lsqfit Documentation

Release 4.5

G. P. Lepage

July 31, 2013

Contents:

CHAPTER

ONE

OVERVIEW AND TUTORIAL

1.1 Introduction

The modules defined here are designed to facilitate least-squares fitting of noisy data by multi-dimensional, nonlinear functions of arbitrarily many parameters. The central module is `lsqfit` because it provides the fitting functions. `lsqfit` makes heavy use of auxiliary module `gvar`, which provides tools that facilitate the analysis of error propagation, and also the creation of complicated multi-dimensional Gaussian distributions.

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 = dict(a=gv.gvar(0.5, 0.5), 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)
print(fit.format(100))      # 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

# save best-fit values in file 'outputfile.p' for later use
```

```
import pickle
pickle.dump(fit.p, open('outputfile.p', 'wb'))
```

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 +- 0.0685565 2.01 +- 0.236643]
>>> 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`. 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    itns = 5

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 = (1e-15,1e-15)    svdnum = (None,None)    reltol/abstol = 0.0001/0

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.

The last section of the code uses Python's pickle module to save the best-fit values of the parameters in a file for later use. They are recovered using pickle again:

```
>>> import pickle
>>> p = pickle.load(open('outputfile.p', 'rb'))
>>> print(p['a'])
0.252798 +- 0.0323152
>>> print(p['b'])
0.448762 +- 0.0647224
>>> print(p['b']/p['a'])
1.77518 +- 0.298185
```

The recovered parameters are gvar.GVars, with their full covariance matrix intact. (pickle works here because the variables in fit.p are stored in a special dictionary of type gvar.BufferDict; gvar.GVars cannot be pickled otherwise.)

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 = dict(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.
- The input data is stored in a dictionary (y) whose values can be gvar.GVars or arrays of gvar.GVars. 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` does allow `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`).

What follows is a brief 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. At the very end, in an appendix, there is a very simple pedagogical example that illustrates the nature of priors and demonstrates some of the simpler techniques supported by `lsqfit`.

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 Making Fake Data

We need data in order to demonstrate curve fitting. The easiest route is to make fake data. The recipe is simple: 1) choose some well defined function `f(x)` of the independent variable `x`; 2) choose values for the `x`s, and therefore the "correct" values for `y=f(x)`; and 3) add random noise to the `y`s, to simulate measurement errors. Here we will work through a simple implementation of this recipe to illustrate how the `gvar` module can be used to build complicated Gaussian distributions (in this case for the correlated noise in the `y`s). A reader eager to fit real data can skip this section on first reading.

For the function `f` we choose something familiar (to some people): a sum of exponentials `sum_i=0..99 a[i] exp(-E[i]*x)`. We take as our exact values for the parameters `a[i]=0.4` and `E[i]=0.9*(i+1)`, which are easy to remember. This is simple in Python:

```
import numpy as np

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

For `x`s we take `1,2,3..10,12,14..20`, and exact `y`s are then given by `f_exact(x)`:

```
>>> x = array([1.,2.,3.,4.,5.,6.,7.,8.,9.,10.,12.,14.,16.,18.,20.])
>>> y_exact = f_exact(x)
>>> print(y_exact)                # correct/exact values for y
[  2.74047100e-01   7.92134506e-02   2.88190008e-02 ... ]
```

Finally we need to add random noise to the `y_exacts` to obtain our fit data. We do this by forming `y_exact*noise` where

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

Here `x_max` is the largest `x` used, and the `c[n]` are Gaussian random variable with means and standard deviations of order 0.01. This is easy to implement in Python using the `gvar` module:

```
import gvar as gv

def make_data(nexp=100, eps=0.01): # make x, y fit data
    x = np.array([1.,2.,3.,4.,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
```

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 +- 0.01
>>> 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)` and have statistical errors:

```
>>> print(y)
[0.275179 +- 0.0027439 0.0795054 +- 0.000796125 ... ]
>>> print(y-f_exact(x))
[0.00113215 +- 0.0027439 0.000291951 +- 0.000796125 ... ]
```

Different `y`s are also correlated (by construction), which becomes clear if we evaluate the covariance matrix for the `y`s:

```
>>> print(gv.evalcov(y))
[[  7.52900382e-06   2.18173029e-06   7.95744444e-07 ... ]
 [  2.18173029e-06   6.33815228e-07   2.31761675e-07 ... ]
 [  7.95744444e-07   2.31761675e-07   8.49651978e-08 ... ]
 ...
]
```

The diagonal elements of the covariance matrix are the variances of the individual `y`s; the off-diagonal elements are a measure of the correlations:

```
< (y[i]-<y[i]>) * (y[j]-<y[j]>) >.
```

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

1.3 Basic Fits

Now that we have fit data, `x,y = make_data()`, we pretend ignorance of the exact functional relationship between `x` and `y` (*i.e.*, `y=f_exact(x)`). Typically we *do* know the functional form and have some *a priori* idea about the

parameter values. The point of the fit is to improve our knowledge of the parameter values, beyond our *a priori* impressions, by analyzing the fit data. Here we see how to do this using the `lsqfit` module.

First we need code to represent the fit function. In this case we know that a sum of exponentials is appropriate, so 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 `b` 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.

