lsqfit Documentation

Release 9.1.4

G. P. Lepage

Feb 16, 2018

CONTENTS

Contents:

CHAPTER

ONE

OVERVIEW AND TUTORIAL

1.1 Introduction

The *lsqfit* module is designed to facilitate least-squares fitting of noisy data by multi-dimensional, nonlinear functions of arbitrarily many parameters, each with a (Bayesian) prior. *lsqfit* makes heavy use of another module, gvar (distributed separately), which provides tools that simplify the analysis of error propagation, and also the creation of complicated multi-dimensional Gaussian distributions. This technology also allows *lsqfit* to calculate exact derivatives of fit functions from the fit functions themselves, using automatic differentiation, thereby avoiding the need to code these by hand (the fitters use the derivatives). The power of the gvar module, particularly for correlated distributions, enables a variety of unusual fitting strategies, as we illustrate below; it is a feature that distinguishes *lsqfit* from standard fitting packages.

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

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

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

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

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

p = fit.p                               # best-fit values for parameters
outputs = dict(a=p['a'], b=p['b'])
```

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

This code fits the function `f(x,a,b)= exp(a+b*x)` (see `fcn(x,p)`) to two sets of data, labeled `data1` and `data2`, by varying parameters `a` and `b` until `f(x['data1'],a,b)` and `f(x['data2'],a,b)` equal `y['data1']` and `y['data2']`, respectively, to within the `y`s' errors.

The means and covariance matrices for the `y`s are specified in the `gv.gvar(...)`s used to create them: thus, for example,

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

shows the means, standard deviations and covariance matrix for the data in the first data set (0.0685565 is the square root of the 0.0047 in the covariance matrix).

The dictionary `prior` gives *a priori* estimates for the two parameters, `a` and `b`: each is assumed to be 0.5±0.5 before fitting. The parameters `p[k]` in the fit function `fcn(x, p)` are stored in a dictionary having the same keys and layout as `prior` (since `prior` specifies the fit parameters for the fitter).

In addition to the `data1` and `data2` data sets, there is an extra piece of input data, `y['b/a']`, which indicates that `b/a` is 2±0.5. The fit function for this data is simply the ratio `b/a` (represented by `p['b']/p['a']` in fit function `fcn(x,p)`). The fit function returns a dictionary having the same keys and layout as the input data `y`.

The output from the code sample above is:

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

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

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

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 8/0.0)

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

Partial % Errors:
                    a       b/a         b
```

```
-------------------------------------
        y:     12.75     16.72     14.30
    prior:      0.92      1.58      1.88
-------------------------------------
    total:     12.78     16.80     14.42
```

The best-fit values for `a` and `b` are 0.253(32) and 0.449(65), respectively; and the best-fit result for `b/a` is 1.78(30), which, because of correlations, is slightly more accurate than might be expected from the separate errors for `a` and `b`. The error budget for each of these three quantities is tabulated at the end and shows that the bulk of the error in each case comes from uncertainties in the `y` data, with only small contributions from uncertainties in the priors `prior`. The fit results corresponding to each piece of input data are also tabulated (`Fit:  ...`); the agreement is excellent, as expected given that the `chi**2` per degree of freedom is only 0.17.

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

```
... as before ...

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

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

... as before ...
```

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

There are several things worth noting from this example:

- The input data (`y`) is expressed in terms of Gaussian random variables — quantities with means and a covariance matrix. These are represented by objects of type `gvar.GVar` in the code; module `gvar` has a variety of tools for creating and manipulating Gaussian random variables (also see below).
- The input data is stored in a dictionary (`y`) whose values can be `gvar.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* allows `y` to be an array instead of a dictionary, which might be preferable for 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.
- The best-fit values of the fit parameters (`fit.p[k]`) are also `gvar.GVar`s and these capture statistical correlations between different parameters that are indicated by the fit. These output parameters can be combined in arithmetic expressions, using standard operators and standard functions, to obtain derived quantities. These operations take account of and track statistical correlations.
- Function `gvar.fmt_errorbudget()` is a useful tool for assessing the origins (`inputs`) of the statistical errors obtained in various final results (`outputs`). It is particularly useful for analyzing the impact of the *a priori* uncertainties encoded in the prior (`prior`).
- Parameter `debug=True` is set in *`lsqfit.nonlinear_fit`*. This is a good idea, particularly in the early stages of a project, because it causes the code to check for various common errors and give more intelligible error messages than would otherwise arise. This parameter can be dropped once code development is over.
- The priors for the fit parameters specify Gaussian distributions, characterized by the means and standard deviations given `gv.gvar(...)`. Some other distributions become available if argument `extend=True` is included in the call to *`lsqfit.nonlinear_fit`*. The distribution for parameter `a`, for example, can then be switched to a log-normal distribution by replacing `prior['a']=gv.gvar(0.5, 0.5)` with:

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

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

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

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

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

```
from __future__ import print_function
```

at the start of your file.

1.2 Gaussian Random Variables and Error Propagation

The inputs and outputs of a nonlinear least squares analysis are probability distributions, and these distributions will be Gaussian provided the input data are sufficiently accurate. *`lsqfit`* assumes this to be the case. (It also provides tests for non-Gaussian behavior, together with methods for dealing with such behavior. See: *Non-Gaussian Behavior; Testing Fits*.)

One of the most distinctive features of *`lsqfit`* is that it is built around a class, `gvar.GVar`, of objects that can be used to represent arbitrarily complicated Gaussian distributions — that is, they represent *Gaussian random variables* that specify the means and covariance matrix of the probability distributions. The input data for a fit are represented by a collection of `gvar.GVar`s that specify both the values and possible errors in the input values. The result of a fit is a collection of `gvar.GVar`s specifying the best-fit values for the fit parameters and the estimated uncertainties in those values.

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

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

```
>>> import gvar as gv
>>> print(gv.gvar(1.0, 0.1), gv.gvar('1.0 +- 0.2'), gv.gvar('1.0(4)'))
1.00(10) 1.00(20) 1.00(40)

>>> print(gv.gvar([1.0, 1.0, 1.0], [0.1, 0.2, 0.41]))
[1.00(10) 1.00(20) 1.00(41)]

>>> print(gv.gvar(['1.0(1)', '1.0(2)', '1.00(41)']))
[1.00(10) 1.00(20) 1.00(41)]

>>> print(gv.gvar(dict(a='1.0(1)', b=['1.0(2)', '1.0(4)'])))
{'a': 1.00(10),'b': array([1.00(20), 1.00(40)], dtype=object)}
```

   `gvar` uses the compact notation 1.234(22) to represent 1.234±0.022 — the digits in parentheses indicate the uncertainty in the rightmost corresponding digits quoted for the mean value. Very large (or small) numbers use a notation like 1.234(22)e+10.

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

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

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

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

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

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

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

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

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

```
>>> import gvar as gv
>>> a = gv.gvar(1, 0.1)
>>> b = a + gv.gvar(0, 0.001)
```

```
>>> print(a, b)
1.00(10) 1.00(10)

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

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

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

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

>>> print gv.evalcov([x, y])
[[ 0.01        0.01388174]
 [ 0.01388174  0.01927153]]
```

The `gvar` module defines versions of the standard Python functions (`sin`, `cos`, ...) that work with `gvar.GVars`. Most any numeric pure-Python function will work with them as well. Numeric functions that are compiled in C or other low-level languages generally do not work with `gvar.GVars`; they should be replaced by equivalent pure-Python functions if they are needed for `gvar.GVar`-valued arguments. See the `gvar` documentation for more information.

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

4. The uncertainties in derived `gvar.GVars` come from the uncertainties in the primary `gvar.GVars` from which they were created. It is easy to create an "error budget" that decomposes the uncertainty in a derived `gvar.GVar` into components coming from each of the primary `gvar.GVars` involved in its creation. For example,

```
>>> import gvar as gv
>>> a = gv.gvar('1.0(1)')
>>> b = gv.gvar('0.9(2)')
>>> x = gv.log(1 + a ** 2)
>>> y = b * gv.cosh(a / 2)
>>> outputs = dict(x=x, y=y)
>>> print(gv.fmt_values(outputs))
Values:
                   y: 1.01(23)
                   x: 0.69(10)

>>> inputs = dict(a=a, b=b)
>>> print(gv.fmt_errorbudget(outputs=outputs, inputs=inputs))
Partial % Errors:
                    y         x
-------------------------------
```

```
        a:      2.31      14.43
        b:     22.22       0.00
-----------------------------
    total:     22.34      14.43
```

The error budget shows that most of `y`'s 22.34% uncertainty comes from `b`, with just 2.3% coming from `a`. The total uncertainty is the sum in quadrature of the two separate uncertainties. The uncertainty in `x` is entirely from `a`, of course.

5. Storing `gvar.GVar`s in a file for later use is somewhat complicated because one generally wants to hold onto their correlations as well as their mean values and standard deviations. One easy way to do this is to put all of the `gvar.GVar`s to be saved into a single array or dictionary that is saved using function `gvar.dump()`: for example, use

```
>>> gv.dump([a, b, x, y], 'outputfile.p')
```

to save the variables defined above in a file named `'outputfile.p'`. Loading the file into a Python code later, with `gvar.load()`, recovers the array with standard deviations and correlations intact:

```
>>> a,b,x,y = gv.load('outputfile.p')
>>> print(a, b, x, y)
1.00(10) 0.90(20) 0.69(10) 1.01(23)
>>> print(y / b, gv.sqrt(gv.exp(x) - 1) / a)
1.128(26) 1(0)
```

This recipe works with arrays of any shape, and also with dictionaries whose values are either `gvar.GVar`s or arrays of `gvar.GVar`s. In particular, the best-fit values for the fit parameters from a fit can be saved using something like `gv.dump(fit.p, 'fitparam.p')`.

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

1.3 Basic Fits

A fit analysis typically requires three types of input:

1. fit data `x, y` (or possibly just `y`);
2. a function `y = f(x,p)` relating values of `y` to to values of `x` and a set of fit parameters `p`; if there is no `x`, then `y = f(p)`;
3. some *a priori* idea about the fit parameters' values (possibly quite imprecise — for example, that a particular parameter is of order 1).

The point of the fit is to improve our knowledge of the parameter values, beyond our *a priori* impressions, by analyzing the fit data. We now show how to do this using the *`lsqfit`* module for a more realistic problem, one that is familiar from numerical simulations of quantum chromodynamics (QCD).

We need code for each of the three fit inputs. The fit data in our example is assembled by the following function:

```
import numpy as np
import gvar as gv

def make_data():
    x = np.array([  5.,   6.,   7.,   8.,   9.,  10.,  12.,  14.])
    ymean = np.array(
        [  4.5022829417e-03,   1.8170543788e-03,   7.3618847843e-04,
           2.9872730036e-04,   1.2128831367e-04,   4.9256559129e-05,
```

```
            8.1263644483e-06,   1.3415253536e-06]
        )
    ycov = np.array(
        [[ 2.1537808808e-09,   8.8161794696e-10,   3.6237356558e-10,
           1.4921344875e-10,   6.1492842463e-11,   2.5353714617e-11,
           4.3137593878e-12,   7.3465498888e-13],
        [  8.8161794696e-10,   3.6193461816e-10,   1.4921610813e-10,
           6.1633547703e-11,   2.5481570082e-11,   1.0540958082e-11,
           1.8059692534e-12,   3.0985581496e-13],
        [  3.6237356558e-10,   1.4921610813e-10,   6.1710468826e-11,
           2.5572230776e-11,   1.0608148954e-11,   4.4036448945e-12,
           7.6008881270e-13,   1.3146405310e-13],
        [  1.4921344875e-10,   6.1633547703e-11,   2.5572230776e-11,
           1.0632830128e-11,   4.4264622187e-12,   1.8443245513e-12,
           3.2087725578e-13,   5.5986403288e-14],
        [  6.1492842463e-11,   2.5481570082e-11,   1.0608148954e-11,
           4.4264622187e-12,   1.8496194125e-12,   7.7369196122e-13,
           1.3576009069e-13,   2.3914810594e-14],
        [  2.5353714617e-11,   1.0540958082e-11,   4.4036448945e-12,
           1.8443245513e-12,   7.7369196122e-13,   3.2498644263e-13,
           5.7551104112e-14,   1.0244738582e-14],
        [  4.3137593878e-12,   1.8059692534e-12,   7.6008881270e-13,
           3.2087725578e-13,   1.3576009069e-13,   5.7551104112e-14,
           1.0403917951e-14,   1.8976295583e-15],
        [  7.3465498888e-13,   3.0985581496e-13,   1.3146405310e-13,
           5.5986403288e-14,   2.3914810594e-14,   1.0244738582e-14,
           1.8976295583e-15,   3.5672355835e-16]]
        )
    return x, gv.gvar(ymean, ycov)
```

The function call `x,y = make_data()` returns eight `x[i]`, and the corresponding values `y[i]` that we will fit. The `y[i]` are `gvar.GVars` (Gaussian random variables — see previous section) built from the mean values in `ymean` and the covariance matrix `ycov`, which shows strong correlations:

```
>>> print(y)                        # fit data
[0.004502(46) 0.001817(19) 0.0007362(79) ... 1.342(19)e-06]

>>> print(gv.evalcorr(y))           # correlation matrix
[[ 1.          0.99853801  0.99397698  ... 0.83814041]
 [ 0.99853801  1.          0.99843828  ... 0.86234032]
 [ 0.99397698  0.99843828  1.          ... 0.88605708]
 ...
 ...
 ...
 [ 0.83814041  0.86234032  0.88605708  ... 1.        ]]
```

These particular data were generated numerically. They come from a function that is a sum of a very large number of decaying exponentials,

```
a[i] * np.exp(-E[i] * x)
```

with coefficients `a[i]` of order 0.5±0.4 and exponents `E[i]` of order i+1±0.4. The function was evaluated with a particular set of parameters `a[i]` and `E[i]`, and then noise was added to create this data. Our challenge is to find estimates for the values of the parameters `a[i]` and `E[i]` that were used to create the data.

Next we need code for the fit function. Here we know that a sum of decaying exponentials is appropriate, and therefore we define the following Python function:

```
import numpy as np

def fcn(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 as arrays in a dictionary, using labels `a` and `E` to access them. These parameters are varied in the fit to find the best-fit values `p=fitp` for which `fcn(x,fitp)` 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. In principle there are infinitely many exponentials; in practice, given the finite precision of our data, we will need only a few.

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

```
import lsqfit
import gvar as gv

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

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

```
>>> print(prior['a'])
[0.50(40) 0.50(40) 0.50(40)]

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

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

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

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

Our complete Python program is, therefore:

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

def main():
    x, y = make_data()                  # collect fit data
    p0 = None                           # make larger fits go faster (opt.)
    for nexp in range(1, 7):
```

```
        print('*********************************** nexp =', nexp)
        prior = make_prior(nexp)
        fit = lsqfit.nonlinear_fit(data=(x, y), fcn=fcn, prior=prior, p0=p0)
        print(fit)                  # print the fit results
        if nexp > 2:
            E = fit.p['E']          # best-fit parameters
            a = fit.p['a']
            print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
            print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
        if fit.chi2 / fit.dof < 1.:
            p0 = fit.pmean          # starting point for next fit (opt.)
        print()

    # error budget analysis
    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('================= Error Budget Analysis')
    print(fit.fmt_values(outputs))
    print(fit.fmt_errorbudget(outputs,inputs))

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

def make_data():                     # assemble fit data
   x = np.array([  5.,   6.,   7.,   8.,   9.,  10.,  12.,  14.])
   ymean = np.array(
       [  4.5022829417e-03,   1.8170543788e-03,   7.3618847843e-04,
          2.9872730036e-04,   1.2128831367e-04,   4.9256559129e-05,
          8.1263644483e-06,   1.3415253536e-06]
       )
   ycov = np.array(
       [[ 2.1537808808e-09,   8.8161794696e-10,   3.6237356558e-10,
          1.4921344875e-10,   6.1492842463e-11,   2.5353714617e-11,
          4.3137593878e-12,   7.3465498888e-13],
       [  8.8161794696e-10,   3.6193461816e-10,   1.4921610813e-10,
          6.1633547703e-11,   2.5481570082e-11,   1.0540958082e-11,
          1.8059692534e-12,   3.0985581496e-13],
       [  3.6237356558e-10,   1.4921610813e-10,   6.1710468826e-11,
          2.5572230776e-11,   1.0608148954e-11,   4.4036448945e-12,
          7.6008881270e-13,   1.3146405310e-13],
       [  1.4921344875e-10,   6.1633547703e-11,   2.5572230776e-11,
          1.0632830128e-11,   4.4264622187e-12,   1.8443245513e-12,
          3.2087725578e-13,   5.5986403288e-14],
       [  6.1492842463e-11,   2.5481570082e-11,   1.0608148954e-11,
          4.4264622187e-12,   1.8496194125e-12,   7.7369196122e-13,
          1.3576009069e-13,   2.3914810594e-14],
       [  2.5353714617e-11,   1.0540958082e-11,   4.4036448945e-12,
```

```
            1.8443245513e-12,   7.7369196122e-13,   3.2498644263e-13,
            5.7551104112e-14,   1.0244738582e-14],
          [  4.3137593878e-12,   1.8059692534e-12,   7.6008881270e-13,
            3.2087725578e-13,   1.3576009069e-13,   5.7551104112e-14,
            1.0403917951e-14,   1.8976295583e-15],
          [  7.3465498888e-13,   3.0985581496e-13,   1.3146405310e-13,
            5.5986403288e-14,   2.3914810594e-14,   1.0244738582e-14,
            1.8976295583e-15,   3.5672355835e-16]]
          )
    return x, gv.gvar(ymean, ycov)

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

We are not sure *a priori* how many exponentials are needed to fit our data. Consequently we write our code to try fitting with each of `nexp=1,2,3..6` terms. (The pieces of the code involving `p0` are optional; they make the more complicated fits go about 30 times faster since the output from one fit is used as the starting point for the next fit — see the discussion of the `p0` parameter for *`lsqfit.nonlinear_fit`*.) Running this code produces the following output, which is reproduced here in some detail in order to illustrate a variety of features:

```
************************************* nexp = 1
Least Square Fit:
  chi2/dof [dof] = 1.2e+03 [8]    Q = 0    logGBF = -4837.2

Parameters:
            a 0   0.00735 (59)      [  0.50 (40) ]  *
            E 0    1.1372 (49)      [  1.00 (40) ]

Settings:
  svdcut/n = 1e-12/1    tol = (1e-08*,1e-10,1e-10)    (itns/time = 11/0.0)

************************************* nexp = 2
Least Square Fit:
  chi2/dof [dof] = 2.2 [8]    Q = 0.024    logGBF = 111.69

Parameters:
            a 0    0.4024 (40)      [  0.50 (40) ]
              1    0.4471 (46)      [  0.50 (40) ]
            E 0   0.90104 (51)      [  1.00 (40) ]
              1    1.8282 (14)      [  2.00 (40) ]

Settings:
  svdcut/n = 1e-12/1    tol = (1e-08*,1e-10,1e-10)    (itns/time = 8/0.0)

************************************* nexp = 3
Least Square Fit:
  chi2/dof [dof] = 0.63 [8]    Q = 0.76    logGBF = 116.29

Parameters:
            a 0    0.4019 (40)      [  0.50 (40) ]
              1     0.406 (14)      [  0.50 (40) ]
              2      0.61 (36)      [  0.50 (40) ]
            E 0   0.90039 (54)      [  1.00 (40) ]
              1    1.8026 (82)      [  2.00 (40) ]
              2      2.83 (19)      [  3.00 (40) ]

