`) gives the correct result. Increasing the number of terms further would have no effect since the last terms added are having no impact, and so end up equal to the prior value — the fit data are not sufficiently precise to add new information about these parameters.

2.4 Bayes Factors

We can test our priors for this fit by re-doing the fit with broader and narrower priors. Setting `prior = gv.gvar(91 * ['0(3)'])` gives an excellent fit,

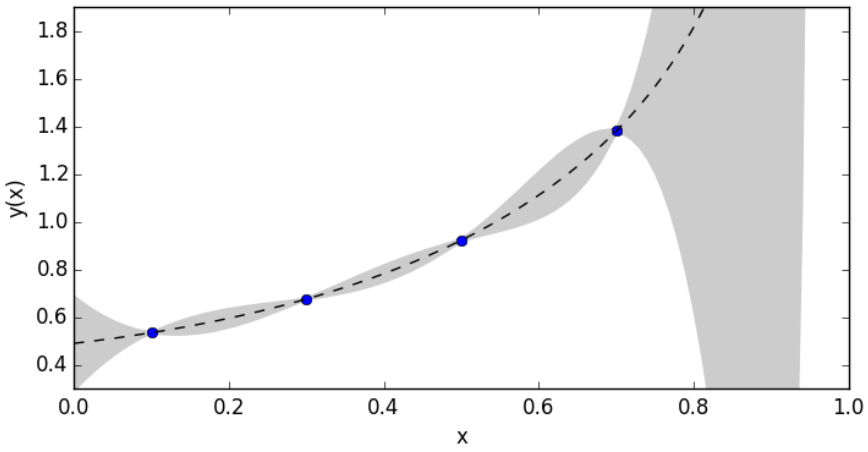
```
Least Square Fit:
  chi2/dof [dof] = 0.039 [5]    Q = 1    logGBF = -5.0993

Parameters:
              0      0.490 (33)       [  0.0 (3.0) ]
              1       0.38 (48)       [  0.0 (3.0) ]
              2       0.6 (1.8)       [  0.0 (3.0) ]
              3       0.5 (2.4)       [  0.0 (3.0) ]
              ...
```

but with a very small `chi2/dof` and somewhat larger errors on the best-fit estimates for the parameters. The logarithm of the (Gaussian) Bayes Factor, `logGBF`, can be used to compare fits with different priors. It is the logarithm of the probability that our data would come from parameters generated at random using the prior. The exponential of `logGBF` is more than 100 times larger with the original priors of `0(1)` than with priors of `0(3)`. This says that our data is more than 100 times more likely to come from a world with parameters of order one than from one with parameters of order three. Put another way it says that the size of the fluctuations in the data are more consistent with coefficients of order one than with coefficients of order three — in the latter case, there would have been larger fluctuations in the data than are actually seen. The `logGBF` values argue for the original prior.

Narrower priors, `prior = gv.gvar(91 * ['0.0(3)'])`, give a poor fit, and also a less optimal `logGBF`:

```
Least Square Fit:
  chi2/dof [dof] = 3.7 [5]    Q = 0.0024    logGBF = -3.3058

Parameters:
              0    0.484 (11)      [  0.00 (30) ]  *
              1    0.454 (98)      [  0.00 (30) ]  *
              2     0.50 (23)      [  0.00 (30) ]  *
              3     0.40 (25)      [  0.00 (30) ]  *
              ...
```

Setting `prior = gv.gvar(91 * ['0(20)'])` gives very wide priors and a rather strange looking fit:



Here fit errors are comparable to the data errors at the data points, as you would expect, but balloon up in between. This is an example of *over-fitting*: the data and priors are not sufficiently accurate to fit the number of parameters used. Specifically the priors are too broad. Again the Bayes Factor signals the problem: `logGBF = -14.479` here, which means that our data are roughly a million times (`=exp(14)`) more likely to to come from a world with coefficients of order one than from one with coefficients of order twenty. That is, the broad priors suggest much larger variations between the leading parameters than is indicated by the data — again, the data are unnaturally regular in a world described by the very broad prior.

Absent useful *a priori* information about the parameters, we can sometimes use the data to suggest a plausible width for a set of priors. We do this by setting the width equal to the value that maximizes `logGBF`. This approach suggests priors of `0.0(6)` for the fit above, which gives results very similar to the fit with priors of `0(1)`. See *Tuning Priors and the Empirical Bayes Criterion* for more details.

The priors are responsible for about half of the final error in our best estimate of `p[0]` (with priors of `0(1)`); the rest comes from the uncertainty in the data. This can be established by creating an error budget using the code

```
inputs = dict(prior=prior, y=y)
outputs = dict(p0=fit.p[0])
print(gv.fmt_errorbudget(inputs=inputs, outputs=outputs))
```

which prints the following table:

```
Partial % Errors:
                            p0
-------------------------------
                 y:       2.67
             prior:       2.23
-------------------------------
             total:       3.48
```

The table shows that the final 3.5% error comes from a 2.7% error due to uncertainties in `y` and a 2.2% error from uncertainties in the prior (added in quadrature).

2.5 Another Solution — Marginalization

There is a second, equivalent way of fitting this data that illustrates the idea of *marginalization*. We really only care about parameter `p[0]` in our fit. This suggests that we remove `n>0` terms from the data *before* we do the fit:

```
ymod[i] = y[i] - sum_n=1...inf prior[n] * x[i] ** n
```

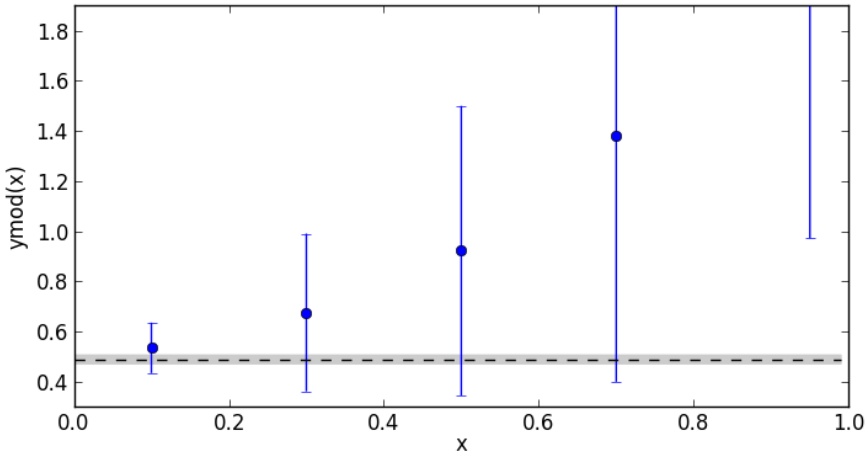
Before the fit, our best estimate for the parameters is from the priors. We use these to create an estimate for the correction to each data point coming from `n>0` terms in `y(x)`. This new data, `ymod[i]`, should be fit with a new fitting function, `ymod(x) = p[0]` — that is, it should be fit to a constant, independent of `x[i]`. The last three lines of the code above are easily modified to implement this idea:

```
import numpy as np
import gvar as gv
import lsqfit

# fit data
y = gv.gvar([
   '0.5351(54)', '0.6762(67)', '0.9227(91)', '1.3803(131)', '4.0145(399)'
   ])
x = np.array([0.1, 0.3, 0.5, 0.7, 0.95])

# fit function
def f(x, p):
   return sum(pn * x ** n for n, pn in enumerate(p))

# prior for the fit
prior = gv.gvar(91 * ['0(1)'])

# marginalize all but one parameter (p[0])
priormod = prior[:1]                        # restrict fit to p[0]
ymod = y - (f(x, prior) - f(x, priormod))   # correct y

fit = lsqfit.nonlinear_fit(data=(x, ymod), prior=priormod, fcn=f)
print(fit.format(maxline=True))
```

Running this code give:

```
Least Square Fit:
  chi2/dof [dof] = 0.35 [5]    Q = 0.88    logGBF = -0.45508

Parameters:
              0   0.489 (17)      [  0.0 (1.0) ]

Fit:
     x[k]          y[k]      f(x[k],p)
-------------------------------------
      0.1    0.54 (10)     0.489 (17)
      0.3    0.68 (31)     0.489 (17)
      0.5    0.92 (58)     0.489 (17)
      0.7    1.38 (98)     0.489 (17)
     0.95    4.0 (3.0)     0.489 (17)  *

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0    (itns/time = 2/0.0)
```

Remarkably this one-parameter fit gives results for `p[0]` that are identical (to machine precision) to our 91-parameter fit above. The 90 parameters for `n>0` are said to have been *marginalized* in this fit. Marginalizing a parameter in this way has no effect if the fit function is linear in that parameter. Marginalization has almost no effect for nonlinear fits as well, provided the fit data have small errors (in which case the parameters are effectively linear). The fit here is:



The constant is consistent with all of the data in `ymod[i]`, even at `x[i]=0.95`, because `ymod[i]` has much larger errors for larger `x[i]` because of the correction terms.

Fitting to a constant is equivalent to doing a weighted average of the data plus the prior, so our fit can be replaced by an average:

```
lsqfit.wavg(list(ymod) + list(priormod))
```

This again gives `0.489(17)` for our final result. Note that the central value for this average is below the central values for every data point in `ymod[i]`. This is a consequence of large positive correlations introduced into `ymod` when we remove the `n>0` terms. These correlations are captured automatically in our code, and are essential — removing the correlations between different `ymod`s results in a final answer, `0.564(97)`, which has a much larger error.

CHAPTER

THREE

CASE STUDY: PENDULUM

This case study shows how to fit a differential equation, using `gvar.ode`, and how to deal with uncertainty in the independent variable of a fit (that is, the `x` in a `y` versus `x` fit).