Next we need to define priors that encapsulate our *a priori* knowledge about the 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 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 ± 0.5, while `E[i]` is of order `1+i±0.5`. 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, 0.5) for i in range(nexp)]
    prior['E'] = [gv.gvar(i+1, 0.5) 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'])
[0.5 +- 0.5 0.5 +- 0.5 0.5 +- 0.5]
>>> print(prior['E'])
[1 +- 0.5 2 +- 0.5 3 +- 0.5]
```

We use dictionary-like class `gvar.BufferDict` for the prior because it allows us to save the prior 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, from this section and the previous one, 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([1.,2.,3.,4.,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, 0.5) for i in range(nexp)]
    prior['E'] = [gv.gvar(i+1, 0.5) 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(3, 20):
        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
        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()
        if fit.chi2 / fit.dof < 1.:
            p0 = fit.pmean            # starting point for next fit (opt.)

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

We are not sure *a priori* how many exponentials are needed to fit our data; given that there are only fifteen `y`s, and these are noisy, there may only be information in the data about the first few terms. Consequently we wrote our code to try fitting with each of `nexp=3,4,5..19` 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 = 3
Least Square Fit:
  chi2/dof [dof] = 6.3e+02 [15]    Q = 0    logGBF = -4465.1    itns = 30

Parameters:
            a 0   0.0288 (11)      [  0.50 (50) ]
              1   0.0354 (13)      [  0.50 (50) ]
              2   0.0779 (30)      [  0.50 (50) ]
            E 0   1.0107 (24)      [  1.00 (50) ]
              1   2.0200 (27)      [  2.00 (50) ]
              2   3.6643 (33)      [  3.00 (50) ]  *
```

```
Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 1.99859 +- 0.00239496   E2/E0 = 3.62551 +- 0.00618821
a1/a0 = 1.2313 +- 0.000474021   a2/a0 = 2.707 +- 0.00130172

************************************ nexp = 4
Least Square Fit:
  chi2/dof [dof] = 0.57 [15]    Q = 0.9    logGBF = 220.04    itns = 220

Parameters:
            a 0    0.4018 (40)      [  0.50 (50) ]
              1    0.4055 (42)      [  0.50 (50) ]
              2    0.4952 (76)      [  0.50 (50) ]
              3     1.124 (12)      [  0.50 (50) ]  *
            E 0   0.90037 (51)      [  1.00 (50) ]
              1    1.8023 (13)      [  2.00 (50) ]
              2    2.7731 (90)      [  3.00 (50) ]
              3     4.383 (21)      [  4.00 (50) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 2.00179 +- 0.00120727   E2/E0 = 3.07996 +- 0.00977469
a1/a0 = 1.00942 +- 0.00299114   a2/a0 = 1.23251 +- 0.0140225

************************************ nexp = 5
Least Square Fit:
  chi2/dof [dof] = 0.45 [15]    Q = 0.97    logGBF = 220.84    itns = 6

Parameters:
            a 0    0.4018 (40)      [  0.50 (50) ]
              1    0.4049 (44)      [  0.50 (50) ]
              2     0.478 (26)      [  0.50 (50) ]
              3      0.63 (28)      [  0.50 (50) ]
              4      0.62 (35)      [  0.50 (50) ]
            E 0   0.90036 (51)      [  1.00 (50) ]
              1    1.8019 (15)      [  2.00 (50) ]
              2     2.759 (22)      [  3.00 (50) ]
              3      4.09 (26)      [  4.00 (50) ]
              4      4.95 (48)      [  5.00 (50) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 2.00133 +- 0.00142399   E2/E0 = 3.06486 +- 0.024111
a1/a0 = 1.00754 +- 0.00422523   a2/a0 = 1.18874 +- 0.0632288

************************************ nexp = 6
Least Square Fit:
  chi2/dof [dof] = 0.45 [15]    Q = 0.97    logGBF = 220.7    itns = 6

Parameters:
            a 0    0.4018 (40)      [  0.50 (50) ]
              1    0.4041 (47)      [  0.50 (50) ]
              2     0.461 (41)      [  0.50 (50) ]
              3      0.60 (24)      [  0.50 (50) ]
              4      0.47 (37)      [  0.50 (50) ]
```

```
              5      0.45 (46)       [  0.50 (50) ]
            E 0   0.90035 (51)       [  1.00 (50) ]
              1    1.8015 (17)       [  2.00 (50) ]
              2     2.746 (34)       [  3.00 (50) ]
              3      3.98 (32)       [  4.00 (50) ]
              4      4.96 (49)       [  5.00 (50) ]
              5      6.01 (50)       [  6.00 (50) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 2.00084 +- 0.00168855   E2/E0 = 3.04939 +- 0.0374116
a1/a0 = 1.00551 +- 0.00561391   a2/a0 = 1.14607 +- 0.101175

************************************ nexp = 7
Least Square Fit:
  chi2/dof [dof] = 0.45 [15]    Q = 0.96    logGBF = 220.6    itns = 6

Parameters:
            a 0    0.4018 (40)       [  0.50 (50) ]
              1    0.4036 (48)       [  0.50 (50) ]
              2     0.452 (47)       [  0.50 (50) ]
              3      0.60 (22)       [  0.50 (50) ]
              4      0.42 (37)       [  0.50 (50) ]
              5      0.42 (46)       [  0.50 (50) ]
              6      0.46 (49)       [  0.50 (50) ]
            E 0   0.90035 (51)       [  1.00 (50) ]
              1    1.8012 (18)       [  2.00 (50) ]
              2     2.739 (39)       [  3.00 (50) ]
              3      3.94 (33)       [  4.00 (50) ]
              4      4.96 (49)       [  5.00 (50) ]
              5      6.02 (50)       [  6.00 (50) ]
              6      7.02 (50)       [  7.00 (50) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 2.00059 +- 0.00181902   E2/E0 = 3.04178 +- 0.0431729
a1/a0 = 1.0045 +- 0.00626848   a2/a0 = 1.1258 +- 0.116336
                                      .
                                      .
                                      .

************************************ nexp = 19
Least Square Fit:
  chi2/dof [dof] = 0.46 [15]    Q = 0.96    logGBF = 220.52    itns = 1

Parameters:
            a 0    0.4018 (40)       [  0.50 (50) ]
              1    0.4033 (49)       [  0.50 (50) ]
              2     0.447 (51)       [  0.50 (50) ]
              3      0.60 (21)       [  0.50 (50) ]
              4      0.38 (37)       [  0.50 (50) ]
              5      0.40 (46)       [  0.50 (50) ]
              6      0.45 (49)       [  0.50 (50) ]
              7      0.48 (50)       [  0.50 (50) ]
              8      0.49 (50)       [  0.50 (50) ]
              9      0.50 (50)       [  0.50 (50) ]
```

```
              10      0.50 (50)       [  0.50 (50) ]
              11      0.50 (50)       [  0.50 (50) ]
              12      0.50 (50)       [  0.50 (50) ]
              13      0.50 (50)       [  0.50 (50) ]
              14      0.50 (50)       [  0.50 (50) ]
              15      0.50 (50)       [  0.50 (50) ]
              16      0.50 (50)       [  0.50 (50) ]
              17      0.50 (50)       [  0.50 (50) ]
              18      0.50 (50)       [  0.50 (50) ]
            E 0    0.90035 (51)       [  1.00 (50) ]
              1     1.8011 (19)       [  2.00 (50) ]
              2      2.734 (42)       [  3.00 (50) ]
              3       3.91 (33)       [  4.00 (50) ]
              4       4.97 (49)       [  5.00 (50) ]
              5       6.02 (50)       [  6.00 (50) ]
              6       7.02 (50)       [  7.00 (50) ]
              7       8.01 (50)       [  8.00 (50) ]
              8       9.00 (50)       [  9.00 (50) ]
              9      10.00 (50)       [ 10.00 (50) ]
             10      11.00 (50)       [ 11.00 (50) ]
             11      12.00 (50)       [ 12.00 (50) ]
             12      13.00 (50)       [ 13.00 (50) ]
             13      14.00 (50)       [ 14.00 (50) ]
             14      15.00 (50)       [ 15.00 (50) ]
             15      16.00 (50)       [ 16.00 (50) ]
             16      17.00 (50)       [ 17.00 (50) ]
             17      18.00 (50)       [ 18.00 (50) ]
             18      19.00 (50)       [ 19.00 (50) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 2.00041 +- 0.00190442   E2/E0 = 3.0363 +- 0.046784
a1/a0 = 1.00376 +- 0.00668978   a2/a0 = 1.11149 +- 0.125413

--------------------- fit with extra information
```