Settings:
  svdcut/n = 1e-12/1    tol = (1e-08*,1e-10,1e-10)    (itns/time = 27/0.0)
```

```
E1/E0 = 2.0020(86)   E2/E0 = 3.14(21)
a1/a0 = 1.011(32)   a2/a0 = 1.52(89)

************************************ nexp = 4
Least Square Fit:
  chi2/dof [dof] = 0.63 [8]    Q = 0.76    logGBF = 116.3

Parameters:
            a 0    0.4019 (40)      [  0.50 (40) ]
              1     0.406 (14)      [  0.50 (40) ]
              2      0.61 (36)      [  0.50 (40) ]
              3      0.50 (40)      [  0.50 (40) ]
            E 0   0.90039 (54)      [  1.00 (40) ]
              1    1.8026 (82)      [  2.00 (40) ]
              2      2.83 (19)      [  3.00 (40) ]
              3      4.00 (40)      [  4.00 (40) ]

Settings:
  svdcut/n = 1e-12/1    tol = (1e-08*,1e-10,1e-10)    (itns/time = 9/0.0)

E1/E0 = 2.0020(86)   E2/E0 = 3.14(21)
a1/a0 = 1.011(32)   a2/a0 = 1.52(89)

************************************ nexp = 5
Least Square Fit:
  chi2/dof [dof] = 0.63 [8]    Q = 0.76    logGBF = 116.3

Parameters:
            a 0    0.4019 (40)      [  0.50 (40) ]
              1     0.406 (14)      [  0.50 (40) ]
              2      0.61 (36)      [  0.50 (40) ]
              3      0.50 (40)      [  0.50 (40) ]
              4      0.50 (40)      [  0.50 (40) ]
            E 0   0.90039 (54)      [  1.00 (40) ]
              1    1.8026 (82)      [  2.00 (40) ]
              2      2.83 (19)      [  3.00 (40) ]
              3      4.00 (40)      [  4.00 (40) ]
              4      5.00 (40)      [  5.00 (40) ]

Settings:
  svdcut/n = 1e-12/1    tol = (1e-08*,1e-10,1e-10)    (itns/time = 4/0.0)

E1/E0 = 2.0020(86)   E2/E0 = 3.14(21)
a1/a0 = 1.011(32)   a2/a0 = 1.52(89)

************************************ nexp = 6
Least Square Fit:
  chi2/dof [dof] = 0.63 [8]    Q = 0.76    logGBF = 116.3

Parameters:
            a 0    0.4019 (40)      [  0.50 (40) ]
              1     0.406 (14)      [  0.50 (40) ]
              2      0.61 (36)      [  0.50 (40) ]
              3      0.50 (40)      [  0.50 (40) ]
              4      0.50 (40)      [  0.50 (40) ]
              5      0.50 (40)      [  0.50 (40) ]
            E 0   0.90039 (54)      [  1.00 (40) ]
```

```
               1    1.8026 (82)       [  2.00 (40) ]
               2      2.83 (19)       [  3.00 (40) ]
               3      4.00 (40)       [  4.00 (40) ]
               4      5.00 (40)       [  5.00 (40) ]
               5      6.00 (40)       [  6.00 (40) ]

Settings:
  svdcut/n = 1e-12/1    tol = (1e-08*,1e-10,1e-10)     (itns/time = 2/0.0)

E1/E0 = 2.0020(86)   E2/E0 = 3.14(21)
a1/a0 = 1.011(32)   a2/a0 = 1.52(89)

================= Error Budget Analysis
Values:
               E2/E0: 3.14(21)
               E1/E0: 2.0020(86)
               a2/a0: 1.52(89)
               a1/a0: 1.011(32)

Partial % Errors:
                E2/E0     E1/E0     a2/a0     a1/a0
-----------------------------------------------------
        a:       5.47      0.07     52.75      0.82
        E:       3.23      0.12     25.36      1.04
        y:       2.08      0.40      5.24      2.78
-----------------------------------------------------
    total:       6.72      0.43     58.78      3.15
```

There are several things to notice here:

- Clearly two exponentials (`nexp=2`) are not sufficient. The `chi**2` per degree of freedom (`chi2/dof`) is significantly larger than one. The `chi**2` improves substantially for `nexp=3` exponentials, and there is essentially no change when further exponentials are added.
- The best-fit values for each parameter are listed for each of the fits, together with the prior values (in brackets, on the right). Values for each `a[i]` and `E[i]` are listed in order, starting at the points indicated by the labels `a` and `E`. Asterisks are printed at the end of the line if the mean best-fit value differs from the prior's mean by more than one standard deviation (see `nexp=1`); 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 data, prior, or fit function.

  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 made the data. For example, `a` and `E` for the first exponential are 0.402(4) and 0.9004(5), respectively, from the fit, while the exact answers are 0.4 and 0.9; and we get 0.406(14) and 1.803(8) for the second exponential where the exact values are 0.4 and 1.8.
- Note in the fit with `nexp=4` how the mean and standard deviation for the parameters governing the fourth (and last) exponential are identical to the values in the corresponding priors: 0.50(40) from the fit for `a` and 4.0(4) for `E`. This tells us that our fit data have no information to add to what we knew *a priori* about these parameters — there isn't enough data and what we have isn't accurate enough.

  This situation remains true of further terms as they are added in the `nexp=5` and later fits. This is why the fit results stop changing once we have `nexp=3` exponentials. There is no point in including further exponentials, beyond the need to verify that the fit has indeed converged. Note that the underlying function from which the data came had 100 exponential terms.
- The last fit includes `nexp=6` exponentials and therefore has 12 parameters. This is in a fit to 8 `y`s. Old-fashioned fits, without priors, are impossible when the number of parameters exceeds the number of data points. That is

clearly not the case here, where the number of terms and parameters can be made arbitrarily large, eventually (after `nexp=3` terms) with no effect at all on the results.

The reason is that the prior that we include for each new parameter is, in effect, a new piece of data (equal to the mean and standard deviation of the *a priori* expectation for that parameter). Each prior 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=6` fit, for example, we actually have 20 pieces of data to fit: the 8 `ys` plus the 12 prior values for the 12 parameters.

The function of priors as fit data becomes obvious if we rewrite our fit function as

```
import numpy as np

def fcn(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 dict(
        y=sum(ai * np.exp(-Ei * x) for ai, Ei in zip(a, E)),
        a=p['a'],
        b=p['b'],
        )
```

and make the following change to the `main()` function:

```
prior = make_prior(nexp)
data = (x, dict(y=y, a=prior['a'], b=prior['b']))
fit = lsqfit.nonlinear_fit(data=data, fcn=fcn, prior=None, p0=p0)
```

This gives exactly the same results, but now with the prior explicitly built into the fit function and data.

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 20-12=8 in this last case. With priors for every parameter, the number of degrees of freedom is always equal to the number of `ys`, irrespective of how many fit parameters there are.

- The Gaussian Bayes Factor (whose logarithm is `logGBF` in the output) is a measure of the likelihood that the actual data being fit could have come from a theory with the prior and fit function used in the fit. The larger this number, the more likely it is that prior/fit-function and data could be related. Here it grows dramatically from the first fit (`nexp=1`) but then stops changing after `nexp=3`. The implication is that this data is much more likely to have come from a theory with `nexp>=3` than one with `nexp=1`.

- 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 ratios automatically account for any correlations in the statistical errors for different parameters. This is evident in the ratio `a1/a0`, which would be 1.010(35) if there was no statistical correlation between our estimates for `a1` and `a0`, but in fact is 1.010(31) in this fit. The modest (positive) correlation is clear from the correlation matrix:

```
>>> print(gv.evalcorr(fit.p['a'][:2]))
[[ 1.          0.36353303]
 [ 0.36353303  1.        ]]
```

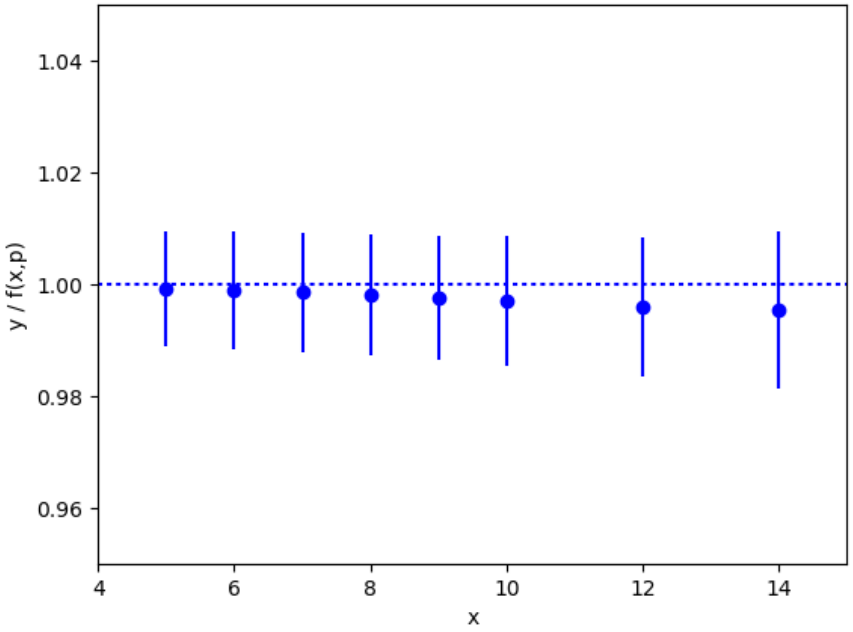
- After the last fit, the code uses function `gvar.fmt_errorbudget` to create an error budget. This requires dictionaries of fit inputs and outputs, and uses the dictionary keys to label columns and rows, respectively, in the error budget table. The table shows, for example, that the 0.43% uncertainty in `E1/E0` comes mostly from the fit data (0.40%), with small contributions from the uncertainties in the priors for `a` and `E` (0.07% and 0.12%, respectively). The total uncertainty is the sum in quadrature of these errors. This breakdown suggests that

reducing the errors in `y` by 25% might reduce the error in `E1/E0` to around 0.3% (and it does). The uncertainty in `E2/E0`, on the other hand, comes mostly from the priors and is less likely to improve (it doesn't).

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

```
Fit:
      x[k]               y[k]           f(x[k],p)
-----------------------------------------------------
         5      0.004502 (46)       0.004506 (46)
         6      0.001817 (19)       0.001819 (19)
         7     0.0007362 (79)      0.0007373 (78)
         8     0.0002987 (33)      0.0002993 (32)
         9     0.0001213 (14)      0.0001216 (13)
        10    0.00004926 (57)     0.00004941 (56)
        12       8.13(10)e-06        8.160(96)e-06
        14      1.342(19)e-06       1.348(17)e-06
```

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



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

```
import matplotlib.pyplot as plt
ratio = y / f(x, fit.pmean)
plt.xlim(4, 15)
plt.ylim(0.95, 1.05)
plt.xlabel('x')
plt.ylabel('y / f(x,p)')
plt.errorbar(x=x, y=gv.mean(ratio), yerr=gv.sdev(ratio), fmt='ob')
plt.plot([4.0, 21.0], [1.0, 1.0], 'b:')
```

```
plt.show()
```

1.4 Chained Fits; Large Data Sets

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. Here we look at two examples that exploit the possibility of chaining fits, where the output of one fit is an input (the prior) to another.

Imagine first 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.)

We can combine the new data with the old fit results by creating a new fit that uses the best-fit parameters, `fit.p`, from the old fit as its prior. To try this out, we modify the `main()` function in the previous section, adding the new fit at the end:

```
def main():
    x, y = make_data()              # collect fit data
    p0 = None                       # make larger fits go faster (opt.)
    for nexp in range(1, 5):
        prior = make_prior(nexp)
        fit = lsqfit.nonlinear_fit(data=(x, y), fcn=fcn, prior=prior, p0=p0)
        if fit.chi2 / fit.dof < 1.:
            p0 = fit.pmean          # starting point for next fit (opt.)

    # print nexp=4 fit results
    print('---------------------- original fit')
    print(fit)
    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])

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

    prior = fit.p                   # prior = best-fit parameters from nexp=4 fit
    data = gv.gvar(1, 1e-5)         # new data for the ratio

    newfit = lsqfit.nonlinear_fit(data=data, fcn=ratio, prior=prior)
    print('\n---------------------- new fit to extra information')
    print(newfit)
    E = newfit.p['E']
    a = newfit.p['a']
    print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
    print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
```

The results of the new fit (to one piece of new data) are at the end of the output:

```
---------------------- original fit
Least Square Fit:
  chi2/dof [dof] = 0.63 [8]    Q = 0.76    logGBF = 116.3
```

```
Parameters:
             a 0    0.4019 (40)       [  0.50 (40) ]
               1     0.406 (14)       [  0.50 (40) ]
               2      0.61 (36)       [  0.50 (40) ]
               3      0.50 (40)       [  0.50 (40) ]
             E 0   0.90039 (54)       [  1.00 (40) ]
               1    1.8026 (82)       [  2.00 (40) ]
               2      2.83 (19)       [  3.00 (40) ]
               3      4.00 (40)       [  4.00 (40) ]

Settings:
  svdcut/n = 1e-12/1    tol = (1e-08*,1e-10,1e-10)    (itns/time = 3/0.0)

E1/E0 = 2.0020(86)   E2/E0 = 3.14(21)
a1/a0 = 1.011(32)   a2/a0 = 1.52(89)


---------------------- new fit to extra information
Least Square Fit:
  chi2/dof [dof] = 0.12 [1]    Q = 0.73    logGBF = 2.4648

Parameters:
             a 0    0.4018 (40)       [  0.4019 (40) ]
               1    0.4018 (40)       [   0.406 (14) ]
               2      0.57 (34)       [    0.61 (36) ]
               3      0.50 (40)       [    0.50 (40) ]
             E 0   0.90033 (51)       [ 0.90039 (54) ]
               1    1.7998 (13)       [  1.8026 (81) ]
               2      2.79 (14)       [    2.83 (19) ]
               3      4.00 (40)       [    4.00 (40) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 14/0.0)

E1/E0 = 1.9991(12)   E2/E0 = 3.10(16)
a1/a0 = 1.000000(10)   a2/a0 = 1.43(85)
```

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

It might seem odd that `E[1]`, for example, is changed at all, since the fit function, `ratio(p)`, makes no mention of it. This is not surprising, however, since `ratio(p)` does depend upon `a[1]`, and `a[1]` is strongly correlated with `E[1]` through the prior (correlation coefficient of 0.94). 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]`.

Obviously, we can use further fits in order to incorporate additional 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 cumulative result of all of these fits.

Note that this particular problem can be done much more simply using a weighted average (*`lsqfit.wavg()`*). Adding the following code onto the end of the `main()` function above

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

print('chi2/dof = {:.2f}\n' .format(new_p.chi2 / new_p.dof))
```

```
print('E:', new_p['E'][:4])
print('a:', new_p['a'][:4])
print('a1/a0:', new_p['a1/a0'])
```

gives the following output:

```
chi2/dof = 0.12

E: [0.90033(51) 1.7998(13) 2.79(14) 4.00(40)]
a: [0.4018(40) 0.4018(40) 0.57(34) 0.50(40)]
a1/a0: 1.000000(10)
```

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

One place where chained fits can be useful is when there is lots of fit data. Imagine, as a second example, a situation that involves 10,000 highly correlated `y[i]`s. A straight fit would take a very long time because part of the fit process involves diagonalizing the fit data's (dense) 10,000×10,000 covariance matrix. Instead we break the data up into batches of 100 and do chained fits of one batch after another:

```
# read data from disk
x, y = read_data()
print('x = [{}  {} ... {}]'.format(x[0], x[1], x[-1]))
print('y = [{}  {} ... {}]'.format(y[0], y[1], y[-1]))
print('corr(y[0],y[9999]) =', gv.evalcorr([y[0], y[-1]])[1,0])
print()

# fit function and prior
def fcn(x, p):
    return p[0] + p[1] * np.exp(- p[2] * x)
prior = gv.gvar(['0(1)', '0(1)', '0(1)'])

# Nstride fits, each to nfit data points
nfit = 100
Nstride = len(y) // nfit
fit_time = 0.0
for n in range(0, Nstride):
    fit = lsqfit.nonlinear_fit(
        data=(x[n::Nstride], y[n::Nstride]), prior=prior, fcn=fcn
        )
    prior = fit.p
    if n in [0, 9]:
        print('******* Results from ', (n+1) * nfit, 'data points')
        print(fit)
print('******* Results from ', Nstride * nfit, 'data points (final)')
print(fit)
```

In the loop, we fit only 100 data points at a time, but the prior we use is the best-fit result from the fit to the previous 100 data points, and its prior comes from fitting the 100 points before those, and so on for 100 fits in all. The output from this code is:

```
x = [0.2  0.200080008001 ... 1.0]
y = [0.836(10)  0.835(10) ... 0.686(10)]
corr(y[0],y[9999]) = 0.990099009901

******* Results from  100 data points
Least Square Fit:
  chi2/dof [dof] = 1.1 [100]    Q = 0.23    logGBF = 523.92

Parameters:
              0    0.494 (13)      [  0.0 (1.0) ]
              1   0.3939 (75)      [  0.0 (1.0) ]
              2    0.715 (23)      [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 11/0.1)

******* Results from  1000 data points
Least Square Fit:
  chi2/dof [dof] = 1.1 [100]    Q = 0.29    logGBF = 544.96

Parameters:
              0    0.491 (10)      [  0.492 (10) ]
              1   0.3969 (24)      [ 0.3965 (25) ]
              2   0.7084 (70)      [ 0.7095 (74) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 6/0.0)

******* Results from  10000 data points (final)
Least Square Fit:
  chi2/dof [dof] = 1 [100]    Q = 0.48    logGBF = 548.63

Parameters:
              0     0.488 (10)       [   0.488 (10) ]
              1   0.39988 (77)       [ 0.39982 (78) ]
              2    0.7002 (23)       [  0.7003 (23) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 4/0.4)
```

It shows the errors on `p[1]` and `p[2]` decreasing steadily as more data points are included. The error on `p[0]`, however, hardly changes at all. This is a consequence of the strong correlation between different `y[i]`s (and its lack of x-dependence). The "correct" answers here are 0.5, 0.4 and 0.7.

Chained fits are slower that straight fits with large amounts of *uncorrelated* data, provided *lsqfit.nonlinear_fit* is informed ahead of time that the data are uncorrelated (by default it checks for correlations, which can be expensive for lots of data). The fitter is informed by using argument `udata` instead of `data` to specify the fit data:

```
x, y = read_data()
print('x = [{}  {} ... {}]'.format(x[0], x[1], x[-1]))
print('y = [{}  {} ... {}]'.format(y[0], y[1], y[-1]))
print()

# fit function and prior
def fcn(x, p):
    return p[0] + p[1] * np.exp(- p[2] * x)
```

```
prior = gv.gvar(['0(1)', '0(1)', '0(1)'])

fit = lsqfit.nonlinear_fit(udata=(x, y), prior=prior, fcn=fcn)
print(fit)
```

Using `udata` rather than `data` causes *`lsqfit.nonlinear_fit`* to ignore correlations in the data, whether they exist or not. Uncorrelated fits are typically much faster when fitting large amounts of data, so it is then possible to fit much more data (e.g., 1,000,000 or more `y[i]`s is straightforward on a laptop).