3.1 The Problem

A pendulum is released at time 0 from angle 1.571(50) (radians). It's angular position is measured at intervals of approximately a tenth of second:

```
t[i]       theta(t[i])
----------------------
0.0          1.571(50)
0.10(1)      1.477(79)
0.20(1)      0.791(79)
0.30(1)     -0.046(79)
0.40(1)     -0.852(79)
0.50(1)     -1.523(79)
0.60(1)     -1.647(79)
0.70(1)     -1.216(79)
0.80(1)     -0.810(79)
0.90(1)      0.185(79)
1.00(1)      0.832(79)
```

Function `theta(t)` satisfies a differential equation:

```
d/dt d/dt theta(t)  =  -(g/l) sin(theta(t))
```

where `g` is the acceleration due to gravity and `l` is the pendulum's length. The challenge is to use the data to improve our very approximate *a priori* estimate 40±20 for `g/l`.

3.2 Pendulum Dynamics

We start by designing a data type that solves the differential equation for `theta(t)`:

```
import numpy as np
import gvar as gv

class Pendulum(object):
    """ Integrator for pendulum motion.

    Input parameters are:
        g/l .... where g is acceleration due to gravity and l the length
```

```
        tol .... precision of numerical integration of ODE
    """
    def __init__(self, g_l, tol=1e-4):
        self.g_l = g_l
        self.odeint = gv.ode.Integrator(deriv=self.deriv, tol=tol)

    def __call__(self, theta0, t_array):
        """ Calculate pendulum angle theta for every t in t_array.

        Assumes that the pendulum is released at time t=0
        from angle theta0 with no initial velocity. Returns
        an array containing theta(t) for every t in t_array.
        """
        # initial values
        t0 = 0
        y0 = [theta0, 0.0]              # theta and dtheta/dt

        # solution  (keep only theta; discard dtheta/dt)
        y = self.odeint.solution(t0, y0)
        return [y(t)[0] for t in t_array]

    def deriv(self, t, y, data=None):
        " Calculate [dtheta/dt, d2theta/dt2] from [theta, dtheta/dt]."
        theta, dtheta_dt = y
        return np.array([dtheta_dt, - self.g_l * gv.sin(theta)])
```

A `Pendulum` object is initialized with a value for `g/l` and a tolerance for the differential-equation integrator, `gvar.ode.Integrator`. Evaluating the object for a given value of `theta(0)` and `t` then calculates `theta(t)`; `t` is an array. We use `gvar.ode` here, rather than some other integrator, because it works with `gvar.GVars`, allowing errors to propagate through the integration.

3.3 Two Types of Input Data

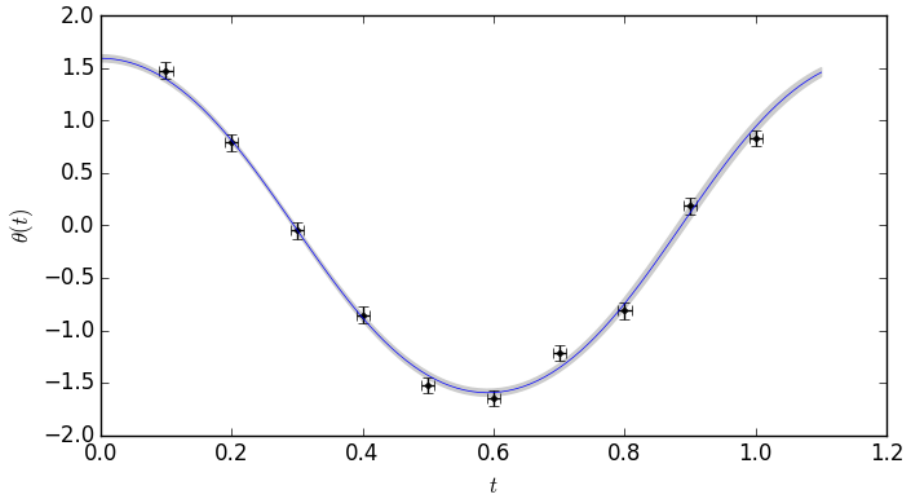
There are two ways to include data in a fit: either as regular data, or as fit parameters with priors. In general dependent variables are treated as regular data, and independent variables with errors are treated as fit parameters, with priors. Here the dependent variable is `theta(t)` and the independent variable is `t`. The independent variable has uncertainties, so we treat the individual values as fit parameters whose priors equal the initial values `t[i]`. The value of `theta(t=0)` is also independent data, and so becomes a fit parameter since it is uncertain. Our fit code therefore is:

```
from __future__ import print_function   # makes this work for python2 and 3

import collections
import numpy as np
import gvar as gv
import lsqfit

def main():
    # pendulum data exhibits experimental error in theta and t
    t = gv.gvar([
        '0.10(1)', '0.20(1)', '0.30(1)', '0.40(1)',  '0.50(1)',
        '0.60(1)',  '0.70(1)',  '0.80(1)',  '0.90(1)', '1.00(1)'
        ])
    theta = gv.gvar([
        '1.477(79)', '0.791(79)', '-0.046(79)', '-0.852(79)',
        '-1.523(79)', '-1.647(79)', '-1.216(79)', '-0.810(79)',
        '0.185(79)', '0.832(79)'
```

```
        ])

    # priors for all fit parameters: g/l, theta(0), and t[i]
    prior = collections.OrderedDict()
    prior['g/l'] = gv.gvar('40(20)')
    prior['theta(0)'] = gv.gvar('1.571(50)')
    prior['t'] = t

    # fit function: use class Pendulum object to integrate pendulum motion
    def fitfcn(p, t=None):
        if t is None:
            t = p['t']
        pendulum = Pendulum(p['g/l'])
        return pendulum(p['theta(0)'], t)

    # do the fit and print results
    fit = lsqfit.nonlinear_fit(data=theta, prior=prior, fcn=fitfcn)
    print(fit.format(maxline=True))
```

The prior is a dictionary containing *a priori* estimates for every fit parameter. The fit parameters are varied to give the best fit to both the data and the priors. The fit function uses a `Pendulum` object to integrate the differential equation for `theta(t)`, generating values for each value of `t[i]` given a value for `theta(0)`. The function returns an array that has the same shape as array `theta`.

The fit is excellent with a `chi**2` per degree of freedom of 0.7:



The gray band in the figure shows the best fit to the data, with the error bars on the fit. The output from this fit is:

```
Least Square Fit:
  chi2/dof [dof] = 0.7 [10]    Q = 0.73    logGBF = 6.359

Parameters:
            g/l    39.82 (87)      [    40 (20) ]
       theta(0)    1.595 (32)      [ 1.571 (50) ]
            t 0   0.0960 (91)      [ 0.100 (10) ]
              1   0.2014 (74)      [ 0.200 (10) ]
              2   0.3003 (67)      [ 0.300 (10) ]
              3   0.3982 (76)      [ 0.400 (10) ]
              4   0.5043 (93)      [ 0.500 (10) ]
```

```
               5    0.600 (10)      [ 0.600 (10) ]
               6   0.7079 (89)      [ 0.700 (10) ]
               7   0.7958 (79)      [ 0.800 (10) ]
               8   0.9039 (78)      [ 0.900 (10) ]
               9   0.9929 (83)      [ 1.000 (10) ]

Fit:
      key          y[key]       f(p)[key]
-------------------------------------------
        0     1.477 (79)     1.412 (42)
        1     0.791 (79)     0.802 (56)
        2    -0.046 (79)    -0.044 (60)
        3    -0.852 (79)    -0.867 (56)
        4    -1.523 (79)    -1.446 (42)
        5    -1.647 (79)    -1.594 (32)
        6    -1.216 (79)    -1.323 (49)  *
        7    -0.810 (79)    -0.776 (61)
        8     0.185 (79)     0.158 (66)
        9     0.832 (79)     0.894 (63)

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0    (itns/time = 4/0.0)
```

The final result for `g/l` is 39.8(9), which is accurate to about 2%. Note that the fit generates (slightly) improved estimates for several of the `t` values and for `theta(0)`.

CHAPTER

FOUR

CASE STUDY: OUTLIERS AND BAYESIAN INTEGRALS

In this case study, we analyze a fit with outliers in the data that distort the least-squares solution. We show one approach to dealing with the outliers that requires using Bayesian integrals in place of least-squares fitting, to fit the data while also modeling the outliers.

This case study is adapted from an example by Jake Vanderplas on his Python blog.

4.1 The Problem

We want to extrapolate a set of data values `y` to `x=0` fitting a linear fit function (`fitfcn(x,p)`) to the data:

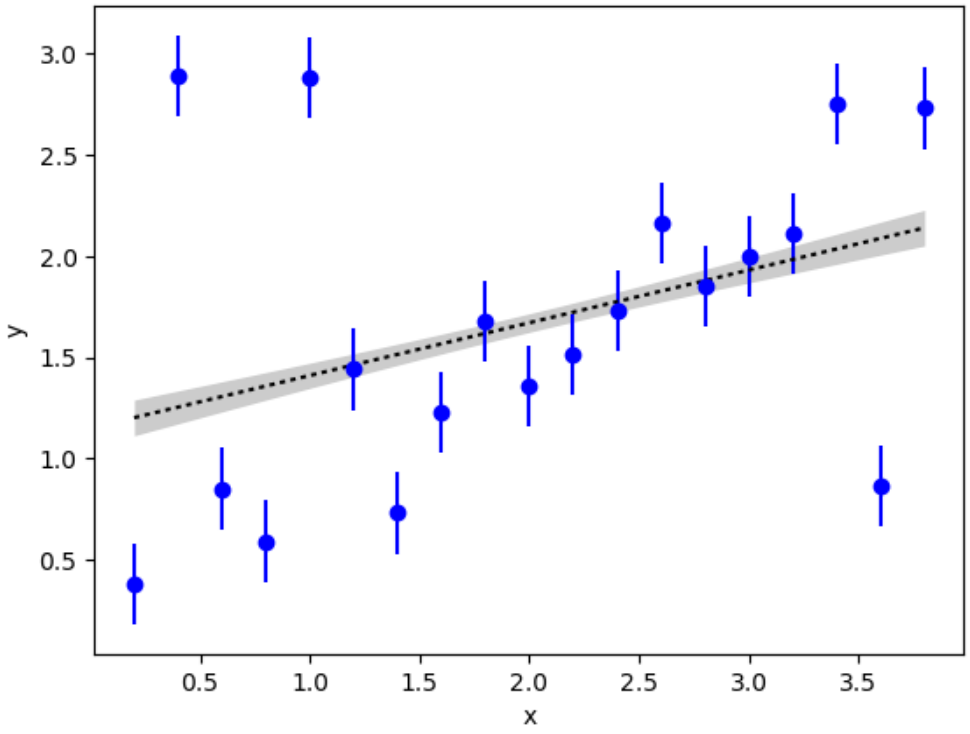
```
import matplotlib.pyplot as plt
import numpy as np

import gvar as gv
import lsqfit

def main():
    # least-squares fit to the data
    x = np.array([
        0.2, 0.4, 0.6, 0.8, 1.,
        1.2, 1.4, 1.6, 1.8, 2.,
        2.2, 2.4, 2.6, 2.8, 3.,
        3.2, 3.4, 3.6, 3.8
        ])
    y = gv.gvar([
        '0.38(20)', '2.89(20)', '0.85(20)', '0.59(20)', '2.88(20)',
        '1.44(20)', '0.73(20)', '1.23(20)', '1.68(20)', '1.36(20)',
        '1.51(20)', '1.73(20)', '2.16(20)', '1.85(20)', '2.00(20)',
        '2.11(20)', '2.75(20)', '0.86(20)', '2.73(20)'
        ])
    fit = lsqfit.nonlinear_fit(data=(x, y), prior=make_prior(), fcn=fitfcn)
    print(fit)

    # plot data
    plt.errorbar(x, gv.mean(y), gv.sdev(y), fmt='o', c='b')

    # plot fit function +- 1 sigma
    xline = np.linspace(x[0], x[-1], 100)
    yline = fitfcn(xline, fit.p)
    plt.plot(xline, gv.mean(yline), 'k--')
    yp = gv.mean(yline) + gv.sdev(yline)
    ym = gv.mean(yline) - gv.sdev(yline)
    plt.fill_between(xline,yp,ym,color='0.8')
```

```
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()

def fitfcn(x, p):
    c = p['c']
    return c[0] + c[1] * x

def make_prior():
    prior = gv.BufferDict(c=gv.gvar(['0(5)', '0(5)']))
    return prior


if __name__ == '__main__':
    main()
```

The fit is not good, with a `chi**2` per degree of freedom that is much larger than one, despite rather broad priors for the intercept and slope:

```
Least Square Fit:
  chi2/dof [dof] = 13 [19]    Q = 1.2e-40    logGBF = -117.45

Parameters:
            c 0   1.149 (95)      [  0.0 (5.0) ]
              1   0.261 (42)      [  0.0 (5.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0    (itns/time = 2/0.0)
```

The problem is evident if we plot the data:



At least three of the data points are outliers: they disagree with other nearby points by several standard deviations.

These outliers have a big impact on the fit (dashed line, with the gray band showing the ±1-sigma region). In particular they pull the `x=0` intercept (`fit.p['c'][0]`) up above one, while the rest of the data suggest an intercept of 0.5 or less.

4.2 A Solution

There are many *ad hoc* prescriptions for handling outliers. In the best of situations one would have an explanation for the outliers and seek to model them accordingly. For example, we might know that some fraction `w` of the time our detector malfunctions, resulting in much larger measurement errors than usual. This model can be represented by a more complicated probability density function (PDF) for the data that consists of a linear combination of the normal PDF with another PDF that is similar but with much larger errors. The relative weights assigned to these two terms would be `1-w` and `w`, respectively.

A modified data prior of this sort is incompatible with the least-squares code in *`lsqfit`*. Here we will incorporate it by replacing the least-squares analysis with a Bayesian integral, where the normal PDF is replaced a modified PDF of the sort described above. The complete code for this analysis is as follows:

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

import gvar as gv
import lsqfit

def main():
    ### 1) least-squares fit to the data
    x = np.array([
        0.2, 0.4, 0.6, 0.8, 1.,
        1.2, 1.4, 1.6, 1.8, 2.,
        2.2, 2.4, 2.6, 2.8, 3.,
        3.2, 3.4, 3.6, 3.8
        ])
    y = gv.gvar([
        '0.38(20)', '2.89(20)', '0.85(20)', '0.59(20)', '2.88(20)',
        '1.44(20)', '0.73(20)', '1.23(20)', '1.68(20)', '1.36(20)',
        '1.51(20)', '1.73(20)', '2.16(20)', '1.85(20)', '2.00(20)',
        '2.11(20)', '2.75(20)', '0.86(20)', '2.73(20)'
        ])
    prior = make_prior()
    fit = lsqfit.nonlinear_fit(data=(x, y), prior=prior, fcn=fitfcn, extend=True)
    print(fit)

    # plot data
    plt.errorbar(x, gv.mean(y), gv.sdev(y), fmt='o', c='b')

    # plot fit function
    xline = np.linspace(x[0], x[-1], 100)
    yline = fitfcn(xline, fit.p)
    plt.plot(xline, gv.mean(yline), 'k:')
    yp = gv.mean(yline) + gv.sdev(yline)
    ym = gv.mean(yline) - gv.sdev(yline)
    plt.fill_between(xline, yp, ym, color='0.8')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.savefig('case-outliers1.png', bbox_inches='tight')
    # plt.show()
```

```
    ### 2) Bayesian integral with modified PDF
    # modified probability density function
    mod_pdf = ModifiedPDF(data=(x, y), fcn=fitfcn, prior=prior)

    # integrator for expectation values with modified PDF
    expval = lsqfit.BayesIntegrator(fit, pdf=mod_pdf)

    # adapt integrator to pdf
    expval(neval=1000, nitn=15)

    # evaluate expectation value of g(p)
    def g(p):
        w = 0.5 + 0.5 * p['2w-1']
        c = p['c']
        return dict(w=[w, w**2], mean=c, outer=np.outer(c,c))

    results = expval(g, neval=1000, nitn=15, adapt=False)
    print(results.summary())

    # parameters c[i]
    mean = results['mean']
    cov = results['outer'] - np.outer(mean, mean)
    c = mean + gv.gvar(np.zeros(mean.shape), gv.mean(cov))
    print('c =', c)
    print(
        'corr(c) =',
        np.array2string(gv.evalcorr(c), prefix=10 * ' '),
        '\n',
        )

    # parameter w
    wmean, w2mean = results['w']
    wsdev = gv.mean(results['w'][1] - wmean ** 2) ** 0.5
    w = wmean + gv.gvar(np.zeros(np.shape(wmean)), wsdev)
    print('w =', w)

    # add new fit to plot
    yline = fitfcn(xline, dict(c=c))
    plt.plot(xline, gv.mean(yline), 'r--')
    yp = gv.mean(yline) + gv.sdev(yline)
    ym = gv.mean(yline) - gv.sdev(yline)
    plt.fill_between(xline, yp, ym, color='r', alpha=0.2)
    plt.show()

class ModifiedPDF:
    """ Modified PDF to account for measurement failure. """

    def __init__(self, data, fcn, prior):
        self.x, self.y = data
        self.fcn = fcn
        self.prior = prior

    def __call__(self, p):
        w = 0.5 + 0.5 * p['2w-1']
        y_fx = self.y - self.fcn(self.x, p)
        data_pdf1 = self.gaussian_pdf(y_fx, 1.)
        data_pdf2 = self.gaussian_pdf(y_fx, 10.)
        prior_pdf = self.gaussian_pdf(
```

```
            p.buf[:len(self.prior.buf)] - self.prior.buf
            )
        return np.prod((1. - w) * data_pdf1 + w * data_pdf2) * np.prod(prior_pdf)

    @staticmethod
    def gaussian_pdf(x, f=1.):
        xmean = gv.mean(x)
        xvar = gv.var(x) * f ** 2
        return gv.exp(-xmean ** 2 / 2. /xvar) / gv.sqrt(2 * np.pi * xvar)

def fitfcn(x, p):
    c = p['c']
    return c[0] + c[1] * x

def make_prior():
    prior = gv.BufferDict(c=gv.gvar(['0(5)', '0(5)']))
    prior['erfinv(2w-1)'] = gv.gvar('0(1)') / 2 ** 0.5
    return prior