There are several things to notice here:

- Clearly three exponentials (`nexp=3`) is not enough. The `chi**2` per degree of freedom (`chi2/dof`) is much larger than one. The `chi**2` improves significantly for `nexp=4` exponentials and by `nexp=6` the fit is as good as it is going to get — 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 0.9003(5), respectively, from the fit where the exact answers are 0.4 and 0.9; and we get 0.45(5) and 2.73(4) for the third exponential where the exact values are 0.4 and 2.7.
- Note in the `nexp=7` fit how the means and standard deviations for the parameters governing the seventh (and last) exponential are almost identical to the values in the corresponding priors: 0.46(49) from the fit for `a` and 7.0(5) for `E`. This tells us that our fit data has little or 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 is truer still of further terms as they are added in the `nexp=8` and later fits. This is why the fit results stop changing once we have `nexp=6` 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=19` exponentials and therefore has 38 parameters. This is in a fit to 15 `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=6` 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=19` fit, for example, we actually have 53 pieces of data to fit: the 15 `ys` plus the 38 prior values for the 38 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 53-38=15 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=3`) but then more-or-less stops changing around `nexp=6`. The implication is that this data is much more likely to have come from a theory with `nexp>=6` than with `nexp=3` (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 obvious in the ratio `a1/a0`, which would be 1.004(16) if there was no statistical correlation between our estimates for `a1` and `a0`, but in fact is 1.004(7) in this fit. The (positive) correlation is evident in the covariance matrix:

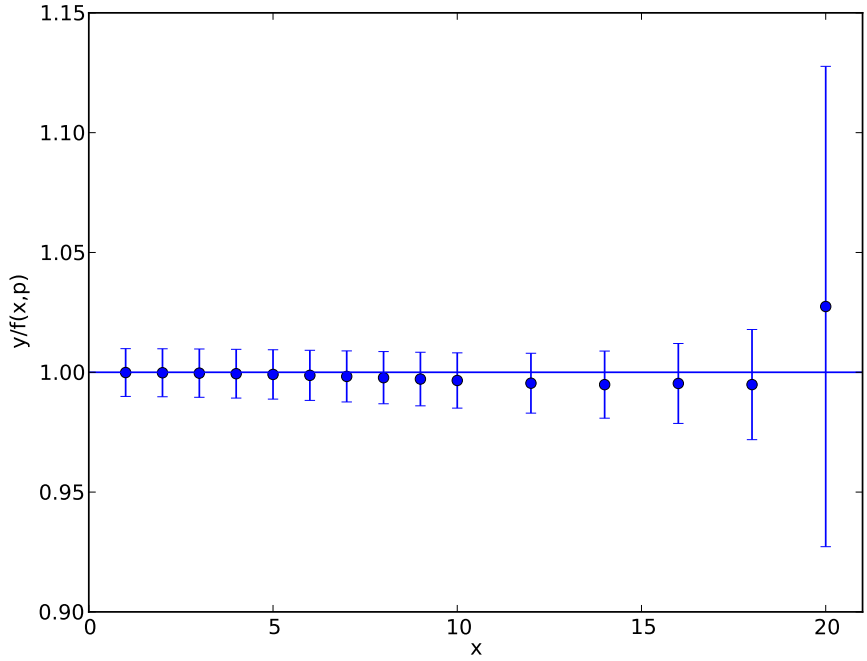
  ```
  >>> print(gv.evalcov([a[0], a[1]]))
  [[  1.61726195e-05   1.65492001e-05]
  [  1.65492001e-05   2.41547633e-05]]
  ```

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(100)`):

```
Fit:
     x[k]               y[k]           f(x[k],p)
-----------------------------------------------
        1         0.2752 (27)         0.2752 (20)
        2        0.07951 (80)        0.07952 (58)
        3        0.02891 (29)        0.02892 (21)
        4        0.01127 (11)       0.011272 (83)
        5       0.004502 (46)       0.004506 (34)
        6       0.001817 (19)       0.001819 (14)
        7      0.0007362 (79)      0.0007375 (57)
        8      0.0002987 (33)      0.0002994 (24)
        9      0.0001213 (14)     0.00012163 (99)
       10     0.00004926 (57)     0.00004943 (41)
       12        8.13(10)e-06        8.164(72)e-06
       14       1.342(19)e-06       1.348(13)e-06
       16       2.217(37)e-07       2.227(23)e-07
```

```
18       3.661(85)e-08       3.679(40)e-08
20        6.24(61)e-09       6.078(71)e-09
```

The fit is excellent over the entire eight 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 pylab as plt
ratio = y / f(x, fit.pmean)
plt.xlim(0, 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([0.0, 21.0], [1.0, 1.0])
plt.show()
```

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:

```
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)
```

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

```
Least Square Fit:
  chi2/dof [dof] = 0.32 [1]    Q = 0.57    logGBF = 3.9303    itns = 2

Parameters:
            a 0    0.4018 (40)      [  0.4018 (40) ]
              1    0.4018 (40)      [  0.4033 (49) ]
              2     0.421 (20)      [   0.447 (51) ]
              3      0.53 (17)      [    0.60 (21) ]
              4      0.46 (34)      [    0.38 (37) ]
              5      0.50 (42)      [    0.40 (46) ]
              6      0.50 (48)      [    0.45 (49) ]
              7      0.50 (50)      [    0.48 (50) ]
              8      0.50 (50)      [    0.49 (50) ]
              9      0.50 (50)      [    0.50 (50) ]
             10      0.50 (50)      [    0.50 (50) ]
             11      0.50 (50)      [    0.50 (50) ]
             12      0.50 (50)      [    0.50 (50) ]
             13      0.50 (50)      [    0.50 (50) ]
             14      0.50 (50)      [    0.50 (50) ]
             15      0.50 (50)      [    0.50 (50) ]
             16      0.50 (50)      [    0.50 (50) ]
             17      0.50 (50)      [    0.50 (50) ]
             18      0.50 (50)      [    0.50 (50) ]
            E 0   0.90030 (51)      [ 0.90035 (51) ]
              1   1.80007 (67)      [  1.8011 (19) ]
              2     2.711 (12)      [   2.734 (42) ]
              3      3.76 (18)      [    3.91 (33) ]
              4      5.02 (48)      [    4.97 (49) ]
              5      6.00 (50)      [    6.02 (50) ]
              6      7.00 (50)      [    7.02 (50) ]
              7      8.00 (50)      [    8.01 (50) ]
              8      9.00 (50)      [    9.00 (50) ]
              9     10.00 (50)      [   10.00 (50) ]
             10     11.00 (50)      [   11.00 (50) ]
             11     12.00 (50)      [   12.00 (50) ]
             12     13.00 (50)      [   13.00 (50) ]
             13     14.00 (50)      [   14.00 (50) ]
             14     15.00 (50)      [   15.00 (50) ]
             15     16.00 (50)      [   16.00 (50) ]
             16     17.00 (50)      [   17.00 (50) ]
             17     18.00 (50)      [   18.00 (50) ]
             18     19.00 (50)      [   19.00 (50) ]

Settings:
```

```
  svdcut = (1e-15,1e-15)    svdnum = (None,None)     reltol/abstol = 0.0001/0
```

Parameters `a[0]` and `E[0]` are essentially unchanged by the new information, but `a[i]` and `E[i]` are much more precise for `i=2` and `i=3`. 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 up `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 the `chi**2` function for the original fit can be well approximated by a quadratic function in the fit parameters — that is, insofar as `exp(-chi**2/2)` is well approximated by a Gaussian distribution in the parameters, as specified by the best-fit means and covariance matrix (in `fit.p`). 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.

1.5 `x` has Error Bars

We now consider variations on our basic fit analysis (described above). 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 ±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([1.,2.,3.,4.,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(3, 20):
        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
        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()
        if fit.chi2/fit.dof<1.:
            p0 = fit.pmean          # starting point for next fit (opt.)
```