1.5 `x` has Errors

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

To illustrate, consider the data assembled by the following `make_data` function:

```
import gvar as gv

def make_data():
    x = gv.gvar([
        '0.73(50)',   '2.25(50)',  '3.07(50)',  '3.62(50)',  '4.86(50)',
        '6.41(50)',   '6.39(50)',  '7.89(50)',  '9.32(50)',  '9.78(50)',
        '10.83(50)', '11.98(50)', '13.37(50)', '13.84(50)', '14.89(50)'
        ])
    y = gv.gvar([
         '3.85(70)',  '5.5(1.7)',  '14.0(2.6)',   '21.8(3.4)',   '47.0(5.2)',
        '79.8(4.6)', '84.9(4.6)',  '95.2(2.2)',   '97.65(79)',   '98.78(55)',
        '99.41(25)', '99.80(12)', '100.127(77)', '100.202(73)', '100.203(71)'
        ])
    return x,y
```

The function call `x,y = make_data()` returns values for the `x[i]`s and the corresponding `y[i]`s, where now both are `gvar.GVar`s.

We want to fit the `y` values with a function of the form:

```
b0 / ((1 + gv.exp(b1 - b2 * x)) ** (1. / b3)).
```

So we have two sets of parameters for which we need priors: the `b[i]`s and the `x[i]`s:

```
import gvar as gv

def make_prior(x):
    prior = gv.BufferDict()
    prior['b'] = gv.gvar(['0(500)', '0(5)', '0(5)', '0(5)'])
    prior['x'] = x
    return prior
```

The prior values for the `x[i]` are just the values returned by `make_data()`. The corresponding fit function is:

```
import gvar as gv

def fcn(p):
    b0, b1, b2, b3 = p['b']
```

```
    x = p['x']
    return b0 / ((1. + gv.exp(b1 - b2 * x)) ** (1. / b3))
```

where the dependent variables `x[i]` are no longer arguments of the function, but rather are fit parameter in dictionary `p`.

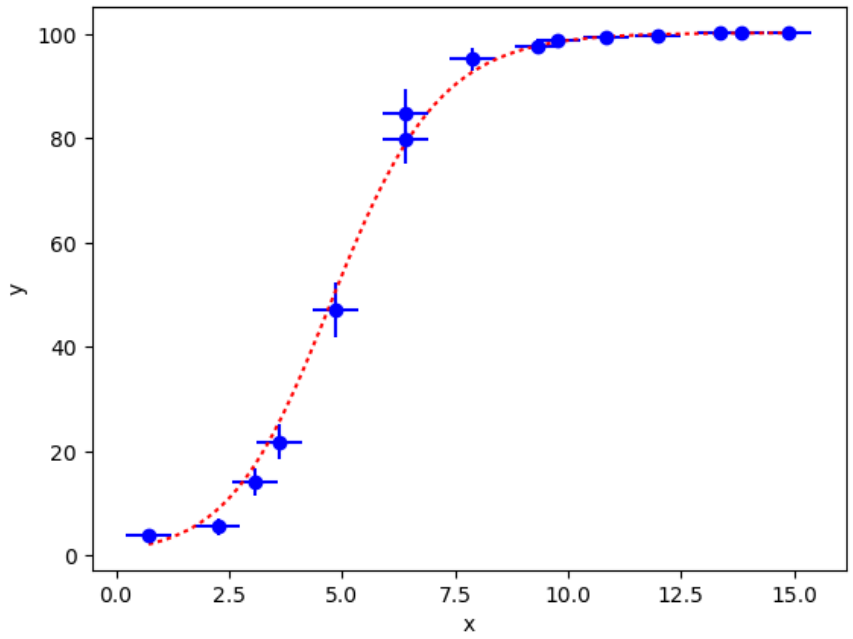
The actual fit is now straightforward:

```
import lsqfit

x, y = make_data()
prior = make_prior(x)
fit = lsqfit.nonlinear_fit(prior=prior, data=y, fcn=fcn)
print(fit)
```

This generates the following output:

```
Least Square Fit:
  chi2/dof [dof] = 0.35 [15]    Q = 0.99    logGBF = -40.156

Parameters:
            b 0   100.238 (60)       [    0 (500) ]
              1      3.5 (1.2)       [  0.0 (5.0) ]
              2    0.797 (87)        [  0.0 (5.0) ]
              3     0.77 (35)        [  0.0 (5.0) ]
            x 0     1.26 (41)        [  0.73 (50) ]  *
              1     1.87 (34)        [  2.25 (50) ]
              2     2.84 (28)        [  3.07 (50) ]
              3     3.42 (29)        [  3.62 (50) ]
              4     4.72 (32)        [  4.86 (50) ]
              5     6.45 (33)        [  6.41 (50) ]
              6     6.69 (35)        [  6.39 (50) ]
              7     8.15 (36)        [  7.89 (50) ]
              8     9.30 (35)        [  9.32 (50) ]
              9     9.91 (37)        [  9.78 (50) ]
             10    10.77 (37)        [ 10.83 (50) ]
             11    11.70 (38)        [ 11.98 (50) ]
             12    13.34 (46)        [ 13.37 (50) ]
             13    13.91 (48)        [ 13.84 (50) ]
             14    14.88 (50)        [ 14.89 (50) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 13/0.1)
```

The fit gives new results for the `b[i]` parameters that are much improved from our prior estimates. Results for many of the `x[i]`s are improved as well, by information from the fit data. The following plot shows the fit (dashed line) compared with the input data for `y`:



1.6 Correlated Parameters; Gaussian Bayes Factor

`gvar.GVar` objects allow for complicated priors, including priors that correlate different fit parameters. The following fit analysis code illustrates how this is done:

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

def main():
    x, y = make_data()
    prior = make_prior()
    fit = lsqfit.nonlinear_fit(prior=prior, data=(x,y), fcn=fcn)
    print(fit)
    print('p1/p0 =', fit.p[1] / fit.p[0], 'p3/p2 =', fit.p[3] / fit.p[2])
    print('corr(p0,p1) =', gv.evalcorr(fit.p[:2])[1,0])

def make_data():
    x = np.array([
        4., 2., 1., 0.5, 0.25, 0.167, 0.125, 0.1, 0.0833, 0.0714, 0.0625
        ])
    y = gv.gvar([
        '0.198(14)', '0.216(15)', '0.184(23)', '0.156(44)', '0.099(49)',
        '0.142(40)', '0.108(32)', '0.065(26)', '0.044(22)', '0.041(19)',
        '0.044(16)'
        ])
    return x, y

def make_prior():
    p = gv.gvar(['0(1)', '0(1)', '0(1)', '0(1)'])
    p[1] = 20 * p[0] + gv.gvar('0.0(1)')        # p[1] correlated with p[0]
    return p
```

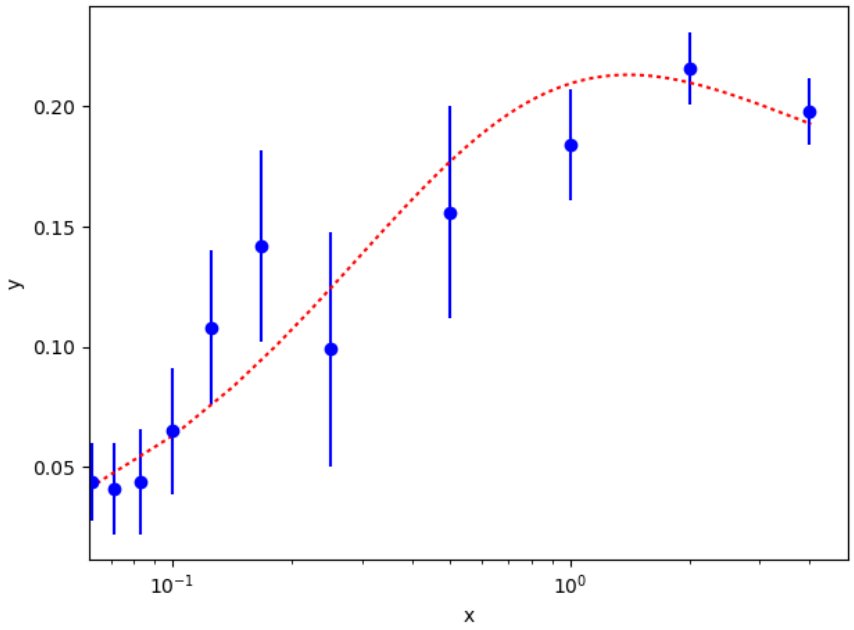
```
def fcn(x, p):
    return (p[0] * (x**2 + p[1] * x)) / (x**2 + x * p[2] + p[3])

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

Here, again, functions `make_data()` and `make_prior()` assemble the fit data and prior, and parameters `p[i]` are adjusted by the fitter to make `fcn(x[i], p)` agree with the data value `y[i]`. The priors are fairly broad (0 ± 1) for all of the parameters, except for `p[1]`. The prior introduces a tight relationship between `p[1]` and `p[0]`: it sets `p[1]=20*p[0]` up to corrections of order 0 ± 0.1. This *a priori* relationship is built into the prior and restricts the fit.

Running the code gives the following output:

```
Least Square Fit:
  chi2/dof [dof] = 0.61 [11]    Q = 0.82    logGBF = 19.129

Parameters:
              0   0.149 (17)      [  0.0 (1.0) ]
              1    2.97 (34)      [     0 (20) ]
              2    1.23 (61)      [  0.0 (1.0) ]  *
              3    0.59 (15)      [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 20/0.1)

p1/p0 = 19.97(67)     p3/p2 = 0.48(22)
corr(p0,p1) = 0.957067820817
```

Note how the ratio `p1/p0` is much more accurate than either quantity separately. The prior introduces a strong, positive correlation between the two parameters that survives the fit: the correlation coefficient is 0.96. Comparing the fit function with the best-fit parameters (dashed line) with the data shows a good fit:



If we omit the constraint in the prior,

```
def make_prior():
    p = gv.gvar(['0(1)', '0(20)', '0(1)', '0(1)'])
    return p
```

we obtain quite different fit results:

```
Least Square Fit:
  chi2/dof [dof] = 0.35 [11]    Q = 0.97    logGBF = 11.036

Parameters:
              0   0.211 (18)      [  0.0 (1.0) ]
              1   -0.02 (14)      [     0 (20) ]
              2    0.07 (10)      [  0.0 (1.0) ]
              3   0.008 (43)      [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 30/0.0)

p1/p0 = -0.08(64)     p3/p2 = 0.10(62)
corr(p0,p1) = -0.592869884703
```

Note that the Gaussian Bayes Factor (see `logGBF` in the output) is larger with the correlated prior (`logGBF = 19.1`) than it was for the uncorrelated prior (`logGBF = 11.0`). 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(19.1-11.0) = 3300` times more likely to get our `x,y` data from a theory with the correlated prior than from one with the uncorrelated prior.) This difference is significant despite the fact that the `chi**2` is lower for the uncorrelated case. `chi**2` tests goodness of fit, but there are usually more ways than one to get a good fit. Some are more plausible than others, and the Bayes Factor helps sort out which.

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

1.7 `y` has No Error; Marginalization

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. 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 it out from the input data. This is called *marginalization*.

To illustrate how it is done, we consider data that is generated from an infinite sum of decaying exponentials, like that in *Basic Fits*:

```
import numpy as np

  def make_data():
```

```
    x = np.array([ 1., 1.2, 1.4, 1.6, 1.8, 2., 2.2, 2.4, 2.6])
    y = np.array([
        0.2740471001620033,  0.2056894154005132,  0.158389402324004,
        0.1241967645280511,  0.0986901274726867,  0.0792134506060024,
        0.0640743982173861,  0.052143504367789 ,  0.0426383022456816,
        ])
    return x, y
```

Now `x,y = make_data()` returns nine `x[i]`s together with the corresponding `y[i]`s, but where the `y[i]`s are exact and so no longer represented by `gvar.GVar`s.

We want to fit these data with a sum of exponentials, as before:

```
import numpy as np

def fcn(x,p):
    a = p['a']        # array of a[i]s
    E = p['E']        # array of E[i]s
    return np.sum(ai * np.exp(-Ei*x) for ai, Ei in zip(a, E))
```

We know that the amplitudes `a[i]` are of order 0.5 ± 0.5, and that the leading exponent `E[0]` is 1 ± 0.1, as are the differences between subsequent exponents `dE[i] = E[i] - E[i-1]`. This *a priori* knowledge is encoded in the priors:

```
import numpy as np
import gvar as gv

def make_prior(nexp):
    prior = gv.BufferDict()
    prior['a'] = gv.gvar(nexp * ['0.5(5)'])
    dE = gv.gvar(nexp * ['1.0(1)'])
    prior['E'] = np.cumsum(dE)
    return prior
```

We use a large number of exponential terms since our `y[i]`s are exact: we keep 100 terms in all, but our results are unchanged with any number greater than about 10. Only a small number `nexp` of these are included in the fit function. The `100-nexp` terms left out are subtracted from the `y[i]` before the fit, using the prior values for the omitted parameters to evaluate these terms. This gives new fit data `ymod[i]`:

```
prior = make_prior(100)

# the first nexp terms are fit; the remainder go into ymod
fit_prior = gv.BufferDict()
ymod_prior = gv.BufferDict()
for k in prior:
  fit_prior[k] = prior[:nexp]
  ymod_prior[k] = prior[nexp:]

ymod = y - fcn(x, ymod_prior)
fit = lsqfit.nonlinear_fit(data=(x, ymod), prior=fit_prior, fcn=fcn)
```

By subtracting `fcn(x, ymod_prior)` from `y`, we remove the parameters that are in `ymod_prior` from the data, and consequently those parameters need not be included in fit function. The fitter uses only the parameters left in `fit_prior`.

Our complete code, therefore, is:

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

def main():
    x, y = make_data()
    prior = make_prior(100)             # 100 exponential terms in all
    p0 = None
    for nexp in range(1, 6):
        # marginalize the last 100 - nexp terms (in ymod_prior)
        fit_prior = gv.BufferDict()     # part of prior used in fit
        ymod_prior = gv.BufferDict()    # part of prior absorbed in ymod
        for k in prior:
            fit_prior[k] = prior[k][:nexp]
            ymod_prior[k] = prior[k][nexp:]
        ymod = y - fcn(x, ymod_prior)   # remove temrs in ymod_prior

        # fit modified data with just nexp terms (in fit_prior)
        fit = lsqfit.nonlinear_fit(
            data=(x, ymod), prior=fit_prior, fcn=fcn, p0=p0, tol=1e-10,
            )

        # print fit information
        print('*********************************** nexp =',nexp)
        print(fit.format(True))
        p0 = fit.pmean

    # print summary information and error budget
    E = fit.p['E']                      # best-fit parameters
    a = fit.p['a']
    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 prior':prior['E'], 'a prior':prior['a'],
        'svd cut':fit.svdcorrection,
        }
    print(fit.fmt_values(outputs))
    print(fit.fmt_errorbudget(outputs, inputs))

def fcn(x,p):
    a = p['a']       # array of a[i]s
    E = p['E']       # array of E[i]s
    return np.sum(ai * np.exp(-Ei*x) for ai, Ei in zip(a, E))

def make_prior(nexp):
    prior = gv.BufferDict()
    prior['a'] = gv.gvar(nexp * ['0.5(5)'])
    dE = gv.gvar(nexp * ['1.0(1)'])
    prior['E'] = np.cumsum(dE)
    return prior

def make_data():
    x = np.array([ 1., 1.2, 1.4, 1.6, 1.8, 2., 2.2, 2.4, 2.6])
    y = np.array([
        0.2740471001620033,  0.2056894154005132,  0.158389402324004 ,
        0.1241967645280511,  0.0986901274726867,  0.0792134506060024,
```

```
        0.0640743982173861,  0.052143504367789 ,  0.0426383022456816,
        ])
    return x, y

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

We loop over `nexp`, moving parameters from `ymod` back into the fit as `nexp` increases. The output from this script is:

```
************************************* nexp = 1
Least Square Fit:
  chi2/dof [dof] = 0.19 [9]    Q = 0.99    logGBF = 79.803

Parameters:
            a 0   0.4067 (32)      [  0.50 (50) ]
            E 0   0.9030 (16)      [  1.00 (10) ]

Fit:
     x[k]           y[k]        f(x[k],p)
----------------------------------------
        1     0.167 (74)     0.1648 (10)
      1.2     0.141 (49)    0.13760 (82)
      1.4     0.118 (32)    0.11487 (65)
      1.6     0.099 (22)    0.09589 (51)
      1.8     0.082 (14)    0.08004 (40)
        2    0.0686 (97)    0.06682 (31)
      2.2    0.0572 (65)    0.05578 (24)
      2.4    0.0476 (44)    0.04656 (19)
      2.6    0.0397 (30)    0.03887 (15)

Settings:
  svdcut/n = 1e-12/2    tol = (1e-10*,1e-10,1e-10)    (itns/time = 11/0.0)

************************************* nexp = 2
Least Square Fit:
  chi2/dof [dof] = 0.19 [9]    Q = 1    logGBF = 81.799

Parameters:
            a 0   0.4015 (23)      [  0.50 (50) ]
              1    0.435 (24)      [  0.50 (50) ]
            E 0   0.9007 (11)      [  1.00 (10) ]
              1    1.830 (28)      [  2.00 (14) ]  *

Fit:
     x[k]            y[k]        f(x[k],p)
----------------------------------------
        1      0.235 (28)      0.2330 (27)
      1.2      0.186 (15)      0.1847 (17)
      1.4     0.1484 (81)      0.1474 (10)
      1.6     0.1190 (44)     0.11833 (63)
      1.8     0.0960 (24)     0.09552 (38)
        2     0.0778 (13)     0.07749 (23)
      2.2    0.06331 (74)     0.06313 (14)
      2.4    0.05173 (41)    0.051624 (84)
      2.6    0.04242 (23)    0.042351 (50)

Settings:
```

```
  svdcut/n = 1e-12/2    tol = (1e-10*,1e-10,1e-10)    (itns/time = 27/0.0)

************************************ nexp = 3
Least Square Fit:
  chi2/dof [dof] = 0.2 [9]    Q = 0.99    logGBF = 83.077

Parameters:
            a 0    0.4011 (18)       [  0.50 (50) ]
              1     0.426 (28)       [  0.50 (50) ]
              2     0.468 (56)       [  0.50 (50) ]
            E 0   0.90045 (77)       [  1.00 (10) ]
              1     1.822 (27)       [  2.00 (14) ]  *
              2      2.84 (12)       [  3.00 (17) ]

Fit:
     x[k]             y[k]          f(x[k],p)
-----------------------------------------------
        1       0.260 (10)        0.2593 (22)
      1.2      0.1998 (45)        0.1995 (11)
      1.4      0.1559 (20)       0.15576 (54)
      1.6     0.12316 (91)       0.12305 (27)
      1.8     0.09824 (41)       0.09818 (13)
        2     0.07902 (19)      0.078988 (62)
      2.2    0.063990 (85)      0.063973 (30)
      2.4    0.052106 (38)      0.052098 (14)
      2.6    0.042622 (17)     0.0426176 (68)

Settings:
  svdcut/n = 1e-12/3    tol = (1e-10*,1e-10,1e-10)    (itns/time = 65/0.1)

************************************ nexp = 4
Least Square Fit:
  chi2/dof [dof] = 0.21 [9]    Q = 0.99    logGBF = 83.212

Parameters:
            a 0    0.4009 (10)       [  0.50 (50) ]
              1     0.424 (22)       [  0.50 (50) ]
              2     0.469 (61)       [  0.50 (50) ]
              3     0.426 (94)       [  0.50 (50) ]
            E 0   0.90036 (44)       [  1.00 (10) ]
              1     1.819 (19)       [  2.00 (14) ]  *
              2      2.83 (11)       [  3.00 (17) ]
              3      3.83 (15)       [  4.00 (20) ]

Fit:
     x[k]              y[k]           f(x[k],p)
-------------------------------------------------
        1       0.2687 (38)        0.26843 (95)
      1.2       0.2039 (14)        0.20376 (39)
      1.4      0.15778 (51)        0.15771 (16)
      1.6      0.12399 (19)       0.123955 (63)
      1.8     0.098616 (69)       0.098603 (25)
        2     0.079187 (26)       0.079182 (10)
      2.2    0.0640650 (96)      0.0640627 (39)
      2.4    0.0521401 (36)      0.0521392 (15)
      2.6    0.0426371 (13)     0.04263670 (60)

Settings:
```

```
  svdcut/n = 1e-12/3    tol = (1e-10*,1e-10,1e-10)    (itns/time = 143/0.1)

************************************ nexp = 5
Least Square Fit:
  chi2/dof [dof] = 0.21 [9]    Q = 0.99    logGBF = 83.137

Parameters:
            a 0    0.4009 (10)       [  0.50 (50) ]
              1     0.424 (22)       [  0.50 (50) ]
              2     0.468 (62)       [  0.50 (50) ]
              3      0.42 (11)       [  0.50 (50) ]
              4      0.45 (18)       [  0.50 (50) ]
            E 0   0.90036 (43)       [  1.00 (10) ]
              1     1.819 (19)       [  2.00 (14) ]  *
              2      2.83 (11)       [  3.00 (17) ]
              3      3.83 (15)       [  4.00 (20) ]
              4      4.83 (18)       [  5.00 (22) ]

Fit:
     x[k]               y[k]          f(x[k],p)
-------------------------------------------------
        1         0.2721 (14)        0.27196 (65)
      1.2        0.20516 (42)        0.20510 (21)
      1.4        0.15824 (13)       0.158219 (69)
      1.6       0.124154 (38)       0.124147 (23)
      1.8       0.098678 (12)      0.0986752 (78)
        2      0.0792099 (36)      0.0792090 (29)
      2.2      0.0640734 (11)      0.0640731 (12)
      2.4     0.05214320 (33)     0.05214310 (61)
      2.6     0.04263821 (10)     0.04263818 (35)

Settings:
  svdcut/n = 1e-12/3    tol = (1e-10*,1e-10,1e-10)    (itns/time = 246/0.3)

Values:
              E2/E0: 3.15(12)
              E1/E0: 2.021(20)
              a2/a0: 1.17(15)
              a1/a0: 1.057(52)

Partial % Errors:
               E2/E0     E1/E0     a2/a0     a1/a0
-----------------------------------------------------
  E prior:      3.86      0.86     12.07      4.50
  svd cut:      0.04      0.04      0.33      0.16
  a prior:      0.84      0.47      5.30      1.93
-----------------------------------------------------
    total:      3.95      0.98     13.19      4.90
```