if __name__ == '__main__':
    main()
```

Here class `ModifiedPDF` implements the modified PDF. As usual the PDF for the parameters (in `__call__`) is the product of a PDF for the data times a PDF for the priors. The data PDF is more complicated than usual, however, as it consists of two Gaussian distributions: one, `data_pdf1`, with the nominal data errors, and the other, `data_pdf2`, with errors that are ten times larger. Parameter `w` determines the relative weight of each data PDF.

The Bayesian integrals are estimated using *`lsqfit.BayesIntegrator`* `expval`, which is created from the least-squares fit output (`fit`). It is used to evaluate expectation values of arbitrary functions of the fit variables. Normally it would use the standard PDF from the least-squares fit, but we replace that PDF here with an instance (`mod_pdf`) of class `ModifiedPDF`.

We have modified `make_prior()` to introduce `2w-1` as a new fit parameter. The inverse error function of this parameter has a Gaussian prior (0±1)/sqrt(2), which makes `2w-1` uniformly distributed across the interval from -1 to 1 (and therefore `w` uniformly distributed between 0 and 1). This parameter has no role in the initial least-squares fit.

We first call `expval` with no function, to allow the integrator to adapt to the modified PDF. We then use the integrator, now with adaptation turned off (`adapt=False`), to evaluate the expectation value of function `g(p)`. The output dictionary `results` contains expectation values of the corresponding entries in the dictionary returned `g(p)`. These data allow us to calculate means, standard deviations and correlation matrices for the fit parameters.

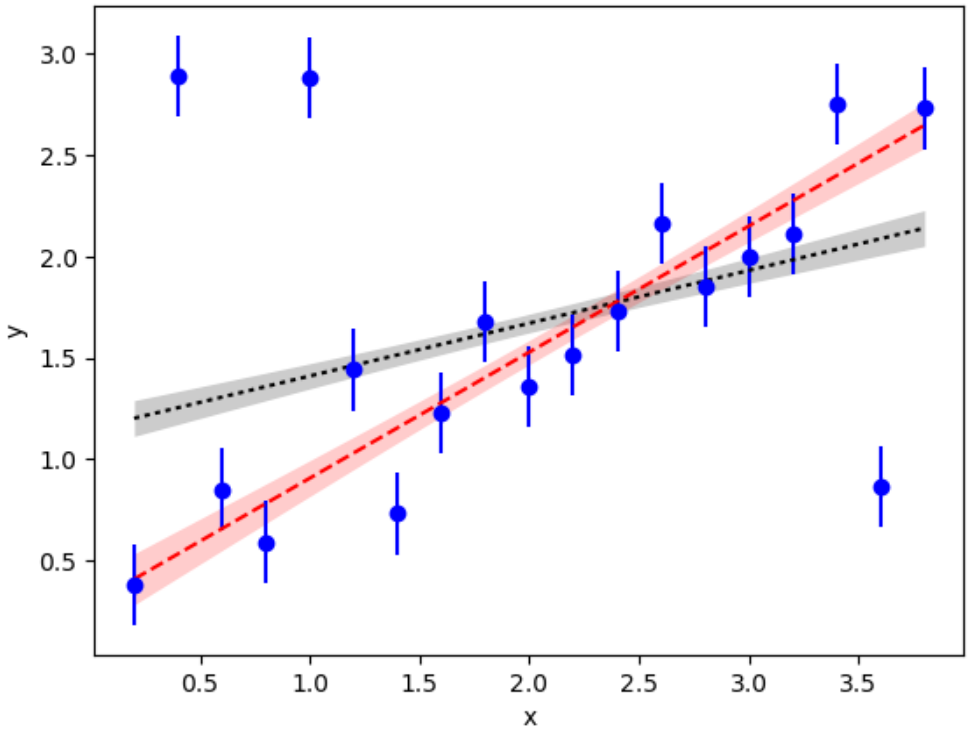
The results from this code are as follows:

```
Least Square Fit:
  chi2/dof [dof] = 13 [19]    Q = 1.2e-40    logGBF = -117.45

Parameters:
             c 0      1.149 (95)       [  0.0 (5.0) ]
               1      0.261 (42)       [  0.0 (5.0) ]
   erfinv(2w-1)    2e-16 +- 0.71       [  0.00 (71) ]
-------------------------------------------------------
           2w-1     3e-16 +- 0.8        [  0.00 (80) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0    (itns/time = 2/0.0)

itn   integral        average         chi2/dof        Q
-------------------------------------------------------
  1   6.82(11)e-11    6.82(11)e-11        0.00     1.00
```

```
  2    7.04(11)e-11     6.930(78)e-11        1.10      0.36
  3    6.775(76)e-11    6.878(58)e-11        0.97      0.49
  4    6.651(97)e-11    6.821(50)e-11        1.04      0.40
  5    6.74(10)e-11     6.806(45)e-11        0.95      0.55
  6    6.740(79)e-11    6.795(39)e-11        0.88      0.69
  7    6.763(87)e-11    6.790(36)e-11        0.93      0.63
  8    7.085(92)e-11    6.827(34)e-11        0.96      0.56
  9    6.873(68)e-11    6.832(31)e-11        0.95      0.59
 10    6.853(75)e-11    6.834(29)e-11        0.95      0.61
 11    6.79(11)e-11     6.830(28)e-11        0.89      0.76
 12    6.833(94)e-11    6.830(27)e-11        0.92      0.71
 13    6.806(81)e-11    6.828(26)e-11        0.93      0.67
 14    6.67(10)e-11     6.817(25)e-11        0.94      0.66
 15    6.725(93)e-11    6.811(24)e-11        0.90      0.77

c = [0.28(14) 0.622(58)]
corr(c) = [[ 1.          -0.90056919]
           [-0.90056919  1.        ]]

w = 0.26(11)

logBF = -23.4099(35)
```

The table after the fit shows results for the normalization of the modified PDF from each of the `nitn=15` iterations of the `vegas` algorithm used to estimate the integrals. The logarithm of the normalization (`logBF`) is -23.4, which is much larger than the value -117.5 of `logGBF` from the least-squares fit. This means that the data much prefer the modified prior (by a factor of `exp(-23.4 + 117.4)` or about 10^{41}.).

The new fit parameters are much more reasonable. In particular the intercept is 0.28(14) rather than the 1.15(10) from the least-squares fit. This is much better suited to the data (see the dashed line in red):



Note, from the correlation matrix, that the intercept and slope are anti-correlated, as one might guess for this fit. The

analysis also gives us an estimate for the failure rate `w=0.26(11)` of our detectors — they fail about a quarter of the time.

4.3 A Variation

Vanderplas in his version of this problem assigns a separate `w` to each data point. This is a slightly different model for the failure that leads to outliers. It is easily implemented here by changing the prior so that `2w-1` (and its inverse error function) is an array:

```
def make_prior():
    prior = gv.BufferDict(c=gv.gvar(['0(5)', '0(5)']))
    prior['erfinv(2w-1)'] = gv.gvar(19 * ['0(1)']) / 2 ** 0.5
    return prior
```

The Bayesian integral then has 21 parameters, rather than the 3 parameters before. The code still takes only 5–6 secs to run (on a 2014 laptop).

The final results are quite similar to the other model:

```
c = [0.30(16) 0.609(68)]
corr(c) = [[ 1.          -0.90919302]
           [-0.90919302  1.        ]]

w = [0.37(25) 0.67(23) 0.40(27) 0.35(27) 0.65(24) 0.49(30) 0.50(29) 0.35(25)
 0.44(27) 0.41(27) 0.37(26) 0.37(26) 0.41(27) 0.37(25) 0.38(26) 0.38(25)
 0.49(29) 0.65(25) 0.38(27)]

logBF = -24.164(63)
```

Note that the logarithm of the Bayes Factor `logBF` is slighly lower for this model than before. It is also less accurately determined (20x), because 21-parameter integrals are considerably more difficult than 3-parameter integrals. More precision can be obtained by increasing `neval`, but the current precision is more than adequate.

Only three of the `w[i]` values listed in the output are more than two standard deviations away from zero. Not surprisingly, these correspond to the unambiguous outliers.

The outliers in this case are pretty obvious; one is tempted to simply drop them. It is clearly better, however, to understand why they have occurred and to quantify the effect if possible, as above. Dropping outliers would be much more difficult if they were, say, three times closer to the rest of the data. The least-squares fit would still be poor (`chi**2` per degree of freedom of 3) and its intercept a bit too high (0.6(1)). Using the modified PDF, on the other hand, would give results very similar to what we obtained above: for example, the intercept would be 0.35(17).

CHAPTER

FIVE

LSQFIT - NONLINEAR LEAST SQUARES FITTING

5.1 Introduction

This package contains tools for nonlinear least-squares curve fitting of data. In general a fit has four inputs:

1. The dependent data `y` that is to be fit — typically `y` is a Python dictionary in an *lsqfit* analysis. Its values `y[k]` are either `gvar.GVar`s or arrays (any shape or dimension) of `gvar.GVar`s that specify the values of the dependent variables and their errors.
2. A collection `x` of independent data — `x` can have any structure and contain any data (or no data).
3. A fit function `f(x, p)` whose parameters `p` are adjusted by the fit until `f(x, p)` equals `y` to within `y`s errors — parameters *p* ' are usually specified by a dictionary whose values `p[k]` are individual parameters or (`numpy`) arrays of parameters. The fit function is assumed independent of `x` (that is, `f(p)`) if `x = False` (or if `x` is omitted from the input data).
4. Initial estimates or *priors* for each parameter in `p` — priors are usually specified using a dictionary `prior` whose values `prior[k]` are `gvar.GVar`s or arrays of `gvar.GVar`s that give initial estimates (values and errors) for parameters `p[k]`.

A typical code sequence has the structure:

```
... collect x, y, prior ...

def f(x, p):
    ... compute fit to y[k], for all k in y, using x, p ...
    ... return dictionary containing the fit values for the y[k]s ...

fit = lsqfit.nonlinear_fit(data=(x, y), prior=prior, fcn=f)
print(fit)      # variable fit is of type nonlinear_fit
```

The parameters `p[k]` are varied until the `chi**2` for the fit is minimized.

The best-fit values for the parameters are recovered after fitting using, for example, `p=fit.p`. Then the `p[k]` are `gvar.GVar`s or arrays of `gvar.GVar`s that give best-fit estimates and fit uncertainties in those estimates. The `print(fit)` statement prints a summary of the fit results.

The dependent variable `y` above could be an array instead of a dictionary, which is less flexible in general but possibly more convenient in simpler fits. Then the approximate `y` returned by fit function `f(x, p)` must be an array with the same shape as the dependent variable. The prior `prior` could also be represented by an array instead of a dictionary.

By default priors are Gaussian/normal distributions, represented by `gvar.GVar`s. Setting *nonlinear_fit* parameter `extend=True` allows for log-normal and sqrt-normal distributions as well. The latter are indicated by replacing the prior (in a dictionary prior) with key `c`, for example, by a prior for the parameter's logarithm or square root, with key `log(c)` or `sqrt(c)`, respectively. *nonlinear_fit* adds parameter `c` to the parameter dictionary, deriving its value from parameter `log(c)` or `sqrt(c)`. The fit function can be expressed directly in

terms of parameter `c` and so is the same no matter which distribution is used for `c`. Note that a sqrt-normal distribution with zero mean is equivalent to an exponential distribution. Additional distributions can be added using *gvar.add_parameter_distribution()*.

The *lsqfit* tutorial contains extended explanations and examples. The first appendix in the paper at http://arxiv.org/abs/arXiv:1406.2279 provides conceptual background on the techniques used in this module for fits and, especially, error budgets.

5.2 nonlinear_fit Objects

class `lsqfit.`**`nonlinear_fit`**(*data*, *fcn*, *prior=None*, *p0=None*, *extend=False*, *svdcut=1e-15*, *debug=False*, ***kargs*)

Nonlinear least-squares fit.

lsqfit.nonlinear_fit fits a (nonlinear) function `f(x, p)` to data `y` by varying parameters `p`, and stores the results: for example,