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=6` terms:

```
************************************ nexp = 6
Least Square Fit:
  chi2/dof [dof] = 0.54 [15]    Q = 0.92    logGBF = 198.93    itns = 6

Parameters:
            a 0      0.4025 (41)        [     0.50 (50) ]
              1       0.429 (32)        [     0.50 (50) ]
              2        0.58 (23)        [     0.50 (50) ]
              3        0.40 (38)        [     0.50 (50) ]
              4        0.42 (46)        [     0.50 (50) ]
              5        0.46 (49)        [     0.50 (50) ]
            E 0     0.90068 (60)        [     1.00 (50) ]
              1       1.818 (20)        [     2.00 (50) ]
              2        2.95 (28)        [     3.00 (50) ]
              3        3.98 (49)        [     4.00 (50) ]
              4        5.02 (50)        [     5.00 (50) ]
              5        6.01 (50)        [     6.00 (50) ]
            x 0    0.999997 (10)        [ 0.999997 (10) ]
              1    1.999958 (20)        [ 1.999958 (20) ]
              2    3.000014 (30)        [ 3.000013 (30) ]
              3    4.000065 (36)        [ 4.000064 (40) ]
              4    5.000047 (34)        [ 5.000069 (50) ]
              5    6.000020 (39)        [ 5.999986 (60) ]
              6    6.999988 (40)        [ 6.999942 (70) ]
              7    7.999956 (42)        [ 7.999982 (80) ]
              8    8.999934 (50)        [ 9.000054 (90) ]  *
              9    9.999923 (59)        [  9.99991 (10) ]
             10   11.999929 (79)        [ 11.99982 (12) ]
             11    13.99992 (11)        [ 13.99991 (14) ]
             12    15.99992 (15)        [ 15.99998 (16) ]
             13    18.00022 (18)        [ 18.00020 (18) ]
             14    20.00016 (20)        [ 20.00016 (20) ]

Settings:
  svdcut = (1e-15,1e-15)    svdnum = (None,None)    reltol/abstol = 0.0001/0
```

```
E1/E0 = 2.01801 +- 0.0216404    E2/E0 = 3.27385 +- 0.305544
a1/a0 = 1.06515 +- 0.0765488    a2/a0 = 1.4485 +- 0.573896
```

This looks quite a bit like what we obtained before, except that now there are 15 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 the errors on the `x` values have been reduced — by information in the fit data.

1.6 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 ± 0.5 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 +- 0.5 1.9 +- 0.5001 2.8 +- 0.5002]
>>> print(E[1] - E[0], E[2] - E[1])
0.9 +- 0.01 0.9 +- 0.01
```

which shows that each `E[i]` separately has an uncertainty of ± 0.5 (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, 0.5) for i in range(nexp)]
    de = [gv.gvar(0.9, 0.01) for i in range(nexp)]
    de[0] = gv.gvar(1, 0.5)
    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=7` terms:

```
************************************ nexp = 7
Least Square Fit:
  chi2/dof [dof] = 0.44 [15]    Q = 0.97    logGBF = 227.47    itns = 3

Parameters:
            a 0    0.4018 (40)       [  0.50 (50) ]
              1    0.4016 (42)       [  0.50 (50) ]
              2     0.404 (12)       [  0.50 (50) ]
              3     0.394 (46)       [  0.50 (50) ]
```

```
              4      0.40 (16)       [  0.50 (50) ]
              5      0.51 (31)       [  0.50 (50) ]
              6      0.52 (42)       [  0.50 (50) ]
          E 0   0.90032 (51)       [  1.00 (50) ]
              1    1.8001 (11)       [  1.90 (50) ]
              2     2.701 (10)       [  2.80 (50) ]
              3     3.601 (14)       [  3.70 (50) ]
              4     4.501 (17)       [  4.60 (50) ]
              5     5.401 (20)       [  5.50 (50) ]
              6     6.301 (22)       [  6.40 (50) ]

Settings:
  svdcut = (1e-15,None)   svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 1.9994 +- 0.00106276   E2/E0 = 2.99989 +- 0.0110965
a1/a0 = 0.999601 +- 0.00254531   a2/a0 = 1.00498 +- 0.0280721
```

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] = 0.900(1)`, which is ten 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 significantly larger with the correlated prior (`logGBF = 227`) than it was for the uncorrelated prior (`logGBF = 221`). 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(227-221) = 400` times more likely to get this 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.

1.7 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=7`:

```
Least Square Fit:
  chi2/dof [dof] = 0.77 [15]    Q = 0.71    logGBF = 233.98    itns = 1

Parameters:
            a 0    0.4026 (40)      [ 0.500 (95) ]  *
              1    0.4025 (41)      [ 0.500 (95) ]  *
              2    0.4071 (80)      [ 0.500 (95) ]
              3     0.385 (20)      [ 0.500 (95) ]  *
              4     0.431 (58)      [ 0.500 (95) ]
              5     0.477 (74)      [ 0.500 (95) ]
              6     0.493 (89)      [ 0.500 (95) ]
            E 0   0.90031 (50)      [  1.00 (50) ]
              1    1.8000 (10)      [  1.90 (50) ]
              2    2.7023 (86)      [  2.80 (50) ]
              3     3.603 (14)      [  3.70 (50) ]
              4     4.503 (17)      [  4.60 (50) ]
              5     5.403 (19)      [  5.50 (50) ]
              6     6.303 (22)      [  6.40 (50) ]