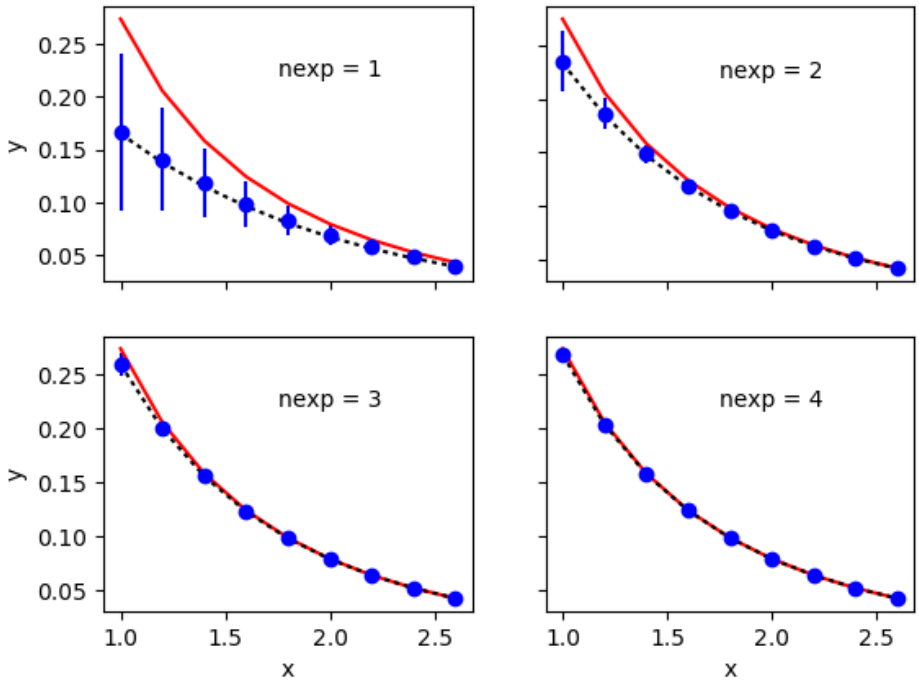
Here we use `fit.format(True)` to print out a table of `x` and `y` (actually `ymod`) values, together with the value of the fit function using the best-fit parameters. There are several things to notice:

- Even the `nexp=1` fit, where we fit the data with just a single exponential, gives results for the two parameters that are accurate to 1% or better. The results don't change much as further terms are shifted from `ymod` to the fit function, and stop changing completely by `nexp=4`.

  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`.

- `ymod` has large uncertainties when `nexp` is small, because of the uncertainties in the priors used to evaluate `fcn(x, ymod_prior)`. This is clear from the following plots:



The solid lines in these plot show the exact results, from `y` in the code. The dashed lines show the fit function with the best-fit parameters for the `nexp` terms used in each fit, and the data points show `ymod` — these last two agree well, as expected from the excellent `chi**2` values. The uncertainties in different `ymod[i]`s are highly correlated with each other because they come from the same priors (in `ymod_prior`). These correlations are evident in the plots and are essential to this procedure.

- Although we motivated this example by the need to deal with `ys` having no errors, it is straightforward to apply the same ideas to a situation where the `ys` have errors. Often in a fit we are interested in only one or two of many fit parameters. Getting rid of the uninteresting parameters (by absorbing them into `ymod`) can greatly reduce the number of parameters varied by the fit, thereby speeding up the fit. Here we are in effect doing a 100-exponential fit to our data, but actually fitting with only a handful of parameters (only 2 for `nexp=1`). Removing parameters in this way is called *marginalization*.

1.8 SVD Cuts and Roundoff Error

All of the fits discussed above have (default) SVD cuts of 1e-12. 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-12, we would have obtained the following output:

```
************************************ nexp = 5
Least Square Fit:
  chi2/dof [dof] = 0.21 [9]    Q = 0.99    logGBF = 85.355

Parameters:
```

```
            a 0    0.4009 (10)       [  0.50 (50) ]
              1     0.424 (22)       [  0.50 (50) ]
              2     0.469 (62)       [  0.50 (50) ]
              3      0.42 (11)       [  0.50 (50) ]
              4      0.46 (18)       [  0.50 (50) ]
            E 0   0.90036 (43)       [  1.00 (10) ]
              1     1.819 (19)       [  2.00 (14) ]  *
              2      2.83 (11)       [  3.00 (17) ]
              3      3.83 (15)       [  4.00 (20) ]
              4      4.83 (18)       [  5.00 (22) ]

Fit:
     x[k]               y[k]      f(x[k],p)
-----------------------------------------
        1        0.2721 (14)     0.272 (30)
      1.2       0.20516 (42)     0.205 (26)
      1.4       0.15824 (13)     0.158 (19)
      1.6      0.124154 (38)     0.124 (13)
      1.8      0.098678 (12)    0.0987 (89)
        2     0.0792099 (36)    0.0792 (61)
      2.2     0.0640734 (11)    0.0641 (43)
      2.4    0.05214320 (33)    0.0521 (31)
      2.6    0.04263821 (10)    0.0426 (24)

Settings:
  svdcut/n = 1e-19/0    tol = (1e-10*,1e-10,1e-10)     (itns/time = 284/0.3)

Values:
              E2/E0: 3(68)
              E1/E0: 2(11)
              a2/a0: 1(54)
              a1/a0: 1(30)

Partial % Errors:
               E2/E0     E1/E0     a2/a0     a1/a0
-------------------------------------------------
  E prior:   1065.47    276.29   2277.85   1406.61
  svd cut:      0.00      0.00      0.00      0.00
  a prior:   1889.45    490.06   4009.51   2496.57
-------------------------------------------------
    total:   2169.15    562.58   4611.37   2865.56
```

The standard deviations quoted for `E1/E0`, *etc.* are much too large compared with the standard deviations than what we obtained in the previous section. This is due to roundoff error. The strong correlations between the different data points (`ymod[i]` — see the previous section) in this analysis result in a data covariance matrix that is too ill-conditioned without an SVD cut.

The inverse of the data's 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 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=5`) cover a range of about 18 orders of magnitude — too large to be handled in normal double precision computation. The smallest eigenvalues and their eigenvectors are likely to be quite inaccurate.

A 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-12)
```

This regulates the singularity of the covariance matrix by replacing its smallest eigenvalues with a larger, minimum eigenvalue. The cost is less precision in the final results since we are decreasing the precision of the input `y` data. This is a conservative move, but numerical stability is worth the trade off. The listing shows that 3 eigenvalues are modified when `svdcut=1e-12` (see entry for `svdcut/n`); no eigenvalues are changed when `svdcut=1e-19`.

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

The results shown in the previous section include an error budget, and it has an entry for the error introduced by the (default) SVD cut (obtained from `fit.svdcorrection`). The contribution is negligible. It is zero when `svdcut=1e-19`, of course, but the instability caused by the ill-conditioned covariance matrix in that case makes it unacceptable.

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

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

gives a new, non-singular matrix

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

where only the upper left sub-matrix is different.

lsqfit.nonlinear_fit uses a default value for `svdcut` of 1e-12. This default can be overridden, as shown above, but for many problems it is a good choice. Roundoff errors become more accute, however, when there are strong correlations between different parts of the fit data or prior. Then much larger `svdcut`s may be needed.

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

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

applies different SVD cuts to the prior and data.

Note that taking `svdcut=-1e-12`, 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-12` (giving, for example, 2.094(94) for `E1/E0`, which is almost five times less precise). Dropping modes is equivalent to setting the corresponding variances to infinity, which is (obviously) much more conservative and less realistic than setting them equal to the SVD-cutoff variance.

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

1.9 y has Unknown Errors

There are situations where the input data `y` is known to have uncertainties, but where we do not know how big those uncertainties are. A common approach is to infer these uncertainties from the fluctuations of the data around the best-fit result.

As an example, consider the following data:

```
x = np.array([1., 2., 3., 4.])
y = np.array([3.4422, 1.2929, 0.4798, 0.1725])
```

We want to fit these data with a simple exponential:

```
p[0] * gv.exp( - p[1] * x)
```

where from we know *a priori* that `p[0]` is 10±1 and `p[1]` is 1±0.1. We assume that the relative uncertainty in `y` is `x`-independent and uncorrelated.

Our strategy is to introduce a relative error for the data and to vary its size to maximize the `logGBF` that results from a fit to our exponential. The choice that maximizes the Bayes Factor is the one that is favored by the data. This procedure is called the *Empirical Bayes* method.

This method is implemented in a driver program

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

which varies parameter `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 are varied as functions of `z`. The optimal fit (that is, the one for which `fit.logGBF` is maximum) and `z` are returned.

Here we want to vary the relative error assigned to the data values, so we use the following code, where the uncertainty in `y[i]` is set equal to `dy[i] = y[i] * z`:

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

# fit data and prior
x = np.array([1., 2., 3., 4.])
y = np.array([3.4422, 1.2929, 0.4798, 0.1725])
prior = gv.gvar(['10(1)', '1.0(1)'])

# fit function
def fcn(x, p):
    return p[0] * gv.exp(-p[1] * x)

# find optimal dy
def fitargs(z):
    dy = y * z
    newy = gv.gvar(y, dy)
    return dict(data=(x, newy), fcn=fcn, prior=prior)

fit, z = lsqfit.empbayes_fit(0.001, fitargs)
print(fit.format(True))
```

This code produces the following output:

```
Least Square Fit:
  chi2/dof [dof] = 0.59 [4]    Q = 0.67    logGBF = 7.4834

Parameters:
              0     9.44 (18)      [ 10.0 (1.0) ]
              1   0.9978 (68)      [  1.00 (10) ]

Fit:
     x[k]           y[k]       f(x[k],p)
----------------------------------------
        1     3.442 (54)      3.481 (45)
        2     1.293 (20)      1.283 (11)
        3    0.4798 (75)     0.4731 (40)
        4    0.1725 (27)     0.1744 (23)

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 3/0.0)
```

The variation in the data suggests a relative error of about 1.6% for the input data. The overall fit is excellent.

It is important to appreciate that the outcome of a such a fit depends in detail on the assumptions you make about `y`'s uncertainties `dy`. We assume `dy/y` is `x`-independent above, but we get a somewhat different answer if instead we assume that `dy` is constant. Then `fitrargs` becomes

```
def fitargs(z):
    dy =  np.ones_like(y) * z
    newy = gv.gvar(y, dy)
    return dict(data=(x, newy), fcn=fcn, prior=prior)
```

and the output is:

```
Least Square Fit:
  chi2/dof [dof] = 0.67 [4]    Q = 0.61    logGBF = 7.7643

Parameters:
              0    9.207 (47)      [ 10.0 (1.0) ]
              1   0.9834 (42)      [  1.00 (10) ]

Fit:
     x[k]           y[k]       f(x[k],p)
----------------------------------------
        1    3.4422 (66)     3.4435 (66)
        2    1.2929 (66)     1.2879 (50)
        3    0.4798 (66)     0.4817 (38)
        4    0.1725 (66)     0.1802 (22)   *

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 3/0.0)
```

The data suggest an uncertainty of 0.0066 in each `y[i]`. Results for the fit parameters `fit.p[i]` are similar in the two cases, but the error on `p[0]` is almost four times smaller with constant `dy`.

There is no way to tell from the data which of these error scenarios for `y` is correct. `logGBF` is slightly larger for the second fit, despite its larger `chi2/dof`, but the difference is not significant. There isn't enough data and it doesn't cover a large enough range to distinguish between these two options. Additional information about the data or data

taking is needed to decide.

The Empirical Bayes method for setting `dy` becomes trivial when there are no priors and when `dy` is assumed to be `x`-independent. Then it is possible to minimize the *chi**2* function without knowing `dy`, since `dy` factors out. The optimal `dy` is just the standard deviation of the fit residuals `y[i] - fcn(x[i],p)` with the best-fit parameters `p`. This assumption is implicit in most fit routines that fit data without errors (and without priors).

1.10 Tuning Priors with 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`. In effect we are using the data to get a feel for what is a reasonable prior. This procedure for setting priors is again, as in the previous section, an example of the Empirical Bayes method and can be implemented using function *`lsqfit.empbayes_fit()`*.

The following code illustrates how this is done:

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

x = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7])
y = np.array([
    '0.133426(95)', '0.20525(15)', '0.27491(20)', '0.32521(25)',
    '0.34223(28)', '0.32394(28)', '0.27857(27)'
    ])

def fcn(x, p):
    return gv.exp(-p[0] - p[1] * x - p[2] * x**2 - p[3] * x**3)

def fitargs(z):
    dp = z
    prior = gv.gvar([gv.gvar(0, dp) for i in range(4)])
    return dict(prior=prior, fcn=fcn, data=(x,y))

fit,z = lsqfit.empbayes_fit(1.0, fitargs)
print(fit.format(True))
```

Here the fitter varies parameters `p` until `fcn(x,p)` equals the input data `y`. We don't know *a priori* how large the coefficients `p[i]` are. In `fitargs` we assume they are all of order `dp = z`. Function `empbayes_fit` varies `z` to maximize `fit.logGBF`. The output is as follows:

```
Least Square Fit:
  chi2/dof [dof] = 0.81 [7]    Q = 0.58    logGBF = 21.274

Parameters:
              0   2.5904 (22)      [  0.0 (5.3) ]
              1   -6.530 (22)      [  0.0 (5.3) ]  *
              2    7.832 (65)      [  0.0 (5.3) ]  *
              3   -1.688 (55)      [  0.0 (5.3) ]

Fit:
     x[k]             y[k]         f(x[k],p)
-----------------------------------------------
      0.1    0.133426 (95)     0.133451 (92)
```

```
      0.2     0.20525 (15)     0.20512 (10)
      0.3     0.27491 (20)     0.27509 (14)
      0.4     0.32521 (25)     0.32516 (15)
      0.5     0.34223 (28)     0.34220 (19)
      0.6     0.32394 (28)     0.32392 (18)
      0.7     0.27857 (27)     0.27859 (26)

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 1/0.0)
```

The data suggest that the coefficients are of order 0 ± 5.3. The actual values of the parameters are, of course, consistent with the Empirical Bayes estimate.

The Bayes factor, `exp(fit.logGBF)`, is useful for deciding about fit functions as well as priors. If we repeat the analysis above but with the following data

```
x = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7])
y = np.array([
    '0.133213(95)', '0.20245(15)', '0.26282(19)', '0.29099(22)',
    '0.27589(22)', '0.22328(19)', '0.15436(14)'
    ])
```

we find that fits with 3 or 4 `p[i]`s give the following results:

```
========== fcn(x,p) = exp(-p[0] - p[1] * x - p[2] * x**2)
Least Square Fit:
  chi2/dof [dof] = 0.86 [7]    Q = 0.53    logGBF = 27.07

Parameters:
              0    2.5911 (12)      [  0.0 (5.3) ]
              1   -6.5420 (68)      [  0.0 (5.3) ]  *
              2    7.8711 (86)      [  0.0 (5.3) ]  *

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 8/0.0)


========== fcn(x,p) = exp(-p[0] - p[1] * x - p[2] * x**2 - p[3] * x**3)
Least Square Fit:
  chi2/dof [dof] = 0.82 [7]    Q = 0.57    logGBF = 22.617

Parameters:
              0   2.5920 (21)     [  0.0 (5.3) ]
              1   -6.553 (22)     [  0.0 (5.3) ]  *
              2    7.905 (64)     [  0.0 (5.3) ]  *
              3   -0.029 (54)     [  0.0 (5.3) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 11/0.0)
```

The two fits are almost equally good, giving almost the same `chi**2` values. The first fit, with only 3 `p[i]`s, however, has a significantly larger `logGBF`. This indicates that this data is `exp(27.1-22.6) = 90` times more likely to come from the theory with only 3 `p[i]`s than from the one with 4. The data much prefer the 3-parameter theory, and they do, as it turns out, come from such a theory. Note that the value for `p[3]` in the second case is consistent with zero, but the errors on the other parameters are much larger if it is included in the fit.

The Empirical Bayes procedure can be abused, because it is possible to make `logGBF` arbitrarily large. For example, setting