```
fit = nonlinear_fit(data=(x, y), fcn=f, prior=prior)    # do fit
print(fit)                                              # print fit results
```

The best-fit values for the parameters are in `fit.p`, while the `chi**2`, the number of degrees of freedom, the logarithm of Gaussian Bayes Factor, the number of iterations, and the cpu time needed for the fit are in `fit.chi2`, `fit.dof`, `fit.logGBF`, `fit.nit`, and `fit.time`, respectively. Results for individual parameters in `fit.p` are of type `gvar.GVar`, and therefore carry information about errors and correlations with other parameters. The fit data and prior can be recovered using `fit.x` (equals `False` if there is no `x`), `fit.y`, and `fit.prior`; the data and prior are corrected for the *svd* cut, if there is one (that is, their covariance matrices have been modified in accordance with the *svd* cut).

Parameters

- **`data`** – Data to be fit by *lsqfit.nonlinear_fit*. It can have any of the following formats:

  `data = x, y` `x` is the independent data that is passed to the fit function with the fit parameters: `fcn(x, p)`. `y` is a dictionary (or array) of `gvar.GVar`s that encode the means and covariance matrix for the data that is to be fit being fit. The fit function must return a result having the same layout as `y`.

  `data = y` `y` is a dictionary (or array) of `gvar.GVar`s that encode the means and covariance matrix for the data being fit. There is no independent data so the fit function depends only upon the fit parameters: `fit(p)`. The fit function must return a result having the same layout as `y`.

  `data = x, ymean, ycov` `x` is the independent data that is passed to the fit function with the fit parameters: `fcn(x, p)`. `ymean` is an array containing the mean values of the fit data. `ycov` is an array containing the covariance matrix of the fit data; `ycov.shape` equals `2*ymean.shape`. The fit function must return an array having the same shape as `ymean`.

  `data = x, ymean, ysdev` `x` is the independent data that is passed to the fit function with the fit parameters: `fcn(x, p)`. `ymean` is an array containing the mean values of the fit data. `ysdev` is an array containing the standard deviations of the fit data; `ysdev.shape` equals `ymean.shape`. The data are assumed to be uncorrelated. The fit function must return an array having the same shape as `ymean`.

  Setting `x=False` in the first, third or fourth of these formats implies that the fit function depends only on the fit parameters: that is, `fcn(p)` instead of `fcn(x, p)`. (This is not assumed if `x=None`.)

- **fcn** (*function*) – The function to be fit to `data`. It is either a function of the independent data `x` and the fit parameters `p` (`fcn(x, p)`), or a function of just the fit parameters (`fcn(p)`) when there is no `x` data or `x=False`. The parameters are tuned in the fit until the function returns values that agree with the `y` data to within the `y`s' errors. The function's return value must have the same layout as the `y` data (a dictionary or an array). The fit parameters `p` are either: 1) a dictionary where each `p[k]` is a single parameter or an array of parameters (any shape); or, 2) a single array of parameters. The layout of the parameters is the same as that of prior `prior` if it is specified; otherwise, it is inferred from of the starting value `p0` for the fit.
- **prior** (dictionary, array, or `None`) – A dictionary (or array) containing *a priori* estimates for all parameters `p` used by fit function `fcn(x, p)` (or `fcn(p)`). Fit parameters `p` are stored in a dictionary (or array) with the same keys and structure (or shape) as `prior`. The default value is `None`; `prior` must be defined if `p0` is `None`.
- **p0** (dictionary, array, string or `None`) – Starting values for fit parameters in fit. *`lsqfit.nonlinear_fit`* adjusts `p0` to make it consistent in shape and structure with `prior` when the latter is specified: elements missing from `p0` are filled in using `prior`, and elements in `p0` that are not in `prior` are discarded. If `p0` is a string, it is taken as a file name and *`lsqfit.nonlinear_fit`* attempts to read starting values from that file; best-fit parameter values are written out to the same file after the fit (for priming future fits). If `p0` is `None` or the attempt to read the file fails, starting values are extracted from `prior`. The default value is `None`; `p0` must be defined if `prior` is `None`.
- **svdcut** (`None` or `float`) – If `svdcut` is nonzero (not `None`), *svd* cuts are applied to every block-diagonal sub-matrix of the covariance matrix for the data `y` and `prior` (if there is a prior). The blocks are first rescaled so that all diagonal elements equal 1 – that is, the blocks are replaced by the correlation matrices for the corresponding subsets of variables. Then, if `svdcut > 0`, eigenvalues of the rescaled matrices that are smaller than `svdcut` times the maximum eigenvalue are replaced by `svdcut` times the maximum eigenvalue. This makes the covariance matrix less singular and less susceptible to roundoff error. When `svdcut < 0`, eigenvalues smaller than `|svdcut|` times the maximum eigenvalue are discarded and the corresponding components in `y` and `prior` are zeroed out.
- **extend** – Log-normal and sqrt-normal distributions can be used for fit priors when `extend=True`, provided the parameters are specified by a dictionary (as opposed to an array). To use such a distribution for a parameter `'c'` in the fit prior, replace `prior['c']` with a prior specifying its logarithm or square root, designated by `prior['log(c)']` or `prior['sqrt(c)']`, respectively. The dictionaries containing parameters generated by *`lsqfit.nonlinear_fit`* will have entries for both `'c'` and `'log(c)'` or `'sqrt(c)'`, so only the prior need be changed to switch to log-normal/sqrt-normal distributions. Setting `extend=False` (the default) restricts all parameters to Gaussian distributions. Additional distributions can be added using *`gvar.add_parameter_distribution()`*.
- **debug** (*boolean*) – Set to `True` for extra debugging of the fit function and a check for roundoff errors. (Default is `False`.)
- **fitterargs** – Dictionary of arguments passed on to *`lsqfit.multifit`*, which does the fitting.

The results from the fit are accessed through the following attributes (of `fit` where `fit = nonlinear_fit(...)`):

chi2
: The minimum `chi**2` for the fit. `fit.chi2 / fit.dof` is usually of order one in good fits; values much less than one suggest that the actual standard deviations in the input data and/or priors are smaller than the standard deviations used in the fit.

cov
Covariance matrix of the best-fit parameters from the fit.

dof
Number of degrees of freedom in the fit, which equals the number of pieces of data being fit when priors are specified for the fit parameters. Without priors, it is the number of pieces of data minus the number of fit parameters.

logGBF
The logarithm of the probability (density) of obtaining the fit data by randomly sampling the parameter model (priors plus fit function) used in the fit. This quantity is useful for comparing fits of the same data to different models, with different priors and/or fit functions. The model with the largest value of `fit.logGBF` is the one prefered by the data. The exponential of the difference in `fit.logGBF` between two models is the ratio of probabilities (Bayes factor) for those models. Differences in `fit.logGBF` smaller than 1 are not very significant. Gaussian statistics are assumed when computing `fit.logGBF`.

p
Best-fit parameters from fit. Depending upon what was used for the prior (or `p0`), it is either: a dictionary (`gvar.BufferDict`) of `gvar.GVar`s and/or arrays of `gvar.GVar`s; or an array (`numpy.ndarray`) of `gvar.GVar`s. `fit.p` represents a multi-dimensional Gaussian distribution which, in Bayesian terminology, is the *posterior* probability distribution of the fit parameters.

pmean
Means of the best-fit parameters from fit (dictionary or array).

psdev
Standard deviations of the best-fit parameters from fit (dictionary or array).

palt
Same as `fit.p` except that the errors are computed directly from `fit.cov`. This is faster but means that no information about correlations with the input data is retained (unlike in `fit.p`); and, therefore, `fit.palt` cannot be used to generate error budgets. `fit.p` and `fit.palt` give the same means and normally give the same errors for each parameter. They differ only when the input data's covariance matrix is too singular to invert accurately (because of roundoff error), in which case an SVD cut is advisable.

p0
The parameter values used to start the fit.

Q
The probability that the `chi**2` from the fit could have been larger, by chance, assuming the best-fit model is correct. Good fits have `Q` values larger than 0.1 or so. Also called the *p-value* of the fit.

svdcorrection
The sum of all SVD corrections, if any, added to the fit data `y` or the prior `prior`.

svdn
The number of eignemodes modified (and/or deleted) by the SVD cut.

nblocks
A dictionary where `nblocks[s]` equals the number of block-diagonal sub-matrices of the `y–prior` covariance matrix that are size `s`-by-`s`. This is sometimes useful for debugging.

time
CPU time (in secs) taken by fit.

The input parameters to the fit can be accessed as attributes. Note in particular attributes:

prior
Prior used in the fit. This may differ from the input prior if an SVD cut is used. It is either a dictionary (`gvar.BufferDict`) or an array (`numpy.ndarray`), depending upon the input. Equals `None` if no prior was specified.

x

The first field in the input `data`. This is sometimes the independent variable (as in 'y vs x' plot), but may be anything. It is set equal to `False` if the `x` field is omitted from the input `data`. (This also means that the fit function has no `x` argument: so `f(p)` rather than `f(x,p)`.)

y

Fit data used in the fit. This may differ from the input data if an SVD cut is used. It is either a dictionary (`gvar.BufferDict`) or an array (`numpy.ndarray`), depending upon the input.

Additional methods are provided for printing out detailed information about the fit, testing fits with simulated data, doing bootstrap analyses of the fit errors, dumping (for later use) and loading parameter values, and checking for roundoff errors in the final error estimates:

format(*maxline=0*, *pstyle='v'*)

Formats fit output details into a string for printing.

The output tabulates the `chi**2` per degree of freedom of the fit (`chi2/dof`), the number of degrees of freedom, the logarithm of the Gaussian Bayes Factor for the fit (`logGBF`), and the number of fit- algorithm iterations needed by the fit. Optionally, it will also list the best-fit values for the fit parameters together with the prior for each (in `[]` on each line). Lines for parameters that deviate from their prior by more than one (prior) standard deviation are marked with asterisks, with the number of asterisks equal to the number of standard deviations (up to five). `format` can also list all of the data and the corresponding values from the fit, again with asterisks on lines where there is a significant discrepancy. At the end it lists the SVD cut, the number of eigenmodes modified by the SVD cut, the tolerances used in the fit, and the time in seconds needed to do the fit. The tolerance used to terminate the fit is marked with an asterisk.

Parameters

- **maxline** (*integer or bool*) – Maximum number of data points for which fit results and input data are tabulated. `maxline<0` implies that only `chi2`, `Q`, `logGBF`, and `itns` are tabulated; no parameter values are included. Setting `maxline=True` prints all data points; setting it equal `None` or `False` is the same as setting it equal to `-1`. Default is `maxline=0`.
- **pstyle** (*'vv', 'v', or 'm'*) – Style used for parameter list. Supported values are 'vv' for very verbose, 'v' for verbose, and 'm' for minimal. When 'm' is set, only parameters whose values differ from their prior values are listed.

Returns String containing detailed information about fit.

fmt_errorbudget(*outputs*, *inputs*, *ndecimal=2*, *percent=True*)

Tabulate error budget for `outputs[ko]` due to `inputs[ki]`.

For each output `outputs[ko]`, `fmt_errorbudget` computes the contributions to `outputs[ko]`'s standard deviation coming from the `gvar.GVar`s collected in `inputs[ki]`. This is done for each key combination `(ko,ki)` and the results are tabulated with columns and rows labeled by `ko` and `ki`, respectively. If a `gvar.GVar` in `inputs[ki]` is correlated with other `gvar.GVar`s, the contribution from the others is included in the `ki` contribution as well (since contributions from correlated `gvar.GVar`s cannot be distinguished). The table is returned as a string.

Parameters

- **outputs** – Dictionary of `gvar.GVar`s for which an error budget is computed.
- **inputs** – Dictionary of: `gvar.GVar`s, arrays/dictionaries of `gvar.GVar`s, or lists of `gvar.GVar`s and/or arrays/dictionaries of `gvar.GVar`s. `fmt_errorbudget` tabulates the parts of the standard deviations of each `outputs[ko]` due to each `inputs[ki]`.
- **ndecimal** (`int`) – Number of decimal places displayed in table.

- **percent** (*boolean*) – Tabulate % errors if `percent is True`; otherwise tabulate the errors themselves.
- **colwidth** (*positive integer or None*) – Width of each column. This is set automatically, to accommodate label widths, if `colwidth=None` (default).
- **verify** (*boolean*) – If `True`, a warning is issued if: 1) different inputs are correlated (and therefore double count errors); or 2) the sum (in quadrature) of partial errors is not equal to the total error to within 0.1% of the error (and the error budget is incomplete or overcomplete). No checking is done if `verify==False` (default).

Returns A table (`str`) containing the error budget. Output variables are labeled by the keys in `outputs` (columns); sources of uncertainty are labeled by the keys in `inputs` (rows).

fmt_values(*outputs*, *ndecimal=None*)

Tabulate `gvar.GVars` in `outputs`.

Parameters

- **outputs** – A dictionary of `gvar.GVar` objects.
- **ndecimal** (`int` or `None`) – Format values `v` using `v.fmt(ndecimal)`.

Returns A table (`str`) containing values and standard deviations for variables in `outputs`, labeled by the keys in `outputs`.

simulated_fit_iter(*n=None*, *pexact=None*, ***kargs*)

Iterator that returns simulation copies of a fit.

Fit reliability can be tested using simulated data which replaces the mean values in `self.y` with random numbers drawn from a distribution whose mean equals `self.fcn(pexact)` and whose covariance matrix is the same as `self.y`'s. Simulated data is very similar to the original fit data, `self.y`, but corresponds to a world where the correct values for the parameters (*i.e.*, averaged over many simulated data sets) are given by `pexact`. `pexact` is usually taken equal to `fit.pmean`.

Each iteration of the iterator creates new simulated data, with different random numbers, and fits it, returning the the *lsqfit.nonlinear_fit* that results. The simulated data has the same covariance matrix as `fit.y`. Typical usage is:

```
...
fit = nonlinear_fit(...)
...
for sfit in fit.simulated_fit_iter(n=3):
    ... verify that sfit.p agrees with pexact=fit.pmean within errors ...
```

Only a few iterations are needed to get a sense of the fit's reliability since we know the correct answer in each case. The simulated fit's output results should agree with `pexact` (=`fit.pmean` here) within the simulated fit's errors.

Simulated fits can also be used to estimate biases in the fit's output parameters or functions of them, should non-Gaussian behavior arise. This is possible, again, because we know the correct value for every parameter before we do the fit. Again only a few iterations may be needed to reliable estimates.

The (possibly non-Gaussian) probability distributions for parameters, or functions of them, can be explored in more detail by setting option `bootstrap=True` and collecting results from a large number of simulated fits. With `bootstrap=True`, the means of the priors are also varied from fit to fit, as in a bootstrap simulation; the new prior means are chosen at random from the prior distribution. Variations in the best-fit parameters (or functions of them) from fit to fit define the probability distributions for those quantities. For example, one would use the following code to analyze the distribution of function `g(p)` of the fit parameters:

```
        fit = nonlinear_fit(...)

        ...

        glist = []
        for sfit in fit.simulated_fit_iter(n=100, bootstrap=True):
            glist.append(g(sfit.pmean))

        ... analyze samples glist[i] from g(p) distribution ...
```

This code generates `n=100` samples `glist[i]` from the probability distribution of `g(p)`. If everything is Gaussian, the mean and standard deviation of `glist[i]` should agree with `g(fit.p).mean` and `g(fit.p).sdev`.

The only difference between simulated fits with `bootstrap=True` and `bootstrap=False` (the default) is that the prior means are varied. It is essential that they be varied in a bootstrap analysis since one wants to capture the impact of the priors on the final distributions, but it is not necessary and probably not desirable when simply testing a fit's reliability.

Parameters

- **n** (integer or `None`) – Maximum number of iterations (equals infinity if `None`).
- **pexact** (`None` or array or dictionary of numbers) – Fit-parameter values for the underlying distribution used to generate simulated data; replaced by `self.pmean` if is `None` (default).
- **bootstrap** (*bool*) – Vary prior means if `True`; otherwise vary only the means in `self.y` (default).

Returns An iterator that returns *lsqfit.nonlinear_fit*s for different simulated data.

Note that additional keywords can be added to overwrite keyword arguments in *lsqfit.nonlinear_fit*.

bootstrap_iter(*n=None*, *datalist=None*)

Iterator that returns bootstrap copies of a fit.

A bootstrap analysis involves three steps: 1) make a large number of "bootstrap copies" of the original input data and prior that differ from each other by random amounts characteristic of the underlying randomness in the original data; 2) repeat the entire fit analysis for each bootstrap copy of the data, extracting fit results from each; and 3) use the variation of the fit results from bootstrap copy to bootstrap copy to determine an approximate probability distribution (possibly non-gaussian) for the fit parameters and/or functions of them: the results from each bootstrap fit are samples from that distribution.

Bootstrap copies of the data for step 2 are provided in `datalist`. If `datalist` is `None`, they are generated instead from the means and covariance matrix of the fit data (assuming gaussian statistics). The maximum number of bootstrap copies considered is specified by `n` (`None` implies no limit).

Variations in the best-fit parameters (or functions of them) from bootstrap fit to bootstrap fit define the probability distributions for those quantities. For example, one could use the following code to analyze the distribution of function `g(p)` of the fit parameters:

```
        fit = nonlinear_fit(...)

        ...

        glist = []
        for sfit in fit.bootstrapped_fit_iter(
            n=100, datalist=datalist, bootstrap=True
            ):
```

```
        glist.append(g(sfit.pmean))

    ... analyze samples glist[i] from g(p) distribution ...
```

This code generates `n=100` samples `glist[i]` from the probability distribution of `g(p)`. If everything is Gaussian, the mean and standard deviation of `glist[i]` should agree with `g(fit.p).mean` and `g(fit.p).sdev`.

Parameters

- **n** (*integer*) – Maximum number of iterations if `n` is not `None`; otherwise there is no maximum.
- **datalist** (sequence or iterator or `None`) – Collection of bootstrap `data` sets for fitter.

Returns Iterator that returns an *lsqfit.nonlinear_fit* object containing results from the fit to the next data set in `datalist`

dump_p (*filename*)

Dump parameter values (`fit.p`) into file `filename`.

`fit.dump_p(filename)` saves the best-fit parameter values (`fit.p`) from a `nonlinear_fit` called `fit`. These values are recovered using `p = nonlinear_fit.load_parameters(filename)` where `p`'s layout is the same as that of `fit.p`.

dump_pmean (*filename*)

Dump parameter means (`fit.pmean`) into file `filename`.

`fit.dump_pmean(filename)` saves the means of the best-fit parameter values (`fit.pmean`) from a `nonlinear_fit` called `fit`. These values are recovered using `p0 = nonlinear_fit.load_parameters(filename)` where `p0`'s layout is the same as `fit.pmean`. The saved values can be used to initialize a later fit (`nonlinear_fit` parameter `p0`).

static load_parameters (*filename*)

Load parameters stored in file `filename`.

`p = nonlinear_fit.load_p(filename)` is used to recover the values of fit parameters dumped using `fit.dump_p(filename)` (or `fit.dump_pmean(filename)`) where `fit` is of type *lsqfit.nonlinear_fit*. The layout of the returned parameters `p` is the same as that of `fit.p` (or `fit.pmean`).

check_roundoff (*rtol=0.25*, *atol=1e-6*)

Check for roundoff errors in fit.p.

Compares standard deviations from fit.p and fit.palt to see if they agree to within relative tolerance `rtol` and absolute tolerance `atol`. Generates a warning if they do not (in which case an *svd* cut might be advisable).

5.3 Functions

`lsqfit.`**empbayes_fit** (*z0*, *fitargs*, ***minargs*)

Call `lsqfit.nonlinear_fit(**fitargs(z))` varying `z`, starting at `z0`, to maximize `logGBF` (empirical Bayes procedure).

The fit is redone for each value of `z` that is tried, in order to determine `logGBF`.

Parameters

- **z0** (*array*) – Starting point for search.

- **fitargs** (*function*) – Function of array `z` that determines which fit parameters to use. The function returns these as an argument dictionary for *lsqfit.nonlinear_fit()*.
- **minargs** (*dictionary*) – Optional argument dictionary, passed on to *lsqfit.multiminex*, which finds the minimum.

Returns A tuple containing the best fit (object of type *lsqfit.nonlinear_fit*) and the optimal value for parameter `z`.

`lsqfit.`**wavg**(*dataseq*, *prior=None*, *fast=False*, ***kargs*)

Weighted average of `gvar.GVar`s or arrays/dicts of `gvar.GVar`s.

The weighted average of several `gvar.GVar`s is what one obtains from a least-squares fit of the collection of `gvar.GVar`s to the one-parameter fit function