Settings:
  svdcut = (1e-15,1e-15)    svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 1.99934 +- 0.0010096   E2/E0 = 3.0015 +- 0.00944816
a1/a0 = 0.999537 +- 0.00248687   a2/a0 = 1.01093 +- 0.0168333
prior['a'] = 0.5 +- 0.0952405
```

Reducing the width of the `prior['a']`s from 0.5 to 0.1 increased `logGBF` from 227 to 234. The error for `a2/a0` is 40% smaller, but the other results are not much 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] = 0.47 [15]    Q = 0.96    logGBF = 254.15    itns = 6

Parameters:
            a 0    0.4018 (40)      [  0.50 (50) ]
              1    0.4018 (40)      [  0.50 (50) ]
            E 0   0.90036 (50)      [  1.00 (50) ]
              1   1.80036 (50)      [  1.90 (50) ]

Settings:
  svdcut = (1e-15,1e-15)    svdnum = (None,None)    reltol/abstol = 0.0001/0


************************************ nexp = 3
Least Square Fit:
  chi2/dof [dof] = 0.5 [15]    Q = 0.94    logGBF = 243.12    itns = 4

Parameters:
            a 0    0.4018 (40)      [  0.50 (50) ]
              1    0.4018 (40)      [  0.50 (50) ]
              2      8(10)e-06      [  0.50 (50) ]
            E 0   0.90035 (50)      [  1.00 (50) ]
              1   1.80034 (50)      [  1.90 (50) ]
              2     2.700 (10)      [  2.80 (50) ]

Settings:
  svdcut = (1e-15,1e-15)    svdnum = (None,None)    reltol/abstol = 0.0001/0
```

Measured by their `chi**2`s, the two fits are almost equally good. The Bayes factor for the first fit, however, is much 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(254 - 243) = 59,874` 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.

1.8 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])
1.9994 +- 0.00106276
>>> print((E[1] / E[0]).partialsdev(fit.prior['E']))
0.000419219538523
>>> print((E[1] / E[0]).partialsdev(fit.prior['a']))
0.000158871440271
```

```
>>> print((E[1] / E[0]).partialsdev(y))
0.000952553004005
```

This shows that the total uncertainty in `E[1]/E[0]`, 0.00106, is the sum in quadrature of a contribution 0.00042 due to the priors specified by `prior['E']`, 0.00016 due to `prior['a']`, and 0.00095 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: 3.000(11)
               E1/E0: 1.9994(11)
               a2/a0: 1.005(28)
               a1/a0: 0.9996(25)

Partial % Errors:
                          E2/E0     E1/E0     a2/a0     a1/a0
-------------------------------------------------------------
                   a:      0.09      0.01      1.09      0.02
                   y:      0.07      0.05      0.77      0.19
                   E:      0.35      0.02      2.44      0.16
-------------------------------------------------------------
               total:      0.37      0.05      2.79      0.25
```

This table shows, for example, that the 0.37% uncertainty in `E2/E0` comes from a 0.09% contribution due to `prior['a']`, a 0.07% contribution due to due to statistical errors in the fit data `y`, and a 0.35% 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 0.02% (from 0.05%) while leaving the `E2/E0` error at 0.37%.

1.9 `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 in principle 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(10))           # 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 (input data correlated with prior):
  chi2/dof [dof] = 0.051 [10]    Q = 1    logGBF = 97.499    itns = 5

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 = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0


************************************ nexp = 2
Least Square Fit (input data correlated with prior):
  chi2/dof [dof] = 0.053 [10]    Q = 1    logGBF = 99.041    itns = 3

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 = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0


************************************ nexp = 3
Least Square Fit (input data correlated with prior):
  chi2/dof [dof] = 0.057 [10]    Q = 1    logGBF = 99.845    itns = 4

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 (76)
      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 (52)
      2.6    0.042602 (67)    0.042601 (26)
      2.8    0.034983 (33)    0.034982 (13)

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0


********************************** nexp = 4
Least Square Fit (input data correlated with prior):
  chi2/dof [dof] = 0.057 [10]    Q = 1    logGBF = 99.841    itns = 4

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 (78)
        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 = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 0.0001/0


E1/E0 = 1.999(24)   E2/E0 = 3.00(21)
a1/a0 = 0.997(77)   a2/a0 = 1.01(25)
```

Here we use `fit.format(10)` 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 `y`s having no errors, it is straightforward to apply the same ideas to a situation where the `y`s 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).

1.10 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 (input data correlated with prior):
  chi2/dof [dof] = 0.057 [10]    Q = 1    logGBF = 100.46    itns = 5

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)        0.267 (16)
      1.2       0.2027 (32)       0.2030 (74)
      1.4       0.1573 (13)       0.1574 (34)
      1.6      0.12378 (54)       0.1238 (15)
      1.8      0.09853 (22)      0.09854 (67)
        2     0.079153 (93)      0.07915 (29)
      2.2     0.064051 (39)      0.06405 (12)
      2.4     0.052134 (16)     0.052134 (41)
```

```
      2.6    0.0426348 (67)    0.0426347 (92)
      2.8    0.0349985 (28)    0.0349985 (40)

Settings:
  svdcut = (1e-19,None)    svdnum = (None,None)    reltol/abstol = 0.0001/0


E1/E0 = 2.00(72)   E2/E0 = 3.0(6.1)
a1/a0 = 1.0(2.3)   a2/a0 = 1.0(6.8)
```

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` 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.