```
prior = gv.gvar([
    '2.5904 +- 2.6e-16', '-6.53012 +- 6.5e-16',
    '7.83211 +- 7.8e-16', '-1.68813 +- 1.7e-16',
    ])
```

in the problem above and then fitting gives `logGBF=52.2`, which is much larger than the alternatives above. This "prior" is ridiculous, however: it has means equal to the best-fit results with standard deviations that are 16 orders of magnitude smaller. This is the kind of prior you get from Empirical Bayes if you vary the means and standard deviations of all parameters independently.

Bayes Theorem explains what is wrong with such priors. The Bayes Factor is proportional to the probability `P(y|model)` that the fit data would arise given the model (priors plus fit function). When selecting models, we really want to maximize `P(model|y)`, the probability of the model given the data. These two probabilities are different, but are related by Bayes Theorem: `P(model|y)` is proportional to `P(y|model)` times `P(model)`, where `P(model)` is the *a priori* probability of the model being correct. When we choose a model by maximizing `logGBF` (that is, by maximizing `P(y|model)`), we are implicitly assuming that the various models we are considering are all equally likely candidates — that is, we are assuming that `P(model)` is approximately constant across the model space we are exploring. The *a priori* probability for the ridiculous prior just above is vanishingly small, and so comparing its `logGBF` to the others is nonsensical.

Note that `empbayes_fit()` allows `fitargs(z)` to return a dictionary of arguments for the fitter together with a `plausibility` for `z`, which corresponds to `log(P(model))` in the discussion above. This allows you steer the search away from completely implausible solutions.

Empirical Bayes tends to be most useful when varying the width of the prior for a single parameter, or varying the widths of a group of parameters together. It is also useful for validating (rather than setting) the choice of a prior or set of priors for a fit, by comparing the optimal choice (according to the data) with choice actually used.

1.11 Positive Parameters; Non-Gaussian Priors

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

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

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

```
import gvar as gv
import lsqfit

y = gv.gvar([
    '-0.17(20)', '-0.03(20)', '-0.39(20)', '0.10(20)', '-0.03(20)',
```

```
    '0.06(20)', '-0.23(20)', '-0.23(20)', '-0.15(20)', '-0.01(20)',
    '-0.12(20)', '0.05(20)', '-0.09(20)', '-0.36(20)', '0.09(20)',
    '-0.07(20)', '-0.31(20)', '0.12(20)', '0.11(20)', '0.13(20)'
    ])
```

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

```
prior = {'a':gv.gvar('0.02(2)')}                # a = average 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'])
```

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

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

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

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 4/0.0)

a = 0.004(18)
```

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

A better analysis uses a log-normal distribution for `a`:

```
prior = {}
prior['log(a)'] = gv.log(gv.gvar('0.02(2)'))    # log(a) not a

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

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

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

The result from this fit is

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

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

```
Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 25/0.0)

a = 0.012(11)
```

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

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

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

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

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

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

The result from this last fit is:

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

Parameters:
  erfinv(50a-1)   -0.42 (68)     [  0.00 (71) ]
-----------------------------------------------
          50a-1   -0.44 (64)     [  0.00 (80) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 15/0.0)

a = 0.011(13)
```

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

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

```
import lsqfit
import gvar as gv

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

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

gv.add_parameter_distribution('f', invf)
```

```
prior = {}
prior['f(a)'] = gv.gvar('0.00(75)')

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

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

does a fit with Gaussian parameter `f(a)`, which forces `a` to lie between 0 and 0.04. This fit gives `a=0.012(12)`, which again agrees well with log-normal fit. The prior 0 ± 0.75 for `f(a)` is chosen to make the prior probability distribution for parameter `a` almost flat across most (80%) of the interval 0.02 ± 0.02.

1.12 Faster Fitters

lsqfit.nonlinear_fit uses fitters from the Gnu Scientific Library (GSL) and/or from the `scipy` Python module to do the actual fitting, depending upon which of these is installed. It is worth trying a different fitter or fit algorithm if a fit is causing trouble, since different fitters are optimized for different problems. The fitter is selected using the `fitter` argument in *lsqfit.nonlinear_fit*. There are currently three fitters available:

fitter='gsl_multifit' The standard GSL least-squares fitter which is wrapped in Python class *lsqfit.gsl_multifit*. This is the default fitter provided GSL is installed. It offers a wide range of options, including several different algorithms that are selected by setting *lsqfit.nonlinear_fit* parameter `alg` equal to `'lm'`, `'subspace2D'`, `'dogleg'`, and so on. See the documentation.

fitter='gsl_v1_multifit' The GSL fitter from version 1 of the GSL library. This is wrapped in Python class *lsqfit.gsl_v1_multifit*. It was the fitter used in *lsqfit* versions earlier than version 9.0. It supports a few different algorithms (parameter `alg`) including `'lmsder'` and `'lmder'`.

fitter='scipy_least_squares' The standard `scipy` least-squares fitter, here provided with an *lsqfit* interface by class *lsqfit.scipy_least_squares*. This is the default fitter when GSL is not available. It also provides a variety of algorithms (set parameter `method`), and other options, such as loss functions for handling outliers. See the `scipy` documentation.

The default configurations for these fitters are chosen to emphasize robustness rather than speed, and therefore some of the non-default options can be much faster. Adding

```
fitter='gsl_multifit', alg='subspace2D', scaler='more', solver='cholesky'
```

to *lsqfit.nonlinear_fit*'s argument list, for example, can double or triple the fitter's speed for large problems. The more robust choices are important for challenging fits, but straightforward fits can be greatly accelerated by using different options. The `scipy_least_squares` fitter can also be much faster than the default. It is worth experimenting when fits become costly.

Method *lsqfit.nonlinear_fit.set()* modifies the defaults used by *lsqfit.nonlinear_fit*. For example, we can make the fast option mentioned above the default choice for any subsequent fit by calling:

```
lsqfit.nonlinear_fit.set(
    fitter='gsl_multifit',
    alg='subspace2D',
    scaler='more',
    solver='cholesky',
    )
```

Default values for parameters `extend`, `svdcut`, `debug`, `maxit`, `fitter`, and `tol` can be reset, as can any parameters that are sent to the underlying fitter (e.g., `alg`, `scaler`, and `solver` here). Calling the function with no arguments returns a dictionary containing the current defaults. `nonlinear_fit.set(clear=True)` restores the original defaults.

lsqfit.nonlinear_fit is easier to use than the underlying fitters because it can handle correlated data, and it automatically generates Jacobian functions for the fitter, using automatic differentiation. It also is integrated with the `gvar` module, which provides powerful tools for error propagation, generating error budgets, and creating potentially complicated priors for Bayesian fitting. The underlying fitters are available from *lsqfit* for use in other more specialized applications.

1.13 Debugging and Troubleshooting

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

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

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

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

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

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

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

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

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

```
>>> import numpy as np
>>> cov = gv.evalcov(prior.flat)     # prior's covariance matrix
```

```
>>> print(np.linalg.det(cov))        # determinant is zero
0.0
```

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

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

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

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

CHAPTER

TWO

NON-GAUSSIAN BEHAVIOR; TESTING FITS

2.1 Introduction

The various analyses in the Tutorial assume implicitly that every probability distribution relevant to a fit is Gaussian. The input data and priors are assumed Gaussian. The `chi**2` function is assumed to be well approximated by a Gaussian in the vicinity of its minimum, in order to estimate uncertainties for the best-fit parameters. Functions of those parameters are assumed to yield results that are described by Gaussian random variables. These assumptions are usually pretty good for high-statistics data, when standard deviations are small, but can lead to problems with low statistics.

Here we present three methods for testing these assumptions. Some of these techniques, like the *statistical bootstrap* and Bayesian integration, can also be used to analyze non-Gaussian results.

2.2 Bootstrap Error Analysis; Non-Gaussian Output

The bootstrap provides an efficient way to check on a fit's validity, and also a method for analyzing non-Gaussian outputs. The strategy is to:

1. make a large number of "bootstrap copies" of the original input data and prior that differ from each other by random amounts characteristic of the underlying randomness in the original data and prior (see the documentation for *`lsqfit.nonlinear_fit.bootstrap_iter()`* for more information);
2. repeat the entire fit analysis for each bootstrap copy of the data and prior, extracting fit results from each;
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.

To illustrate, we return to our fit in the section on *Correlated Parameters; Gaussian Bayes Factor*, where the uncertainties on the final parameters were relatively large. We will use a boostrap analysis to check the error estimates coming out of that fit. We do this by adding code right after the fit, in the `main()` function:

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

def main():
    x, y = make_data()
    prior = make_prior()
    fit = lsqfit.nonlinear_fit(prior=prior, data=(x,y), fcn=fcn)
    print(fit)
    print('p1/p0 =', fit.p[1] / fit.p[0], 'p3/p2 =', fit.p[3] / fit.p[2])
    print('corr(p0,p1) =', gv.evalcorr(fit.p[:2])[1,0])
```

```
    # boostrap analysis: collect bootstrap data
    print('\nBootstrap Analysis:')
    Nbs = 40                # number of bootstrap copies
    output = {'p':[], 'p1/p0':[], 'p3/p2':[]}
    for bsfit in fit.bootstrap_iter(Nbs):
        p = bsfit.pmean
        output['p'].append(p)
        output['p1/p0'].append(p[1] / p[0])
        output['p3/p2'].append(p[3] / p[2])

    # average over bootstrap copies and tabulate results
    output = gv.dataset.avg_data(output, bstrap=True)
    print(gv.tabulate(output))
    print('corr(p0,p1) =', gv.evalcorr(output['p'][:2])[1,0])

def make_data():
    x = np.array([
        4., 2., 1., 0.5, 0.25, 0.167, 0.125, 0.1, 0.0833, 0.0714, 0.0625
        ])
    y = gv.gvar([
        '0.198(14)', '0.216(15)', '0.184(23)', '0.156(44)', '0.099(49)',
        '0.142(40)', '0.108(32)', '0.065(26)', '0.044(22)', '0.041(19)',
        '0.044(16)'
        ])
    return x, y

def make_prior():
    p = gv.gvar(['0(1)', '0(1)', '0(1)', '0(1)'])
    p[1] = 20 * p[0] + gv.gvar('0.0(1)')      # p[1] correlated with p[0]
    return p

def fcn(x, p):
    return (p[0] * (x**2 + p[1] * x)) / (x**2 + x * p[2] + p[3])

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

The `bootstrap_iter` produces fits `bsfit` for each of `Nbs=40` different bootstrap copies of the input data (`y` and the prior). We collect the mean values for the various parameters and functions of parameters from each fit, ignoring the uncertainties, and then calculate averages and covariance matrices from these results using `gvar.dataset.avg_data()`.

Most of the bootstrap results agree with the results coming directly from the fit:

```
Least Square Fit:
  chi2/dof [dof] = 0.61 [11]    Q = 0.82    logGBF = 19.129

Parameters:
              0   0.149 (17)      [  0.0 (1.0) ]
              1    2.97 (34)      [     0 (20) ]
              2    1.23 (61)      [  0.0 (1.0) ]  *
              3    0.59 (15)      [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 20/0.1)

p1/p0 = 19.97(67)     p3/p2 = 0.48(22)
corr(p0,p1) = 0.957067820817
```

```
Bootstrap Averages:
  key/index         value
-------------------------
        p 0     0.146 (19)
          1      2.91 (29)
          2      1.14 (76)
          3      0.59 (16)
      p3/p2      0.49 (40)
      p1/p0     19.86 (80)
corr(p0,p1) = 0.949819729559
```

In particular, the bootstrap analysis confirms the previous error estimates (to within 10-30%, since `Nbs=40`) except for `p3/p2`, where the error is substantially larger in the bootstrap analysis.

If `p3/p2` is important, one might want to look more closely at its distribution. We use the bootstrap to create histograms of the probability distributions of `p3/p2` and `p1/p01` by adding the following code to the end of the `main()` function:

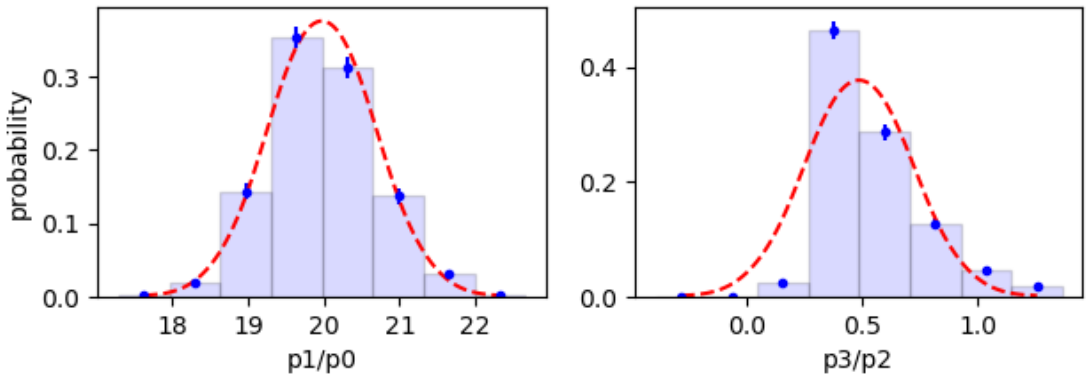
```
print('Histogram Analysis:')
count = {'p1/p0':[], 'p3/p2':[]}
hist = {
    'p1/p0':gv.PDFHistogram(fit.p[1] / fit.p[0]),
    'p3/p2':gv.PDFHistogram(fit.p[3] / fit.p[2]),
    }

# collect bootstrap data
for bsfit in fit.bootstrap_iter(n=1000):
    p = bsfit.pmean
    count['p1/p0'].append(hist['p1/p0'].count(p[1] / p[0]))
    count['p3/p2'].append(hist['p3/p2'].count(p[3] / p[2]))

# calculate averages and covariances
count = gv.dataset.avg_data(count)

# print histogram statistics and show plots
import matplotlib.pyplot as plt
pltnum = 1
for k in count:
    print(k + ':')
    print(hist[k].analyze(count[k]).stats)
    plt.subplot(1, 2, pltnum)
    plt.xlabel(k)
    hist[k].make_plot(count[k], plot=plt)
    if pltnum == 2:
        plt.ylabel('')
    pltnum += 1
plt.show()
```

Here we do 1000 bootstrap copies (rather than 40) to improve the accuracy of the bootstrap results. The output from this code shows statistical analyses of the histogram data for `p1/p0` and `p3/p2`:

```
Histogram Analysis:
p1/p0:
   mean = 19.970(23)    sdev = 0.722(16)    skew = 0.127(75)    ex_kurt = -0.01(14)
   median = 19.939(28)    plus = 0.737(35)    minus = 0.654(31)
p3/p2:
   mean = 0.5206(76)    sdev = 0.2396(64)    skew = 0.671(83)    ex_kurt = 0.88(18)
```

```
    median = 0.4894(79)    plus = 0.309(18)    minus = 0.1492(54)
```

The code also displays histograms of the probability distributions, where the dashed lines show the results expected directly from the fit (that is, in the Gaussian approximation):



While the distribution for `p1/p0` is consistent with the fit results (dashed line) and Gaussian, the distribution for `p3/p2` is significantly skewed, with a much longer tail to the right. The final result for `p3/p2` might more accurately be summarized as 0.48 with errors of +0.31 and -0.15, although the Gaussian estimate of 0.48±0.22 would suffice for many applications. The skewed distribution for `p3/p2` is not particularly surprising given the ±50% uncertainty in the denominator `p2`.

2.3 Bayesian Integrals

Bayesian expectation values provide an alternative to least-squares fits. These expectation values are integrals over the fit parameters that are weighted by the probability density function (PDF for the parameters) proportional to `exp(-chi**2/2)`, where `chi**2` includes contributions from both the data and the priors. They can be used to calculate mean values of the parameters, their covariances, and the means and covariances of any function of the parameters. These will agree with the best-fit results of our least-squares fits provided `chi**2` is well approximated by its quadratic expansion in the parameters — that is, insofar as `exp(-chi**2/2)` is well approximated by the Gaussian distribution in the parameters specified by their best-fit means and covariance matrix (from `fit.p`).

Here we use *`lsqfit.BayesIntegrator`* to evaluate Bayesian expectation values. *`lsqfit.BayesIntegrator`* uses the `vegas` module for adaptive multi-dimensional integration to evaluate expectation values. It integrates arbitrary functions of the parameters, multiplied by the probability density function, over the entire parameter space. (Module `vegas` must be installed for *`lsqfit.BayesIntegrator`*.)

To illustrate *`lsqfit.BayesIntegrator`*, we again revisit the analysis in the section on *Correlated Parameters; Gaussian Bayes Factor*. We modify the end of the `main()` function of our original code to evaluate the means and covariances of the parameters, and also their probability histograms, using a Bayesian integral:

```
import matplotlib.pyplot as plt
import numpy as np
import gvar as gv
import lsqfit

def main():
    x, y = make_data()
    prior = make_prior()
    fit = lsqfit.nonlinear_fit(prior=prior, data=(x,y), fcn=fcn)
    print(fit)
```

```
    # Bayesian integrator
    expval = lsqfit.BayesIntegrator(fit)

    # adapt integrator expval to PDF from fit
    neval = 1000
    nitn = 10
    expval(neval=neval, nitn=nitn)

    # <g(p)> gives mean and covariance matrix, and counts for histograms
    hist = [
        gv.PDFHistogram(fit.p[0]), gv.PDFHistogram(fit.p[1]),
        gv.PDFHistogram(fit.p[2]), gv.PDFHistogram(fit.p[3]),
        ]
    def g(p):
        return dict(
            mean=p,
            outer=np.outer(p, p),
            count=[
                hist[0].count(p[0]), hist[1].count(p[1]),
                hist[2].count(p[2]), hist[3].count(p[3]),
                ],
            )

    # evaluate expectation value of g(p)
    results = expval(g, neval=neval, nitn=nitn, adapt=False)

    # analyze results
    print('\nIterations:')
    print(results.summary())
    print('Integration Results:')
    pmean = results['mean']
    pcov =  results['outer'] - np.outer(pmean, pmean)
    print('    mean(p) =', pmean)
    print('    cov(p) =\n', pcov)

    # create GVars from results
    p = gv.gvar(gv.mean(pmean), gv.mean(pcov))
    print('\nBayesian Parameters:')
    print(gv.tabulate(p))

    # show histograms
    print('\nHistogram Statistics:')
    count = results['count']
    for i in range(4):
        # print histogram statistics
        print('p[{}]:'.format(i))
        print(hist[i].analyze(count[i]).stats)
        # make histogram plots
        plt.subplot(2, 2, i + 1)
        plt.xlabel('p[{}]'.format(i))
        hist[i].make_plot(count[i], plot=plt)
        if i % 2 != 0:
            plt.ylabel('')
    plt.show()

def make_data():
    x = np.array([
        4.    ,  2.    ,  1.    ,  0.5   ,  0.25  ,  0.167 ,  0.125 ,
```

```
        0.1   ,  0.0833,  0.0714,  0.0625
        ])
    y = gv.gvar([
        '0.198(14)', '0.216(15)', '0.184(23)', '0.156(44)', '0.099(49)',
        '0.142(40)', '0.108(32)', '0.065(26)', '0.044(22)', '0.041(19)',
        '0.044(16)'
        ])
    return x, y

def make_prior():
    p = gv.gvar(['0(1)', '0(1)', '0(1)', '0(1)'])
    p[1] = 20 * p[0] + gv.gvar('0.0(1)')      # p[1] correlated with p[0]
    return p

def fcn(x, p):
    return (p[0] * (x**2 + p[1] * x)) / (x**2 + x * p[2] + p[3])

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

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

We first call the integrator without a function. This allows it to adapt to the probability density function from the fit without the extra overhead of evaluating a function of the parameters. The integrator uses `nitn=10` iterations of the `vegas` algorithm, with at most `neval=1000` evaluations of the probability density function for each iteration.