```
def f(p):
    return N * [p[0]]
```

where `N` is the number of `gvar.GVar`s. The average is the best-fit value for `p[0]`. `gvar.GVar`s with smaller standard deviations carry more weight than those with larger standard deviations. The averages computed by `wavg` take account of correlations between the `gvar.GVar`s.

If `prior` is not `None`, it is added to the list of data used in the average. Thus `wavg([x2, x3], prior=x1)` is the same as `wavg([x1, x2, x3])`.

Typical usage is

```
x1 = gvar.gvar(...)
x2 = gvar.gvar(...)
x3 = gvar.gvar(...)
xavg = wavg([x1, x2, x3])    # weighted average of x1, x2 and x3
```

where the result `xavg` is a `gvar.GVar` containing the weighted average.

The individual `gvar.GVar`s in the last example can be replaced by multidimensional distributions, represented by arrays of `gvar.GVar`s or dictionaries of `gvar.GVar`s (or arrays of `gvar.GVar`s). For example,

```
x1 = [gvar.gvar(...), gvar.gvar(...)]
x2 = [gvar.gvar(...), gvar.gvar(...)]
x3 = [gvar.gvar(...), gvar.gvar(...)]
xavg = wavg([x1, x2, x3])
    # xavg[i] is wgtd avg of x1[i], x2[i], x3[i]
```

where each array `x1`, `x2` ... must have the same shape. The result `xavg` in this case is an array of `gvar.GVar`s, where the shape of the array is the same as that of `x1`, etc.

Another example is

```
x1 = dict(a=[gvar.gvar(...), gvar.gvar(...)], b=gvar.gvar(...))
x2 = dict(a=[gvar.gvar(...), gvar.gvar(...)], b=gvar.gvar(...))
x3 = dict(a=[gvar.gvar(...), gvar.gvar(...)])
xavg = wavg([x1, x2, x3])
    # xavg['a'][i] is wgtd avg of x1['a'][i], x2['a'][i], x3['a'][i]
    # xavg['b'] is gtd avg of x1['b'], x2['b']
```

where different dictionaries can have (some) different keys. Here the result `xavg` is a `gvar.BufferDict` ‘ having the same keys as `x1`, etc.

Weighted averages can become costly when the number of random samples being averaged is large (100s or more). In such cases it might be useful to set parameter `fast=True`. This causes `wavg` to estimate the weighted average by incorporating the random samples one at a time into a running average:

```
    result = prior
    for dataseq_i in dataseq:
        result = wavg([result, dataseq_i], ...)
```

This method is much faster when `len(dataseq)` is large, and gives the exact result when there are no correlations between different elements of list `dataseq`. The results are approximately correct when `dataseq[i]` and `dataseq[j]` are correlated for `i!=j`.

Parameters

- **dataseq** – The `gvar.GVar`s to be averaged. `dataseq` is a one-dimensional sequence of `gvar.GVar`s, or of arrays of `gvar.GVar`s, or of dictionaries containing `gvar.GVar`s or arrays of `gvar.GVar`s. All `dataseq[i]` must have the same shape.
- **prior** (`gvar.GVar` or array/dictionary of `gvar.GVar`s) – Prior values for the averages, to be included in the weighted average. Default value is `None`, in which case `prior` is ignored.
- **fast** (*bool*) – Setting `fast=True` causes `wavg` to compute an approximation to the weighted average that is much faster to calculate when averaging a large number of samples (100s or more). The default is `fast=False`.
- **kargs** (*dict*) – Additional arguments (e.g., `svdcut`) to the fitter used to do the averaging.

Results returned by `gvar.wavg()` have the following extra attributes describing the average:

`lsqfit.`**chi2**
`chi**2` for weighted average.

`lsqfit.`**dof**
Effective number of degrees of freedom.

`lsqfit.`**Q**
The probability that the `chi**2` could have been larger, by chance, assuming that the data are all Gaussain and consistent with each other. Values smaller than 0.1 or suggest that the data are not Gaussian or are inconsistent with each other. Also called the *p-value*.

Quality factor *Q* (or *p-value*) for fit.

`lsqfit.`**time**
Time required to do average.

`lsqfit.`**svdcorrection**
The *svd* corrections made to the data when `svdcut` is not `None`.

`lsqfit.`**fit**
Fit output from average.

`lsqfit.`**gammaQ**`()`
Return the normalized incomplete gamma function `Q(a,x) = 1-P(a,x)`.

```
Q(a, x) = 1/Gamma(a) * \int_x^\infty dt exp(-t) t ** (a-1) = 1 - P(a, x)
```

Note that `gammaQ(ndof/2., chi2/2.)` is the probabilty that one could get a `chi**2` larger than `chi2` with `ndof` degrees of freedom even if the model used to construct `chi2` is correct.

`gvar.`**add_parameter_distribution**`()`
Add new parameter distribution for use in fits.

This function adds new distributions for the parameters used in *`lsqfit.nonlinear_fit`*. For example, the code

```
import gvar as gv
gv.add_parameter_distribution('log', gv.exp)
```

enables the use of log-normal distributions for parameters. The log-normal distribution is invoked for a parameter `p` by including `log(p)` rather than `p` itself in the fit prior. log-normal, sqrt-normal, and erfinv-normal distributions are included by default. (Setting a prior `prior[erfinv(w)]` equal to `gv.gvar('0(1)') / gv.sqrt(2)` means that the prior probability for `w` is distributed uniformly between -1 and 1, and is zero elsewhere.)

These distributions are implemented by replacing a fit parameter `p` by a new fit parameter `fcn(p)` where `fcn` is some function. `fcn(p)` is assumed to have a Gaussian distribution, and parameter `p` is recovered using the inverse function `invfcn` where `p=invfcn(fcn(p))`.

Parameters

- **name** (*str*) – Distribution's name.
- **invfcn** – Inverse of the transformation function.

`gvar.`**del_parameter_distribution**`()`
Delete parameter distribution `name`.

`gvar.`**add_parameter_parentheses**`()`
Return dictionary with proper keys for parameter distributions (legacy code).

This utility function helps fix legacy code that uses parameter keys like `logp` or `sqrtp` instead of `log(p)` or `sqrt(p)`, as now required. This method creates a copy of dictionary `p'' but with keys like ''logp` or `sqrtp` replaced by `log(p)` or `sqrt(p)`. So setting

```
p = add_parameter_parentheses(p)
```

fixes the keys in `p` for log-normal and sqrt-normal parameters.

5.4 Classes for Bayesian Integrals

class `lsqfit.`**BayesPDF**(*fit*, *svdcut=1e-15*)
Bayesian probability density function corresponding to *nonlinear_fit* `fit`.

The probability density function is the exponential of `-1/2` times the `chi**2` function (data and priors) used in `fit` divided by `norm`.

Parameters

- **fit** – Fit from *nonlinear_fit*.
- **svdcut** (*non-negative float or None*) – If not `None`, replace covariance matrix of `g` with a new matrix whose small eigenvalues are modified: eigenvalues smaller than `svdcut` times the maximum eigenvalue `eig_max` are replaced by `svdcut*eig_max`. This can ameliorate problems caused by roundoff errors when inverting the covariance matrix. It increases the uncertainty associated with the modified eigenvalues and so is conservative. Setting `svdcut=None` or `svdcut=0` leaves the covariance matrix unchanged. Default is `1e-15`.

__call__(*p*)
Probability density function evaluated at `p`.

logpdf(*p*)
Logarithm of the probability density function evaluated at `p`.

class lsqfit.**BayesIntegrator**(*fit*, *limit=1e15*, *scale=1*, *pdf=None*, *svdcut=1e-15*)

vegas integrator for Bayesian fit integrals.

Parameters

- **fit** – Fit from *nonlinear_fit*.
- **limit** (*positive float*) – Limits the integrations to a finite region of size limit times the standard deviation on either side of the mean. This can be useful if the functions being integrated misbehave for large parameter values (e.g., numpy.exp overflows for a large range of arguments). Default is 1e15.
- **scale** (*positive float*) – The integration variables are rescaled to emphasize parameter values of order scale times the corresponding standard deviations. The rescaling does not change the value of the integral but it can reduce uncertainties in the vegas estimates. Default is 1.0.
- **pdf** (*function*) – Probability density function pdf(p) of the fit parameters to use in place of the normal PDF associated with the least-squares fit used to create the integrator.
- **svdcut** (*non-negative float or None*) – If not None, replace covariance matrix of g with a new matrix whose small eigenvalues are modified: eigenvalues smaller than svdcut times the maximum eigenvalue eig_max are replaced by svdcut*eig_max. This can ameliorate problems caused by roundoff errors when inverting the covariance matrix. It increases the uncertainty associated with the modified eigenvalues and so is conservative. Setting svdcut=None or svdcut=0 leaves the covariance matrix unchanged. Default is 1e-15.

BayesIntegrator(fit) is a vegas integrator that evaluates expectation values for the multi-dimensional Bayesian distribution associated with *nonlinear_fit* fit: the probability density function is the exponential of the chi**2 function (times -1/2), for data and priors, used in the fit. For linear fits, it is equivalent to vegas.PDFIntegrator(fit.p), since the chi**2 function is quadratic in the fit parameters; but they can differ significantly for nonlinear fits.

BayesIntegrator integrates over the entire parameter space but first reexpresses the integrals in terms of variables that diagonalize the covariance matrix of the best-fit parameters fit.p from *nonlinear_fit* and are centered at the best-fit values. This greatly facilitates the integration using vegas, making integrals over 10s or more of parameters feasible. (The vegas module must be installed separately in order to use BayesIntegrator.)

A simple illustration of BayesIntegrator is given by the following code, which we use to evaluate the mean and standard deviation for s*g where s and g are fit parameters:

```
import lsqfit
import gvar as gv
import numpy as np

# least-squares fit
x = np.array([0.1, 1.2, 1.9, 3.5])
y = gv.gvar(['1.2(1.0)', '2.4(1)', '2.0(1.2)', '5.2(3.2)'])
prior = gv.gvar(dict(a='0(5)', s='0(2)', g='2(2)'))
def f(x, p):
    return p['a'] + p['s'] * x ** p['g']
fit = lsqfit.nonlinear_fit(data=(x,y), prior=prior, fcn=f, debug=True)
print(fit)

# Bayesian integral to evaluate expectation value of s*g
def g(p):
    sg = p['s'] * p['g']
    return [sg, sg**2]
```

```
expval = lsqfit.BayesIntegrator(fit, limit=20.)
warmup = expval(neval=4000, nitn=10)
results = expval(g, neval=4000, nitn=15, adapt=False)
print(results.summary())
print('results =', results, '\n')
sg, sg2 = results
sg_sdev = (sg2 - sg**2) ** 0.5
print('s*g from Bayes integral:  mean =', sg, '  sdev =', sg_sdev)
print('s*g from fit:', fit.p['s'] * fit.p['g'])
```

where the `warmup` calls to the integrator are used to adapt it to probability density function from the fit, and then the integrator is used to evaluate the expectation value of `g(p)`, which is returned in array `results`. Here `neval` is the (approximate) number of function calls per iteration of the `vegas` algorithm and `nitn` is the number of iterations. We use the integrator to calculated the expectation value of `s*g` and `(s*g)**2` so we can compute a mean and standard deviation.

The output from this code shows that the Gaussian approximation for `s*g` (0.76(66)) is somewhat different from the result obtained from a Bayesian integral (0.48(54)):

```
Least Square Fit:
  chi2/dof [dof] = 0.32 [4]    Q = 0.87    logGBF = -9.2027

Parameters:
              a    1.61 (90)      [  0.0 (5.0) ]
              s    0.62 (81)      [  0.0 (2.0) ]
              g    1.2 (1.1)      [  2.0 (2.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 0.0001/0*    (itns/time = 10/0.0)

itn   integral        average         chi2/dof        Q
-------------------------------------------------------
  1   1.034(21)       1.034(21)           0.00     1.00
  2   1.034(21)       1.034(15)           0.56     0.64
  3   1.024(18)       1.030(12)           0.37     0.90
  4   1.010(18)       1.0254(98)          0.47     0.89
  5   1.005(17)       1.0213(85)          0.55     0.88
  6   1.013(19)       1.0199(78)          0.69     0.80
  7   0.987(16)       1.0152(70)          0.78     0.72
  8   1.002(18)       1.0135(66)          0.90     0.59
  9   1.036(20)       1.0160(62)          0.86     0.66
 10   1.060(20)       1.0204(60)          0.94     0.55

results = [0.4837(32) 0.5259(47)]

s*g from Bayes integral:  mean = 0.4837(32)   sdev = 0.5403(25)
s*g from fit: 0.78(66)
```

The table shows estimates of the probability density function's normalization from each of the `vegas` iterations used by the integrator to estimate the final results.

In general functions being integrated can return a number, or an array of numbers, or a dictionary whose values are numbers or arrays of numbers. This allows multiple expectation values to be evaluated simultaneously.

See the documentation with the `vegas` module for more details on its use, and on the attributes and methods associated with integrators. The example above sets `adapt=False` when computing final results. This gives more reliable error estimates when `neval` is small. Note that `neval` may need to be much larger (tens or hundreds of thousands) for more difficult high-dimension integrals.

__call__ (*f=None*, *mpi=False*, *pdf=None*, ***kargs*)
Estimate expectation value of function `f(p)`.

Uses multi-dimensional integration modules `vegas` to estimate the expectation value of `f(p)` with respect to the probability density function associated with *nonlinear_fit* `fit`.

Parameters

- **f** (*function*) – Function `f(p)` to integrate. Integral is the expectation value of the function with respect to the distribution. The function can return a number, an array of numbers, or a dictionary whose values are numbers or arrays of numbers.
- **pdf** (*function*) – Probability density function `pdf(p)` of the fit parameters to use in place of the normal PDF associated with the least-squares fit used to create the integrator.

All other keyword arguments are passed on to a `vegas` integrator; see the `vegas` documentation for further information.

5.5 Other Classes

class `lsqfit.`**multifit** (*x0*, *n*, *f*, *tol=1e-4*, *maxit=1000*, *alg='lmsder'*, *analyzer=None*)
Fitter for nonlinear least-squares multidimensional fits.

Parameters

- **x0** (`numpy` array of floats) – Starting point for minimization.
- **n** (*positive integer*) – Length of vector returned by the fit function `f(x)`.
- **f** (*function*) – Fit function: *multifit* minimizes `sum_i f_i(x)**2` by varying parameters `x`. The parameters are a 1-d `numpy` array of either numbers or `gvar.GVar`s.
- **tol** (*tuple or float*) – Setting `tol=(reltol, abstol)` causes the fit to stop searching for a solution when `|dx_i| <= abstol + reltol * |x_i|`. With version 2 or higher of the GSL library, `tol=(xtol, gtol, ftol)` can be used, where the fit stops when any one of the following three criteria is satisfied:

  1. step size small: `|dx_i| <= xtol * (xtol + |x_i|)`;
  2. gradient small: `||g . x||_inf <= gtol * ||f||^2`;
  3. residuals small: `||f(x+dx) - f(x)|| <= ftol * max(||f(x)||, 1)`.

  Recommended values are: `xtol=1/10**d` for `d` digits of precision in the parameters; `gtol=1e-6` to account for roundoff errors in gradient `g` (unless the second order derivative vanishes at minimum as well, in which case `gtol=0` might be good); and `ftol<<1`. Setting `tol=reltol` is equivalent to setting `tol=(reltol, 0.0)`. The default setting is `tol=0.0001`.
- **maxit** (*integer*) – Maximum number of iterations in search for minimum; default is 1000.
- **alg** (*string*) – *GSL* algorithm to use for minimization. Two options are currently available: `"lmsder"`, the scaled *LMDER* algorithm (default); and `"lmder"`, the unscaled *LMDER* algorithm. With version 2 of the GSL library, another option is `"lmniel"`, which can be useful when there is much more data than parameters.
- **analyzer** (*function*) – Optional function of `x, [...f_i(x)...], [[..df_ij(x)..]]` which is called after each iteration. This can be used to inspect intermediate steps in the minimization, if needed.

multifit is a function-class whose constructor does a least squares fit by minimizing `sum_i f_i(x)**2` as a function of vector `x`. The following attributes are available:

x
Location of the most recently computed (best) fit point.

cov
Covariance matrix at the minimum point.

f
The fit function `f(x)` at the minimum in the most recent fit.

J
Gradient `J_ij = df_i/dx[j]` for most recent fit.

nit
Number of iterations used in last fit to find the minimum.

stopping_criterion
Criterion used to stop fit: 0 => didn't converge 1 => step size small 2 => gradient small 3 => residuals small

error
`None` if fit successful; an error message otherwise.

multifit is a wrapper for the `multifit` *GSL* routine.

class `lsqfit.`**multiminex**(*x0*, *f*, *tol=1e-4*, *maxit=1000*, *step=1*, *alg='nmsimplex2'*, *analyzer=None*)
Minimizer for multidimensional functions.

Parameters

- **x0** (`numpy` array of floats) – Starting point for minimization search.
- **f** (*function*) – Function `f(x)` to be minimized by varying vector `x`.
- **tol** (*float*) – Minimization stops when `x` has converged to with tolerance `tol`; default is `1e-4`.
- **maxit** (*integer*) – Maximum number of iterations in search for minimum; default is 1000.
- **step** (*number*) – Initial step size to use in varying components of `x`; default is 1.
- **alg** (*string*) – *GSL* algorithm to use for minimization. Three options are currently available: `"nmsimplex"`, Nelder Mead Simplex algorithm; `"nmsimplex2"`, an improved version of `"nmsimplex"` (default); and `"nmsimplex2rand"`, a version of `"nmsimplex2"` with random shifts in the start position.
- **analyzer** (*function*) – Optional function of `x, f(x), it`, where `it` is the iteration number, which is called after each iteration. This can be used to inspect intermediate steps in the minimization, if needed.

multiminex is a function-class whose constructor minimizes a multidimensional function `f(x)` by varying vector `x`. This routine does *not* use user-supplied information about the gradient of `f(x)`. The following attributes are available:

x
Location of the most recently computed minimum (1-d array).

f
Value of function `f(x)` at the most recently computed minimum.

nit
Number of iterations required to find most recent minimum.

error
None if fit successful; an error message otherwise.

multiminex is a wrapper for the multimin *GSL* routine.

5.6 Requirements

lsqfit relies heavily on the gvar, and numpy modules. Several utility functions are in lsqfit_util. Also the minimization routines are from the Gnu Scientific Library (*GSL*).

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

l

Symbols

A

B

C

D

E

F

G

J

L

M

N

P

Q

S

T

W

X

Y