The standard 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)
```

Then the inverse of the `y` covariance matrix is computed from its eigenvalues and eigenvectors, but with any eigenvalue smaller than `svdcut` times the largest eigenvalue replaced by the cutoff (that is, by `svdcut` times the largest eigenvalue). This limits the singularity of the covariance matrix, leading to improved numerical stability. The cost is less precision in the final results since we are in effect decreasing the precision of the input `y` data. This is a conservative move, but numerical stability is worth the tradeoff.

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 error budget is interesting in this case. There is no contribution from the original `y` data since it was 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 covariance matrix. The *SVD* contribution 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(24)
               a2/a0: 1.01(25)
               a1/a0: 0.997(77)
```

```
Partial % Errors:
                         E2/E0     E1/E0     a2/a0     a1/a0
-------------------------------------------------------------
                  a:      3.76      0.71     11.80      4.39
                svd:      0.29      0.10      0.13      0.55
                  E:      5.87      0.99     22.35      6.30
-------------------------------------------------------------
              total:      6.98      1.22     25.27      7.70
```

Here the contribution from the *SVD* cut is almost negligible.

Note that covariance matrices are rescaled so that all diagonal elements equal one before the *SVD* cut is applied. This means, among other things, that uncorrelated errors — that is, diagonal sub-matrices of the covariance matrix — are unaffected by *SVD* cuts. 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]]
```

`lsqfit.nonlinear_fit` uses a default value for `svdcut` of 1e-15, and applies *SVD* cuts to the covariance matrices from both the fit data and the prior. 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 positive correlations between different parts of the fit data or prior. Then much larger `svdcut`s may be needed.

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 argument list of `lsqfit.nonlinear_fit`.

1.11 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(17)   E2/E1 = 2.98(18)
a1/a0 = 0.995(55)   a2/a0 = 0.97(28)
E1 = 1.799(16)   a1 = 0.398(23)
```

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`.

1.12 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(3, 20):
        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
        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()
        if fit.chi2 / fit.dof < 1.:
            p0 = fit.pmean          # starting point for next fit (opt.)

    # 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 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] = 0.27 [15]    Q = 1    logGBF = 228.78    itns = 46

Parameters:
            a 0    0.3968 (40)       [  0.50 (50) ]
              1    0.3983 (41)       [  0.50 (50) ]
              2     0.390 (12)       [  0.50 (50) ]
              3     0.453 (45)       [  0.50 (50) ]
              4      0.15 (15)       [  0.50 (50) ]
              5      0.72 (31)       [  0.50 (50) ]
              6      0.71 (42)       [  0.50 (50) ]
            E 0   0.89969 (51)       [  1.00 (50) ]
              1    1.8011 (11)       [  1.90 (50) ]
              2     2.703 (10)       [  2.80 (50) ]
              3     3.603 (14)       [  3.70 (50) ]
              4     4.503 (17)       [  4.60 (50) ]
              5     5.403 (20)       [  5.50 (50) ]
              6     6.303 (22)       [  6.40 (50) ]

Settings:
  svdcut = (None,None)    svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 2.00188 +- 0.00106514   E2/E0 = 3.00474 +- 0.0111774
a1/a0 = 1.00397 +- 0.00252259   a2/a0 = 0.984026 +- 0.0282035
```

```
Simulated Fit Values - Exact Values:
E1/E0: 0.00247635 +- 0.00106514   E2/E0: 0.00485 +- 0.0111774
a1/a0: 0.00436695 +- 0.00252259   a2/a0: -0.0209515 +- 0.0282035

Parameter chi2/dof [dof] = 1.78 [4]   Q = 0.1

************************************ simulation
Least Square Fit:
  chi2/dof [dof] = 0.73 [15]    Q = 0.76    logGBF = 225.31    itns = 25

Parameters:
            a 0    0.4034 (40)      [  0.50 (50) ]
              1    0.4037 (42)      [  0.50 (50) ]
              2     0.403 (12)      [  0.50 (50) ]
              3     0.412 (46)      [  0.50 (50) ]
              4      0.34 (16)      [  0.50 (50) ]
              5      0.56 (31)      [  0.50 (50) ]
              6      0.57 (42)      [  0.50 (50) ]
            E 0   0.90024 (51)      [  1.00 (50) ]
              1    1.8005 (11)      [  1.90 (50) ]
              2     2.701 (10)      [  2.80 (50) ]
              3     3.601 (14)      [  3.70 (50) ]
              4     4.501 (17)      [  4.60 (50) ]
              5     5.401 (20)      [  5.50 (50) ]
              6     6.301 (22)      [  6.40 (50) ]

Settings:
  svdcut = (None,None)   svdnum = (None,None)    reltol/abstol = 0.0001/0

E1/E0 = 1.99998 +- 0.00105728   E2/E0 = 3.00061 +- 0.0111101
a1/a0 = 1.00066 +- 0.00253149   a2/a0 = 0.999036 +- 0.0280364