We then use the optimized integrator to evaluate the expectation value of function `g(p)` (turning adaptation off with `adapt=False`). The expectation value of `g(p)` is returned in dictionary `results`.

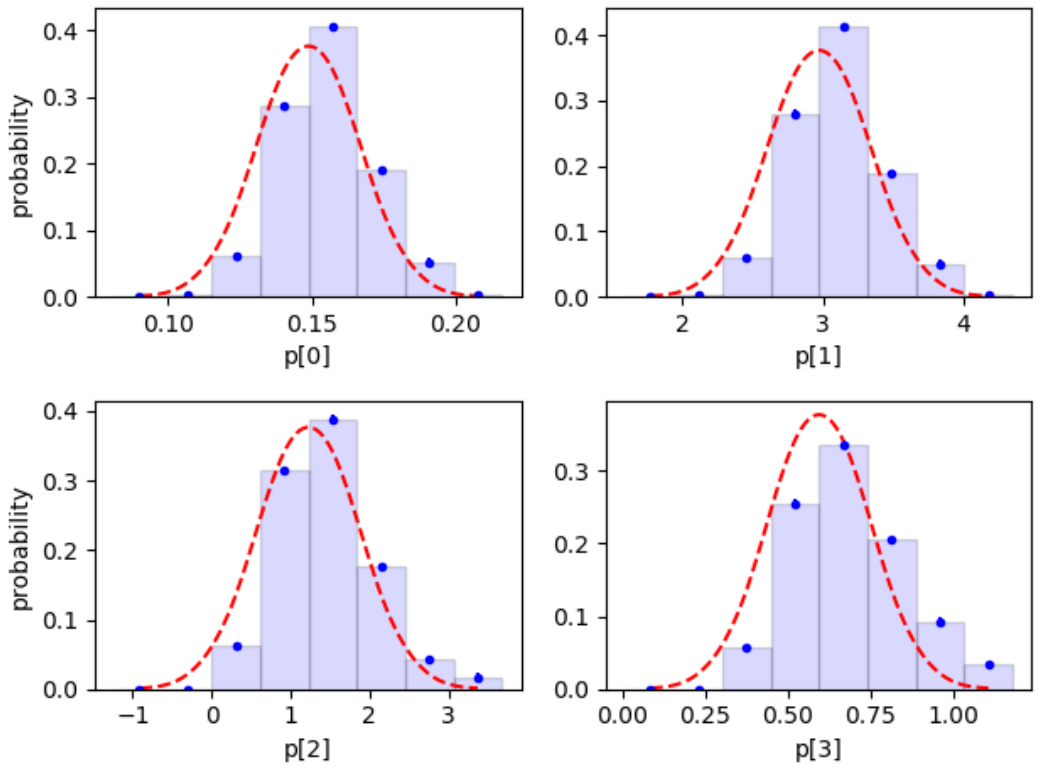
The results from this script are:

```
Least Square Fit:
  chi2/dof [dof] = 0.61 [11]    Q = 0.82    logGBF = 19.129

Parameters:
              0   0.149 (17)      [  0.0 (1.0) ]
              1    2.97 (34)      [     0 (20) ]
              2    1.23 (61)      [  0.0 (1.0) ]  *
              3    0.59 (15)      [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 20/0.1)


Iterations:
itn   integral        average         chi2/dof        Q
-----------------------------------------------------------
  1   1.051(32)       1.051(32)           0.00     1.00
  2   1.015(21)       1.033(19)           0.74     0.94
  3   1.046(24)       1.037(15)           0.74     0.99
  4   1.058(36)       1.042(15)           0.75     1.00
  5   1.009(28)       1.036(13)           0.71     1.00
  6   0.982(24)       1.027(11)           0.72     1.00
  7   1.016(22)       1.025(10)           0.72     1.00
  8   1.031(27)       1.0260(97)          0.70     1.00
```

```
  9   1.122(77)       1.037(12)           0.72     1.00
 10   1.023(24)       1.035(11)           0.73     1.00

Integration Results:
    mean(p) = [0.15514(35) 3.1019(71) 1.453(16) 0.6984(34)]
    cov(p) =
[[0.000252(11) 0.00488(23) 0.00882(57) 0.001436(88)]
 [0.00488(23) 0.1044(47) 0.177(11) 0.0298(18)]
 [0.00882(57) 0.177(11) 0.362(29) 0.0331(37)]
 [0.001436(88) 0.0298(18) 0.0331(37) 0.0345(16)]]

Bayesian Parameters:
  index         value
---------------------
      0    0.155 (16)
      1     3.10 (32)
      2     1.45 (60)
      3     0.70 (19)

Histogram Statistics:
p[0] -
   mean = 0.15518(33)   sdev = 0.01659(29)   skew = 0.119(62)   ex_kurt = 0.52(40)
   median = 0.15471(23)   plus = 0.01610(47)   minus = 0.01553(24)
p[1] -
   mean = 3.1056(68)   sdev = 0.3382(61)   skew = 0.158(48)   ex_kurt = 0.15(10)
   median = 3.0969(46)   plus = 0.3244(97)   minus = 0.3157(48)
p[2] -
   mean = 1.454(16)   sdev = 0.626(20)   skew = 0.505(94)   ex_kurt = 0.37(17)
   median = 1.4082(86)   plus = 0.615(17)   minus = 0.5520(83)
p[3] -
   mean = 0.6717(41)   sdev = 0.1956(46)   skew = -0.39(10)   ex_kurt = 1.54(16)
   median = 0.6730(26)   plus = 0.2013(67)   minus = 0.1537(22)
```

The iterations table shows results from each of the `nitn=10` vegas iterations used to evaluate the expectation values. Estimates for the integral of the probability density function are listed for each iteration. (Results from the integrator are approximate, with error estimates.) These are consistent with each other and with the (more accurate) overall average.

The integration results show that the Bayesian estimates for the means of the parameters are accurate to roughly 1% or better, which is sufficiently accurate here given the size of the standard deviations. Estimates for the covariance matrix elements are less accurate, which is typical. This information is converted into `gvar.GVars` for the parameters and tabulated under "Bayesian Parameters," for comparison with the original fit results — the agreement is pretty good. This is further confirmed by the (posterior) probability distributions for each parameter:



The means are shifted slightly from the fit results and there is modest skewing, but the differences are not great.

As a second example of Bayesian integration, we return briefly to the problem described in *Positive Parameters; Non-Gaussian Priors*: we want the average `a` of noisy data subject the constraint that the average must be positive. The constraint is likely to introduce strong distortions in the probability density function (PDF) given that the fit analysis suggests a value of 0.011 ± 0.013. We plot the actual PDF using the following code, beginning with a fit that uses a flat prior (between 0 and 0.04):

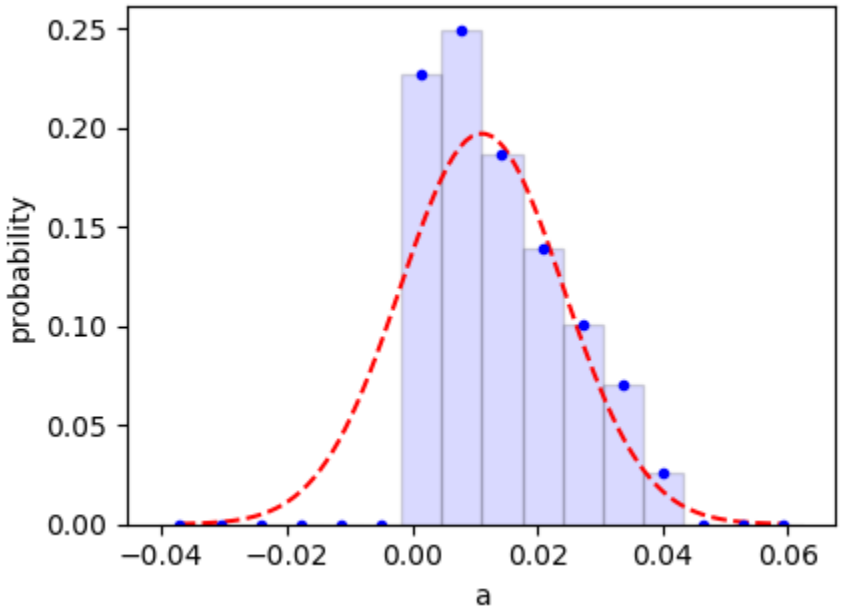
```
import gvar as gv
import lsqfit

# data, prior, and fit function
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)'
    ])

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

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

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

```
# Bayesian analysis: histogram for a
hist = gv.PDFHistogram(a, nbin=16, binwidth=0.5)

def g(p):
    a = (1 + p['50*a-1']) / 50
    return hist.count(a)

expval = lsqfit.BayesIntegrator(fit)
expval(neval=1009, nitn=10)
count = expval(g, neval=1000, nitn=10, adapt=False)

# print out results and show plot
print('\nHistogram Analysis:')
print (hist.analyze(count).stats)

hist.make_plot(count, show=True)
```

The output from this script is

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

Parameters:
  erfinv(50a-1)   -0.42 (68)     [  0.00 (71) ]
-----------------------------------------------
          50a-1   -0.44 (64)     [  0.00 (80) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 15/0.0)

a = 0.011(13)

Histogram Analysis:
   mean = 0.014034(17)   sdev = 0.010534(14)   skew = 0.6080(13)   ex_kurt = -0.
↪6186(24)
   median = 0.011854(20)   plus = 0.014434(36)   minus = 0.008584(17)
```

and the probability distribution for `a` looks like



This distribution is distorted between `a=0` and the mean value, but otherwise is fairly similar to the Gaussian result 0.11 ± 0.13 (dashed line). A more accurate summary of the result for `a` would be 0.12 with an error of +0.14 and -0.09, though again the Gaussian result is not terribly misleading even in this case.

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

We used Bayesian integrals here to deal with non-Gaussian behavior in fit outputs. The case study *Case Study: Outliers and Bayesian Integrals* shows how to use them when the input data is not quite Gaussian.

2.4 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. Method `lsqfit.nonlinear_fit.simulated_fit_iter`` ()` returns an iterator that creates any number of such simulations of the original fit.

A key assumption underlying least-squares fitting is that the fit data `y[i]` are random samples from a distribution whose mean is the fit function `fcn(x, fitp)` evaluated with the best-fit values `fitp` for the parameters. `simulated_fit_iter` iterators generate simulated data by drawing other random samples from the same distribution, assigning them the same covariance matrix as the original data. The simulated data are fit using the same priors and fitter settings as in the original fit, and the results (an *`lsqfit.nonlinear_fit`* object) are returned by the iterator. Fit results from simulated data should agree, within errors, with the original fit results since the simulated data are from the same distribution as the original data. There is a problem with the fitting protocol if this is not the case most of the time.

To illustrate we again examine the fits in the section on *Correlated Parameters; Gaussian Bayes Factor*: we add three fit simulations at the end of the `main()` function:

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

def main():
    x, y = make_data()
    prior = make_prior()
    fit = lsqfit.nonlinear_fit(prior=prior, data=(x,y), fcn=fcn)
    print(40 * '*' + ' real fit')
    print(fit.format(True))

    # 3 simulated fits
    for sfit in fit.simulated_fit_iter(n=3):
        # print simulated fit details
        print(40 * '=' + ' simulation')
        print(sfit.format(True))

        # compare simulated fit results with exact values (pexact=fit.pmean)
        diff = sfit.p - sfit.pexact
        print('\nsfit.p - pexact =', diff)
        print(gv.fmt_chi2(gv.chi2(diff)))
        print

def make_data():
    x = np.array([
        4.    ,  2.    ,  1.    ,  0.5   ,  0.25  ,  0.167 ,  0.125 ,
        0.1   ,  0.0833,  0.0714,  0.0625
        ])
    y = gv.gvar([
        '0.198(14)', '0.216(15)', '0.184(23)', '0.156(44)', '0.099(49)',
        '0.142(40)', '0.108(32)', '0.065(26)', '0.044(22)', '0.041(19)',
        '0.044(16)'
        ])
    return x, y

def make_prior():
    p = gv.gvar(['0(1)', '0(1)', '0(1)', '0(1)'])
    p[1] = 20 * p[0] + gv.gvar('0.0(1)')        # p[1] correlated with p[0]
    return p

def fcn(x, p):
    return (p[0] * (x**2 + p[1] * x)) / (x**2 + x * p[2] + p[3])

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

This code produces the following output, showing how the input data fluctuate from simulation to simulation:

```
**************************************** real fit
Least Square Fit:
  chi2/dof [dof] = 0.61 [11]    Q = 0.82    logGBF = 19.129

Parameters:
              0   0.149 (17)      [  0.0 (1.0) ]
              1    2.97 (34)      [     0 (20) ]
              2    1.23 (61)      [  0.0 (1.0) ]  *
              3    0.59 (15)      [  0.0 (1.0) ]
```

```
Fit:
     x[k]           y[k]       f(x[k],p)
-------------------------------------
        4    0.198 (14)      0.193 (11)
        2    0.216 (15)      0.210 (10)
        1    0.184 (23)      0.209 (15)  *
      0.5    0.156 (44)      0.177 (15)
     0.25    0.099 (49)      0.124 (13)
    0.167    0.142 (40)      0.094 (12)  *
    0.125    0.108 (32)      0.075 (11)  *
      0.1    0.065 (26)     0.0629 (96)
   0.0833    0.044 (22)     0.0538 (87)
   0.0714    0.041 (19)     0.0471 (79)
   0.0625    0.044 (16)     0.0418 (72)

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 20/0.1)

======================================== simulation
Least Square Fit:
  chi2/dof [dof] = 1.2 [11]    Q = 0.27    logGBF = 15.278

Parameters:
              0   0.134 (14)      [  0.0 (1.0) ]
              1    2.68 (29)      [     0 (20) ]
              2    0.68 (47)      [  0.0 (1.0) ]
              3    0.54 (12)      [  0.0 (1.0) ]

Fit:
     x[k]           y[k]       f(x[k],p)
-------------------------------------
        4    0.200 (14)      0.186 (12)
        2    0.192 (15)      0.212 (10)  *
        1    0.242 (23)      0.221 (16)
      0.5    0.163 (44)      0.187 (18)
     0.25    0.089 (49)      0.126 (15)
    0.167    0.130 (40)      0.093 (13)
    0.125    0.103 (32)      0.073 (11)
      0.1    0.046 (26)     0.0599 (96)
   0.0833    0.054 (22)     0.0508 (85)
   0.0714    0.004 (19)     0.0441 (77)  **
   0.0625    0.060 (16)     0.0389 (70)  *

Settings:
  svdcut/n = None/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 7/0.0)


sfit.p - pexact = [-0.015(14) -0.29(29) -0.54(47) -0.05(12)]
chi2/dof = 0.34 [4]    Q = 0.85

======================================== simulation
Least Square Fit:
  chi2/dof [dof] = 1.1 [11]    Q = 0.38    logGBF = 17.048

Parameters:
              0   0.156 (18)      [  0.0 (1.0) ]
              1    3.12 (36)      [     0 (20) ]
              2    1.35 (66)      [  0.0 (1.0) ]  *
```

```
              3    0.77 (20)      [  0.0 (1.0) ]

Fit:
     x[k]           y[k]       f(x[k],p)
---------------------------------------
        4    0.207 (14)      0.201 (11)
        2    0.224 (15)      0.214 (10)
        1    0.163 (23)      0.206 (14)  *
      0.5    0.162 (44)      0.167 (15)
     0.25    0.124 (49)      0.113 (14)
    0.167    0.111 (40)      0.084 (12)
    0.125    0.085 (32)      0.066 (11)
      0.1    0.097 (26)     0.0550 (93)  *
   0.0833    0.020 (22)     0.0469 (83)  *
   0.0714    0.043 (19)     0.0409 (75)
   0.0625    0.031 (16)     0.0362 (68)

Settings:
  svdcut/n = None/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 23/0.0)


sfit.p - pexact = [0.008(18) 0.15(36) 0.13(66) 0.18(20)]
chi2/dof = 0.22 [4]    Q = 0.93

======================================= simulation
Least Square Fit:
  chi2/dof [dof] = 0.76 [11]    Q = 0.68    logGBF = 17.709

Parameters:
              0   0.138 (14)      [  0.0 (1.0) ]
              1    2.77 (29)      [     0 (20) ]
              2    0.72 (46)      [  0.0 (1.0) ]
              3    0.53 (11)      [  0.0 (1.0) ]

Fit:
     x[k]           y[k]       f(x[k],p)
---------------------------------------
        4    0.196 (14)      0.193 (12)
        2    0.218 (15)      0.221 (10)
        1    0.240 (23)      0.231 (16)
      0.5    0.157 (44)      0.198 (18)
     0.25    0.157 (49)      0.134 (14)
    0.167    0.022 (40)      0.099 (12)  *
    0.125    0.070 (32)      0.078 (11)
      0.1    0.090 (26)     0.0644 (96)
   0.0833    0.045 (22)     0.0547 (86)
   0.0714    0.053 (19)     0.0475 (78)
   0.0625    0.059 (16)     0.0420 (70)  *

Settings:
  svdcut/n = None/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 10/0.0)


sfit.p - pexact = [-0.010(14) -0.21(29) -0.51(46) -0.06(11)]
chi2/dof = 0.77 [4]    Q = 0.54
```

The parameters `sfit.p` produced by the simulated fits agree well with the original fit parameters `pexact=fit.`

pmean, with good fits in each case. We calculate the chi**2 for the difference sfit.p - pexact in each case; good chi**2 values validate the parameter values, standard deviations, and correlations.

CHAPTER

THREE

CASE STUDY: SIMPLE EXTRAPOLATION

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

3.1 The Problem

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

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

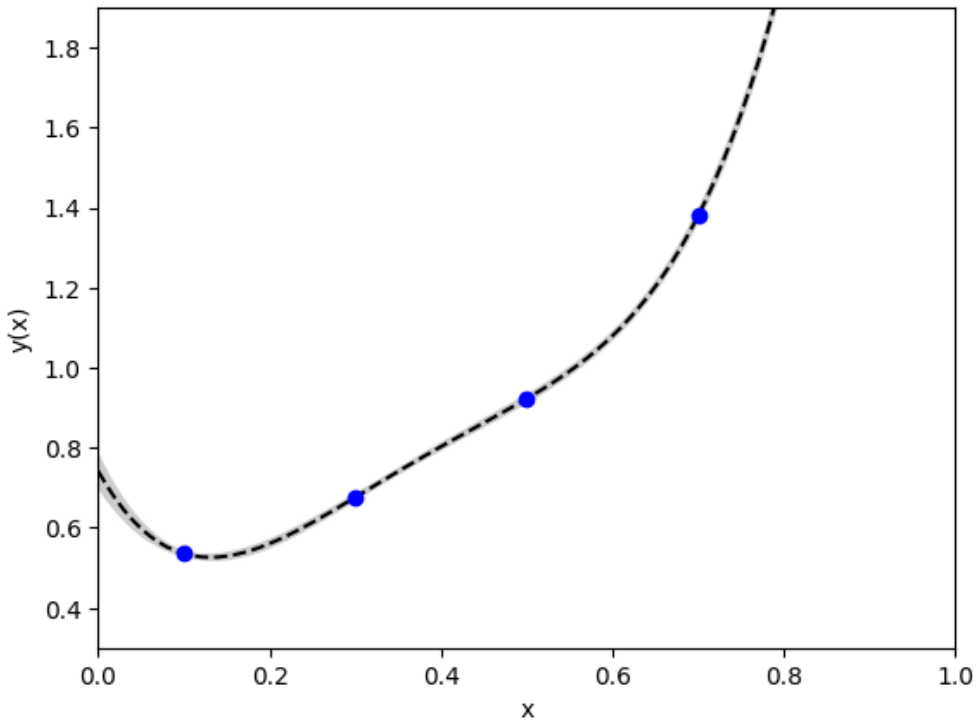
We know that `y(x)` has a Taylor expansion in `x`:

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

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

3.2 A Bad Solution

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



This fit was generated using the following code:

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

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

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

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

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

```
Least Square Fit (no prior):
  chi2/dof [dof] = 4.5e-24 [0]    Q = 0    logGBF = None

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

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

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)     (itns/time = 11/0.0)
```

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

3.3 A Better Solution — Priors

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

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

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

with `N=90`.

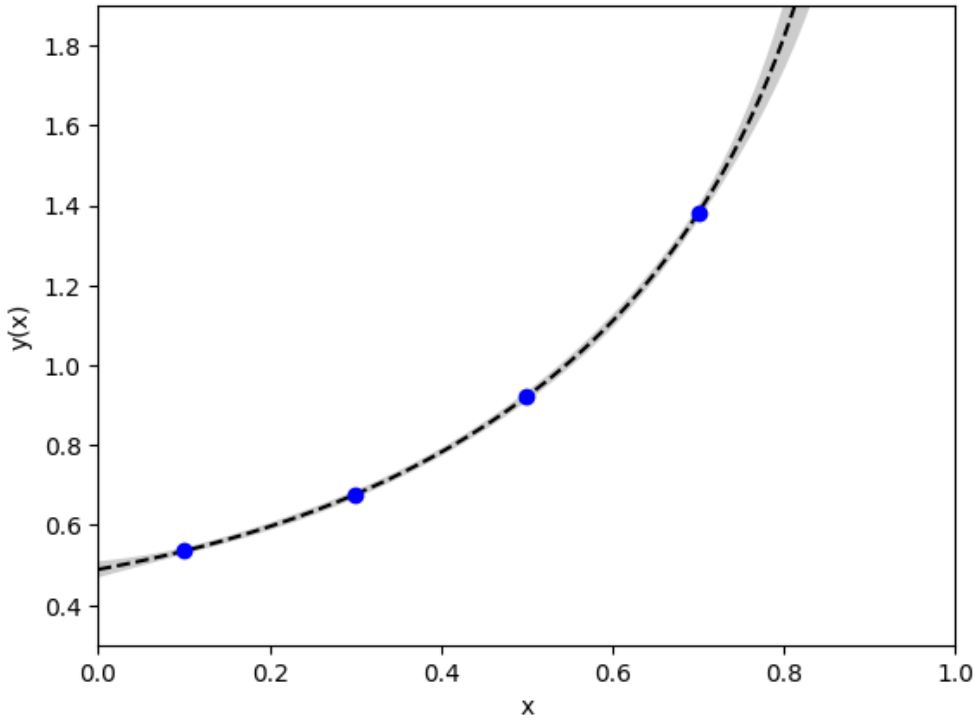
Fitting a 91-parameter formula to five pieces of data is also impossible. Here, however, we have extra (*prior*) information: each coefficient is order one, which we make specific by saying that they equal 0 ± 1. We include these *a priori* estimates for the parameters as extra data that must be fit,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

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

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

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

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

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

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

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



The fit code output is:

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

Parameters:
              0      0.489 (17)      [  0.0 (1.0) ]
              1       0.40 (20)      [  0.0 (1.0) ]
              2       0.60 (64)      [  0.0 (1.0) ]
              3       0.44 (80)      [  0.0 (1.0) ]
              4       0.28 (87)      [  0.0 (1.0) ]
              5       0.19 (87)      [  0.0 (1.0) ]
              6       0.16 (90)      [  0.0 (1.0) ]
              7       0.16 (93)      [  0.0 (1.0) ]
              8       0.17 (95)      [  0.0 (1.0) ]
              9       0.18 (96)      [  0.0 (1.0) ]
             10       0.19 (97)      [  0.0 (1.0) ]
             11       0.19 (97)      [  0.0 (1.0) ]
             12       0.19 (97)      [  0.0 (1.0) ]
             13       0.19 (97)      [  0.0 (1.0) ]
             14       0.18 (97)      [  0.0 (1.0) ]
             15       0.18 (97)      [  0.0 (1.0) ]
             16       0.17 (97)      [  0.0 (1.0) ]
             17       0.16 (98)      [  0.0 (1.0) ]
```

```
 18       0.16 (98)       [  0.0 (1.0) ]
 19       0.15 (98)       [  0.0 (1.0) ]
 20       0.14 (98)       [  0.0 (1.0) ]
 21       0.14 (98)       [  0.0 (1.0) ]
 22       0.13 (98)       [  0.0 (1.0) ]
 23       0.12 (99)       [  0.0 (1.0) ]
 24       0.12 (99)       [  0.0 (1.0) ]
 25       0.11 (99)       [  0.0 (1.0) ]
 26       0.11 (99)       [  0.0 (1.0) ]
 27       0.10 (99)       [  0.0 (1.0) ]
 28       0.10 (99)       [  0.0 (1.0) ]
 29       0.09 (99)       [  0.0 (1.0) ]
 30       0.09 (99)       [  0.0 (1.0) ]
 31       0.08 (99)       [  0.0 (1.0) ]
 32       0.08 (99)       [  0.0 (1.0) ]
 33       0.07 (99)       [  0.0 (1.0) ]
 34     0.07 (1.00)       [  0.0 (1.0) ]
 35     0.07 (1.00)       [  0.0 (1.0) ]
 36     0.06 (1.00)       [  0.0 (1.0) ]
 37     0.06 (1.00)       [  0.0 (1.0) ]
 38     0.06 (1.00)       [  0.0 (1.0) ]
 39     0.06 (1.00)       [  0.0 (1.0) ]
 40     0.05 (1.00)       [  0.0 (1.0) ]
 41     0.05 (1.00)       [  0.0 (1.0) ]
 42     0.05 (1.00)       [  0.0 (1.0) ]
 43     0.04 (1.00)       [  0.0 (1.0) ]
 44     0.04 (1.00)       [  0.0 (1.0) ]
 45     0.04 (1.00)       [  0.0 (1.0) ]
 46     0.04 (1.00)       [  0.0 (1.0) ]
 47     0.04 (1.00)       [  0.0 (1.0) ]
 48     0.03 (1.00)       [  0.0 (1.0) ]
 49     0.03 (1.00)       [  0.0 (1.0) ]
 50     0.03 (1.00)       [  0.0 (1.0) ]
 51     0.03 (1.00)       [  0.0 (1.0) ]
 52     0.03 (1.00)       [  0.0 (1.0) ]
 53     0.03 (1.00)       [  0.0 (1.0) ]
 54     0.03 (1.00)       [  0.0 (1.0) ]
 55     0.02 (1.00)       [  0.0 (1.0) ]
 56     0.02 (1.00)       [  0.0 (1.0) ]
 57     0.02 (1.00)       [  0.0 (1.0) ]
 58     0.02 (1.00)       [  0.0 (1.0) ]
 59     0.02 (1.00)       [  0.0 (1.0) ]
 60     0.02 (1.00)       [  0.0 (1.0) ]
 61     0.02 (1.00)       [  0.0 (1.0) ]
 62     0.02 (1.00)       [  0.0 (1.0) ]
 63     0.02 (1.00)       [  0.0 (1.0) ]
 64     0.02 (1.00)       [  0.0 (1.0) ]
 65     0.01 (1.00)       [  0.0 (1.0) ]
 66     0.01 (1.00)       [  0.0 (1.0) ]
 67     0.01 (1.00)       [  0.0 (1.0) ]
 68     0.01 (1.00)       [  0.0 (1.0) ]
 69     0.01 (1.00)       [  0.0 (1.0) ]
 70     0.01 (1.00)       [  0.0 (1.0) ]
 71     0.01 (1.00)       [  0.0 (1.0) ]
 72     0.01 (1.00)       [  0.0 (1.0) ]
 73     0.01 (1.00)       [  0.0 (1.0) ]
 74   0.009 (1.000)       [  0.0 (1.0) ]
 75   0.009 (1.000)       [  0.0 (1.0) ]
```

```
              76   0.008 (1.000)       [  0.0 (1.0) ]
              77   0.008 (1.000)       [  0.0 (1.0) ]
              78   0.007 (1.000)       [  0.0 (1.0) ]
              79   0.007 (1.000)       [  0.0 (1.0) ]
              80   0.007 (1.000)       [  0.0 (1.0) ]
              81   0.006 (1.000)       [  0.0 (1.0) ]
              82   0.006 (1.000)       [  0.0 (1.0) ]
              83   0.006 (1.000)       [  0.0 (1.0) ]
              84   0.005 (1.000)       [  0.0 (1.0) ]
              85   0.005 (1.000)       [  0.0 (1.0) ]
              86   0.005 (1.000)       [  0.0 (1.0) ]
              87   0.005 (1.000)       [  0.0 (1.0) ]
              88   0.004 (1.000)       [  0.0 (1.0) ]
              89   0.004 (1.000)       [  0.0 (1.0) ]
              90   0.004 (1.000)       [  0.0 (1.0) ]

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

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)     (itns/time = 8/0.1)
```

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

3.4 Bayes Factors

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

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

Parameters:
              0      0.490 (33)       [  0.0 (3.0) ]
              1       0.38 (48)       [  0.0 (3.0) ]
              2        0.6 (1.8)      [  0.0 (3.0) ]
              3        0.5 (2.4)      [  0.0 (3.0) ]
              4        0.3 (2.6)      [  0.0 (3.0) ]
              5        0.2 (2.6)      [  0.0 (3.0) ]
              6        0.1 (2.7)      [  0.0 (3.0) ]
              7        0.1 (2.8)      [  0.0 (3.0) ]
              8        0.2 (2.8)      [  0.0 (3.0) ]
              9        0.2 (2.9)      [  0.0 (3.0) ]
             10        0.2 (2.9)      [  0.0 (3.0) ]
```

```
 11       0.2 (2.9)       [  0.0 (3.0) ]
 12       0.2 (2.9)       [  0.0 (3.0) ]
 13       0.2 (2.9)       [  0.0 (3.0) ]
 14       0.2 (2.9)       [  0.0 (3.0) ]
 15       0.2 (2.9)       [  0.0 (3.0) ]
 16       0.2 (2.9)       [  0.0 (3.0) ]
 17       0.2 (2.9)       [  0.0 (3.0) ]
 18       0.2 (2.9)       [  0.0 (3.0) ]
 19       0.2 (2.9)       [  0.0 (3.0) ]
 20       0.1 (2.9)       [  0.0 (3.0) ]
 21       0.1 (2.9)       [  0.0 (3.0) ]
 22       0.1 (2.9)       [  0.0 (3.0) ]
 23       0.1 (3.0)       [  0.0 (3.0) ]
 24       0.1 (3.0)       [  0.0 (3.0) ]
 25       0.1 (3.0)       [  0.0 (3.0) ]
 26       0.1 (3.0)       [  0.0 (3.0) ]
 27       0.1 (3.0)       [  0.0 (3.0) ]
 28       0.1 (3.0)       [  0.0 (3.0) ]
 29       0.1 (3.0)       [  0.0 (3.0) ]
 30     0.09 (2.98)       [  0.0 (3.0) ]
 31     0.09 (2.98)       [  0.0 (3.0) ]
 32     0.08 (2.98)       [  0.0 (3.0) ]
 33     0.08 (2.98)       [  0.0 (3.0) ]
 34     0.07 (2.99)       [  0.0 (3.0) ]
 35     0.07 (2.99)       [  0.0 (3.0) ]
 36     0.07 (2.99)       [  0.0 (3.0) ]
 37     0.06 (2.99)       [  0.0 (3.0) ]
 38     0.06 (2.99)       [  0.0 (3.0) ]
 39     0.06 (2.99)       [  0.0 (3.0) ]
 40     0.05 (2.99)       [  0.0 (3.0) ]
 41     0.05 (2.99)       [  0.0 (3.0) ]
 42     0.05 (2.99)       [  0.0 (3.0) ]
 43     0.05 (2.99)       [  0.0 (3.0) ]
 44     0.04 (2.99)       [  0.0 (3.0) ]
 45     0.04 (3.00)       [  0.0 (3.0) ]
 46     0.04 (3.00)       [  0.0 (3.0) ]
 47     0.04 (3.00)       [  0.0 (3.0) ]
 48     0.04 (3.00)       [  0.0 (3.0) ]
 49     0.03 (3.00)       [  0.0 (3.0) ]
 50     0.03 (3.00)       [  0.0 (3.0) ]
 51     0.03 (3.00)       [  0.0 (3.0) ]
 52     0.03 (3.00)       [  0.0 (3.0) ]
 53     0.03 (3.00)       [  0.0 (3.0) ]
 54     0.03 (3.00)       [  0.0 (3.0) ]
 55     0.03 (3.00)       [  0.0 (3.0) ]
 56     0.02 (3.00)       [  0.0 (3.0) ]
 57     0.02 (3.00)       [  0.0 (3.0) ]
 58     0.02 (3.00)       [  0.0 (3.0) ]
 59     0.02 (3.00)       [  0.0 (3.0) ]
 60     0.02 (3.00)       [  0.0 (3.0) ]
 61     0.02 (3.00)       [  0.0 (3.0) ]
 62     0.02 (3.00)       [  0.0 (3.0) ]
 63     0.02 (3.00)       [  0.0 (3.0) ]
 64     0.02 (3.00)       [  0.0 (3.0) ]
 65     0.02 (3.00)       [  0.0 (3.0) ]
 66     0.01 (3.00)       [  0.0 (3.0) ]
 67     0.01 (3.00)       [  0.0 (3.0) ]
 68     0.01 (3.00)       [  0.0 (3.0) ]
```

```
              69     0.01 (3.00)       [  0.0 (3.0) ]
              70     0.01 (3.00)       [  0.0 (3.0) ]
              71     0.01 (3.00)       [  0.0 (3.0) ]
              72     0.01 (3.00)       [  0.0 (3.0) ]
              73     0.01 (3.00)       [  0.0 (3.0) ]
              74    0.009 (3.000)      [  0.0 (3.0) ]
              75    0.009 (3.000)      [  0.0 (3.0) ]
              76    0.009 (3.000)      [  0.0 (3.0) ]
              77    0.008 (3.000)      [  0.0 (3.0) ]
              78    0.008 (3.000)      [  0.0 (3.0) ]
              79    0.007 (3.000)      [  0.0 (3.0) ]
              80    0.007 (3.000)      [  0.0 (3.0) ]
              81    0.007 (3.000)      [  0.0 (3.0) ]
              82    0.006 (3.000)      [  0.0 (3.0) ]
              83    0.006 (3.000)      [  0.0 (3.0) ]
              84    0.006 (3.000)      [  0.0 (3.0) ]
              85    0.005 (3.000)      [  0.0 (3.0) ]
              86    0.005 (3.000)      [  0.0 (3.0) ]
              87    0.005 (3.000)      [  0.0 (3.0) ]
              88    0.005 (3.000)      [  0.0 (3.0) ]
              89    0.004 (3.000)      [  0.0 (3.0) ]
              90    0.004 (3.000)      [  0.0 (3.0) ]

Fit:
     x[k]           y[k]       f(x[k],p)
-----------------------------------------
      0.1    0.5351 (54)     0.5351 (54)
      0.3    0.6762 (67)     0.6763 (67)
      0.5    0.9227 (91)     0.9226 (91)
      0.7     1.380 (13)      1.380 (13)
     0.95     4.014 (40)      4.014 (40)

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08,1e-10*,1e-10)    (itns/time = 9/0.1)
```

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

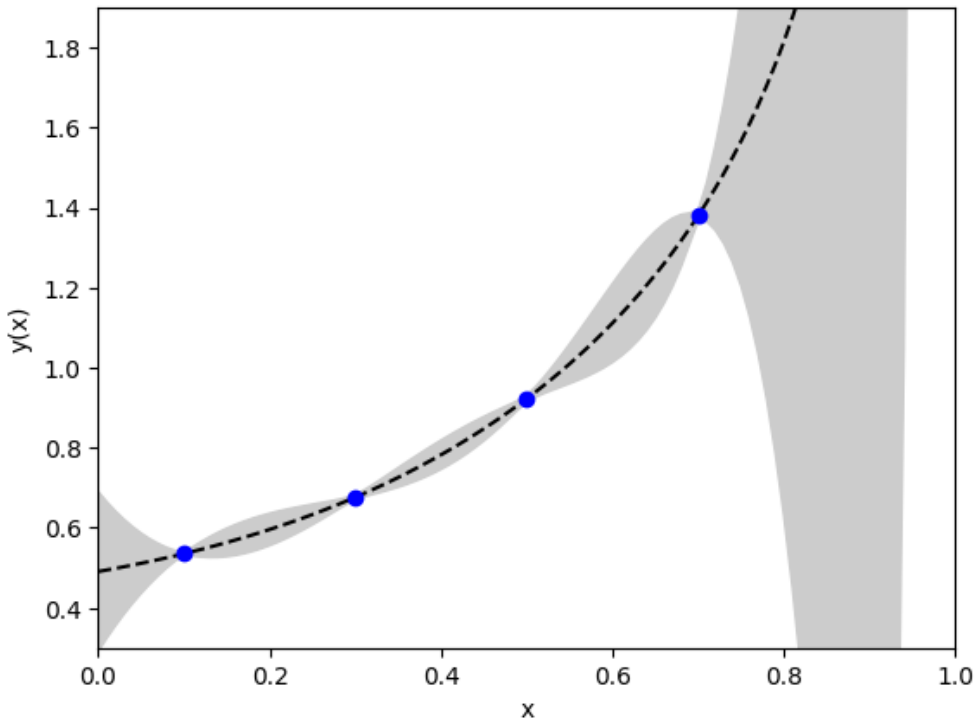
Narrower priors, `prior = gv.gvar(91 * ['0.0(3)'])`, give a poor fit, and also a less optimal `logGBF`:

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

Parameters:
               0    0.484 (11)      [  0.00 (30) ]  *
               1    0.454 (98)      [  0.00 (30) ]  *
               2     0.50 (23)      [  0.00 (30) ]  *
               3     0.40 (25)      [  0.00 (30) ]  *
               4     0.31 (26)      [  0.00 (30) ]  *
               5     0.26 (27)      [  0.00 (30) ]
               6     0.23 (28)      [  0.00 (30) ]
```