Simulated Fit Values - Exact Values:
E1/E0: 0.000580316 +- 0.00105728   E2/E0: 0.000719649 +- 0.0111101
a1/a0: 0.00106294 +- 0.00253149   a2/a0: -0.0059414 +- 0.0280364

Parameter chi2/dof [dof] = 0.13 [4]   Q = 1.0

************************************ simulation
Least Square Fit:
  chi2/dof [dof] = 0.92 [15]    Q = 0.54    logGBF = 223.86    itns = 2

Parameters:
            a 0    0.4003 (40)      [  0.50 (50) ]
              1    0.4002 (42)      [  0.50 (50) ]
              2     0.402 (12)      [  0.50 (50) ]
              3     0.390 (46)      [  0.50 (50) ]
              4      0.41 (16)      [  0.50 (50) ]
              5      0.49 (31)      [  0.50 (50) ]
              6      0.50 (42)      [  0.50 (50) ]
            E 0   0.90040 (51)      [  1.00 (50) ]
              1    1.8003 (11)      [  1.90 (50) ]
              2     2.701 (10)      [  2.80 (50) ]
              3     3.601 (14)      [  3.70 (50) ]
              4     4.501 (17)      [  4.60 (50) ]
              5     5.401 (20)      [  5.50 (50) ]
              6     6.301 (22)      [  6.40 (50) ]
```

```
Settings:
  svdcut = (None,None)    svdnum = (None,None)     reltol/abstol = 0.0001/0

E1/E0 = 1.99949 +- 0.0010671   E2/E0 = 2.99932 +- 0.0110989
a1/a0 = 0.999605 +- 0.00255698   a2/a0 = 1.0041 +- 0.0280514

Simulated Fit Values - Exact Values:
E1/E0: 9.04515e-05 +- 0.0010671   E2/E0: -0.000569657 +- 0.0110989
a1/a0: 4.25028e-06 +- 0.00255698   a2/a0: -0.000873644 +- 0.0280514

Parameter chi2/dof [dof] = 0.14 [4]   Q = 1.0
```

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 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.

1.13 Positive Parameters

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 would be made on a parameter- by-parameter basis, when creating the priors, and would have no impact on the definition of the fit function itself.

`lsqfit` provides a decorator, `lsqfit.transform_p`, for fit functions that makes this possible. This decorator only works for fit functions that use dictionaries for their parameters. Given a prior `prior` for a fit, the decorator is used in the following way: for example,

```
@lsqfit.transform_p(prior.keys(), 0)
def fitfcn(p):
    ...
```

when the parameter argument is the first argument of the fit function, or

```
@lsqfit.transform_p(prior.keys(), 1)
def fitfcn(x, p):
    ...
```

when the parameter argument is the second argument of the fit function (see the `lsqfit.transform_p` documentation for more detail). Consider any parameter `p['XX']` used in `fitfcn`. The prior distribution for that parameter can now be turned into a log-normal distribution by replacing `prior['XX']` with `prior['logXX']` (or `prior['log(XX)']`) when defining the prior, thereby assigning a Gaussian distribution to `logXX` rather than to `XX`. Nothing need be changed in the fit function, other than adding the decorator. The decorator automatically detects parameters whose keys begin with `'log'` and adds new parameters to the parameter-dictionary for `fitfcn` that are exponentials of those parameters.

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(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    itns = 2

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

Settings:
  svdcut = (1e-15,1e-15)    svdnum = (None,None)    reltol/abstol = 0.0001/0

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(loga=gv.log(gv.gvar(0.02, 0.02))) # loga not a

@lsqfit.transform_p(prior.keys(), 0)
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.transformed_p['a'].fmt())     # exp(loga)
```

The fit parameter is now `log(a)` rather than `a` itself, but we are able to use the identical fit function. Here `fit.transformed_p` is the same as `fit.p` but augmented to include the exponentials of any log-normal variables — that is, `a` as well as `loga`. Rather than including all keys, the decorator can be written with a list containing just the variables to be transformed: here, `@lsqfit.transform_p(['loga'], 0)`.

The result from this fit is

```
Least Square Fit:
  chi2/dof [dof] = 0.85 [20]    Q = 0.65    logGBF = 5.252    itns = 12

Parameters:
            loga   -4.44 (97)      [ -3.9 (1.0) ]

Settings:
  svdcut = (1e-15,1e-15)    svdnum = (None,None)    reltol/abstol = 0.0001/0

a = 0.012(11)
```

which is more compelling. The "correct" value for `a` here is 0.015 (from the method used to generate the `ys`).

`lsqfit.transform_p()` also allows parameters to be replaced by their square roots as fit parameters — for example, define `prior['sqrt(a)']` (or `prior['sqrta']`) rather than `prior['a']` when creating the prior. This again guarantees positive parameters. The prior for `p['a']` is an exponential distribution if the mean of `p['sqrt(a)']` is zero. Using `prior['sqrt(a)']` in place of `prior['a']` in the example above leads to `a = 0.010(13)`, which is almost identical to the result obtained from the log-normal distribution.

1.14 Troubleshooting

`lsqfit.nonlinear_fit` error messages that come from inside the *gsl* routnines doing the fits 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). Setting `debug=True` in the argument list of `lsqfit.nonlinear_fit` might result in more intelligible error messages. This option also causes the fitter to check for significant roundoff errors in the matrix inversions of the covariance matrices.

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-14,1e-14)` 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 +- 1 1 +- 1
>>> 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
```