```
 7     0.21 (29)      [  0.00 (30) ]
 8     0.21 (29)      [  0.00 (30) ]
 9     0.20 (29)      [  0.00 (30) ]
10     0.19 (29)      [  0.00 (30) ]
11     0.19 (29)      [  0.00 (30) ]
12     0.18 (29)      [  0.00 (30) ]
13     0.17 (29)      [  0.00 (30) ]
14     0.17 (29)      [  0.00 (30) ]
15     0.16 (29)      [  0.00 (30) ]
16     0.15 (29)      [  0.00 (30) ]
17     0.15 (29)      [  0.00 (30) ]
18     0.14 (29)      [  0.00 (30) ]
19     0.13 (29)      [  0.00 (30) ]
20     0.13 (29)      [  0.00 (30) ]
21     0.12 (30)      [  0.00 (30) ]
22     0.11 (30)      [  0.00 (30) ]
23     0.11 (30)      [  0.00 (30) ]
24     0.10 (30)      [  0.00 (30) ]
25     0.10 (30)      [  0.00 (30) ]
26     0.09 (30)      [  0.00 (30) ]
27     0.09 (30)      [  0.00 (30) ]
28     0.08 (30)      [  0.00 (30) ]
29     0.08 (30)      [  0.00 (30) ]
30     0.08 (30)      [  0.00 (30) ]
31     0.07 (30)      [  0.00 (30) ]
32     0.07 (30)      [  0.00 (30) ]
33     0.07 (30)      [  0.00 (30) ]
34     0.06 (30)      [  0.00 (30) ]
35     0.06 (30)      [  0.00 (30) ]
36     0.06 (30)      [  0.00 (30) ]
37     0.05 (30)      [  0.00 (30) ]
38     0.05 (30)      [  0.00 (30) ]
39     0.05 (30)      [  0.00 (30) ]
40     0.05 (30)      [  0.00 (30) ]
41     0.04 (30)      [  0.00 (30) ]
42     0.04 (30)      [  0.00 (30) ]
43     0.04 (30)      [  0.00 (30) ]
44     0.04 (30)      [  0.00 (30) ]
45     0.04 (30)      [  0.00 (30) ]
46     0.03 (30)      [  0.00 (30) ]
47     0.03 (30)      [  0.00 (30) ]
48     0.03 (30)      [  0.00 (30) ]
49     0.03 (30)      [  0.00 (30) ]
50     0.03 (30)      [  0.00 (30) ]
51     0.03 (30)      [  0.00 (30) ]
52     0.02 (30)      [  0.00 (30) ]
53     0.02 (30)      [  0.00 (30) ]
54     0.02 (30)      [  0.00 (30) ]
55     0.02 (30)      [  0.00 (30) ]
56     0.02 (30)      [  0.00 (30) ]
57     0.02 (30)      [  0.00 (30) ]
58     0.02 (30)      [  0.00 (30) ]
59     0.02 (30)      [  0.00 (30) ]
60     0.02 (30)      [  0.00 (30) ]
61     0.02 (30)      [  0.00 (30) ]
62     0.01 (30)      [  0.00 (30) ]
63     0.01 (30)      [  0.00 (30) ]
64     0.01 (30)      [  0.00 (30) ]
```

```
              65     0.01 (30)      [  0.00 (30) ]
              66     0.01 (30)      [  0.00 (30) ]
              67     0.01 (30)      [  0.00 (30) ]
              68     0.01 (30)      [  0.00 (30) ]
              69     0.01 (30)      [  0.00 (30) ]
              70     0.01 (30)      [  0.00 (30) ]
              71    0.009 (300)     [  0.00 (30) ]
              72    0.009 (300)     [  0.00 (30) ]
              73    0.008 (300)     [  0.00 (30) ]
              74    0.008 (300)     [  0.00 (30) ]
              75    0.008 (300)     [  0.00 (30) ]
              76    0.007 (300)     [  0.00 (30) ]
              77    0.007 (300)     [  0.00 (30) ]
              78    0.006 (300)     [  0.00 (30) ]
              79    0.006 (300)     [  0.00 (30) ]
              80    0.006 (300)     [  0.00 (30) ]
              81    0.006 (300)     [  0.00 (30) ]
              82    0.005 (300)     [  0.00 (30) ]
              83    0.005 (300)     [  0.00 (30) ]
              84    0.005 (300)     [  0.00 (30) ]
              85    0.005 (300)     [  0.00 (30) ]
              86    0.004 (300)     [  0.00 (30) ]
              87    0.004 (300)     [  0.00 (30) ]
              88    0.004 (300)     [  0.00 (30) ]
              89    0.004 (300)     [  0.00 (30) ]
              90    0.003 (300)     [  0.00 (30) ]

Fit:
     x[k]           y[k]       f(x[k],p)
-----------------------------------------
      0.1    0.5351 (54)     0.5344 (53)
      0.3    0.6762 (67)     0.6787 (56)
      0.5    0.9227 (91)     0.9191 (75)
      0.7     1.380 (13)      1.382 (13)
     0.95     4.014 (40)      4.008 (40)

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)     (itns/time = 6/0.1)
```

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



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

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

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

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

which prints the following table:

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

The table shows that the final 3.5% error comes from a 2.7% error due to uncertainties in `y` and a 2.2% error from

uncertainties in the prior (added in quadrature).

3.5 Another Solution — Marginalization

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

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

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

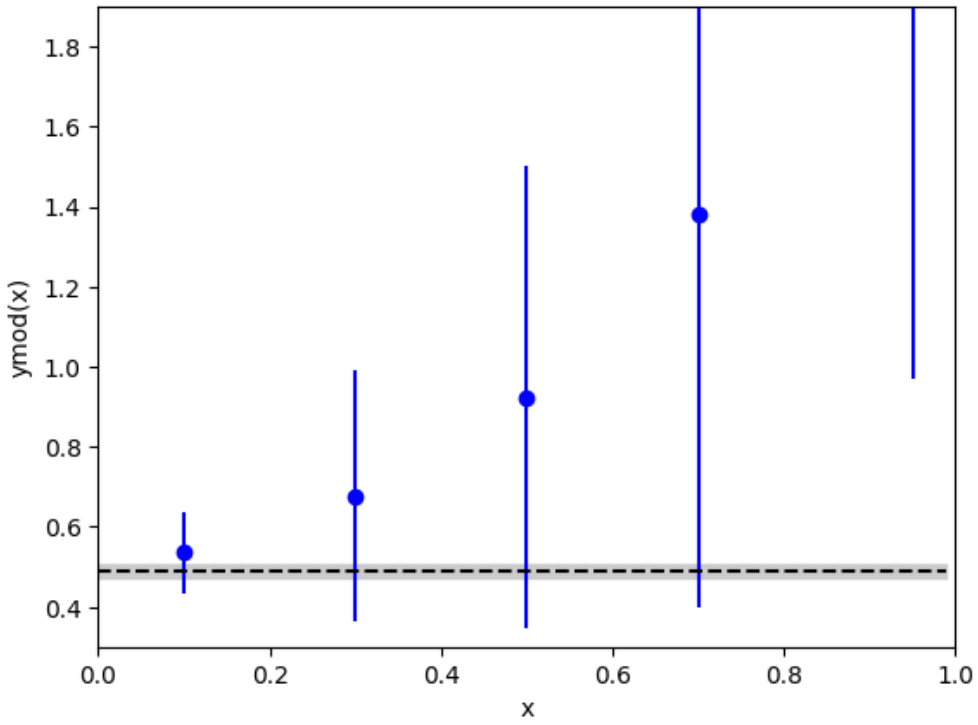
```
import numpy as np
import gvar as gv
import lsqfit

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

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

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

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

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

Running this code give:

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

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

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

Settings:
```

```
  svdcut/n = 1e-12/0     tol = (1e-08*,1e-10,1e-10)     (itns/time = 4/0.0)
```

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



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

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

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

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

CHAPTER

FOUR

CASE STUDY: PENDULUM

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

4.1 The Problem

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

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

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

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

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

4.2 Pendulum Dynamics

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

```
import numpy as np
import gvar as gv

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

    Input parameters are:
```

```
        g/l .... where g is acceleration due to gravity and l the length
        tol .... precision of numerical integration of ODE
    """
    def __init__(self, g_l, tol=1e-4):
        self.g_l = g_l
        self.odeint = gv.ode.Integrator(deriv=self.deriv, tol=tol)

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

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

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

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

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

4.3 Two Types of Input Data

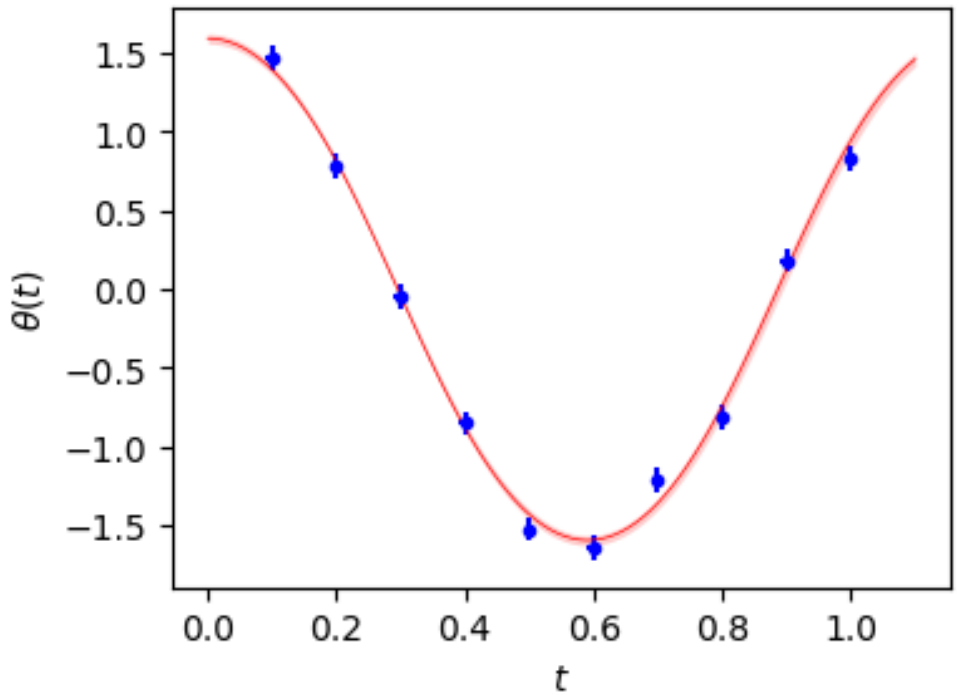
There are two ways to include data in a fit: either as regular data, or as fit parameters with priors. In general dependent variables are treated as regular data, and independent variables with errors are treated as fit parameters, with priors. Here the dependent variable is `theta(t)` and the independent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

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

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

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

```
        '-1.523(79)', '-1.647(79)', '-1.216(79)', '-0.810(79)',
        '0.185(79)', '0.832(79)'
        ])

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

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

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

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

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



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

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

```
Parameters:
            g/l    39.82 (87)      [    40 (20) ]
       theta(0)    1.595 (32)      [ 1.571 (50) ]
            t 0   0.0960 (91)      [ 0.100 (10) ]
              1   0.2014 (74)      [ 0.200 (10) ]
              2   0.3003 (67)      [ 0.300 (10) ]
              3   0.3982 (76)      [ 0.400 (10) ]
              4   0.5043 (93)      [ 0.500 (10) ]
              5    0.600 (10)      [ 0.600 (10) ]
              6   0.7079 (89)      [ 0.700 (10) ]
              7   0.7958 (79)      [ 0.800 (10) ]
              8   0.9039 (78)      [ 0.900 (10) ]
              9   0.9929 (83)      [ 1.000 (10) ]

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

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)     (itns/time = 7/0.1)
```

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

CHAPTER

FIVE

CASE STUDY: OUTLIERS AND BAYESIAN INTEGRALS

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

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

5.1 The Problem

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

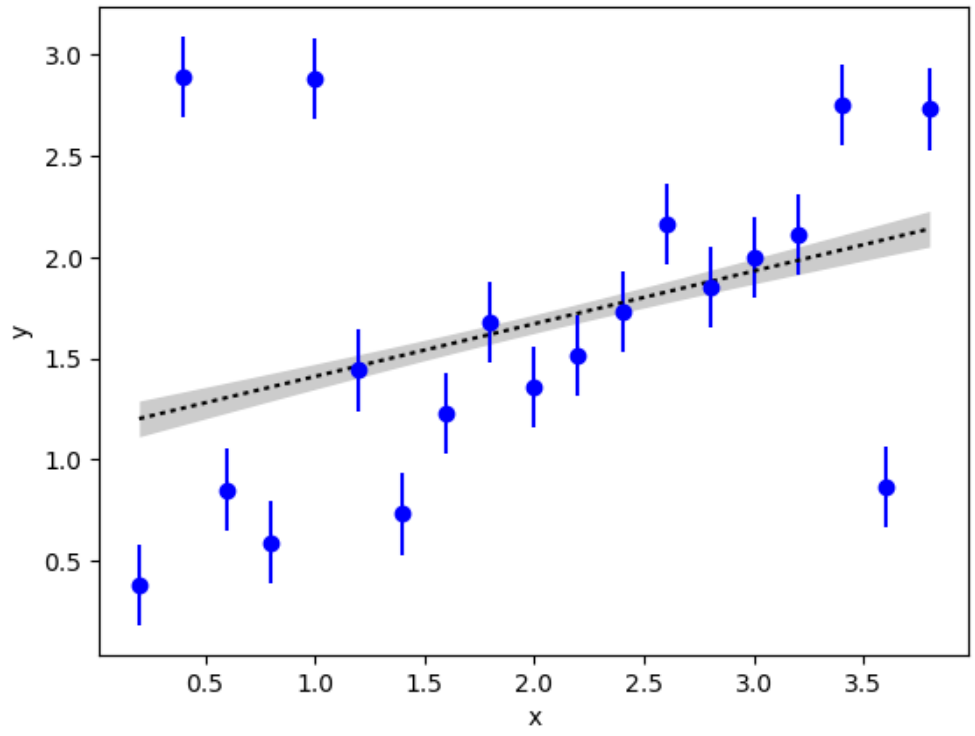
```
import matplotlib.pyplot as plt
import numpy as np

import gvar as gv
import lsqfit

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

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

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

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

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

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


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

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

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

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

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08,1e-10*,1e-10)    (itns/time = 4/0.1)
```

The problem is evident if we plot the data:



At least three of the data points are outliers: they disagree with other nearby points by several standard deviations. These outliers have a big impact on the fit (dashed line, with the gray band showing the ±1-sigma region). In particular they pull the `x=0` intercept (`fit.p['c'][0]`) up above one, while the rest of the data suggest an intercept of 0.5 or less.

5.2 A Solution

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

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

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

import gvar as gv
import lsqfit

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

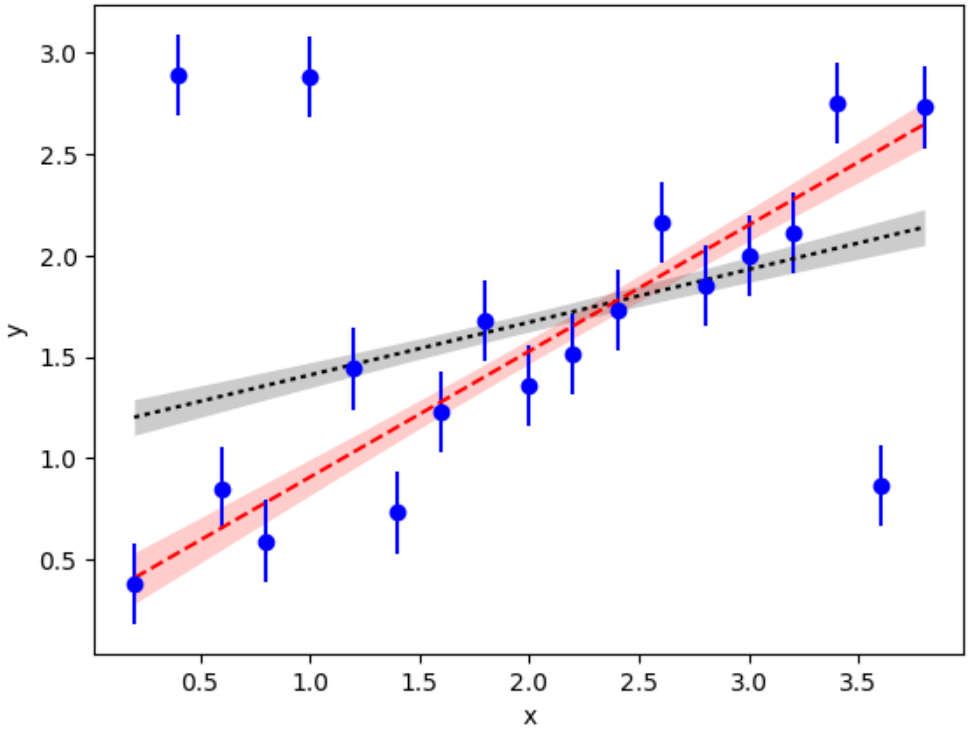
The results from this code are as follows:

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

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

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08,1e-10*,1e-10)    (itns/time = 4/0.0)

itn   integral        average         chi2/dof        Q
```

```
-------------------------------------------------------
  1   6.82(11)e-11    6.82(11)e-11        0.00     1.00
  2   7.04(11)e-11    6.930(78)e-11       1.10     0.36
  3   6.775(76)e-11   6.878(58)e-11       0.97     0.49
  4   6.651(97)e-11   6.821(50)e-11       1.04     0.40
  5   6.74(10)e-11    6.806(45)e-11       0.95     0.55
  6   6.740(79)e-11   6.795(39)e-11       0.88     0.69
  7   6.763(87)e-11   6.790(36)e-11       0.93     0.63
  8   7.085(92)e-11   6.827(34)e-11       0.96     0.56
  9   6.873(68)e-11   6.832(31)e-11       0.95     0.59
 10   6.853(75)e-11   6.834(29)e-11       0.95     0.61
 11   6.79(11)e-11    6.830(28)e-11       0.89     0.76
 12   6.833(94)e-11   6.830(27)e-11       0.92     0.71
 13   6.806(81)e-11   6.828(26)e-11       0.93     0.67
 14   6.67(10)e-11    6.817(25)e-11       0.94     0.66
 15   6.725(93)e-11   6.811(24)e-11       0.90     0.77

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

w = 0.26(11)

logBF = -23.4099(35)
```

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

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



Note, from the correlation matrix, that the intercept and slope are anti-correlated, as one might guess for this fit. The analysis also gives us an estimate for the failure rate `w=0.26(11)` of our detectors — they fail about a quarter of the time.

5.3 A Variation

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

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

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

The final results are quite similar to the other model:

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

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

logBF = -24.164(63)
```

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

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

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

CHAPTER

SIX

LSQFIT - NONLINEAR LEAST SQUARES FITTING

6.1 Introduction

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

1. The dependent data `y` that is to be fit — typically `y` is a Python dictionary in an *lsqfit* analysis. Its values `y[k]` are either `gvar.GVar`s or arrays (any shape or dimension) of `gvar.GVar`s that specify the values of the dependent variables and their errors.
2. A collection `x` of independent data — `x` can have any structure and contain any data, or it can be omitted.
3. A fit function `f(x, p)` whose parameters `p` are adjusted by the fit until `f(x, p)` equals `y` to within `y`s errors — parameters *p* ' are usually specified by a dictionary whose values `p[k]` are individual parameters or (`numpy`) arrays of parameters. The fit function is assumed independent of `x` (that is, `f(p)`) if `x = False` (or if `x` is omitted from the input data).
4. Initial estimates or *priors* for each parameter in `p` — priors are usually specified using a dictionary `prior` whose values `prior[k]` are `gvar.GVar`s or arrays of `gvar.GVar`s that give initial estimates (values and errors) for parameters `p[k]`.

A typical code sequence has the structure:

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

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

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

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

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

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

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

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

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

6.2 nonlinear_fit Objects

class lsqfit.**nonlinear_fit**(*data*, *fcn*, *prior=None*, *p0=None*, *extend=False*, *svdcut=1e-12*, *debug=False*, *tol=1e-8*, *maxit=1000*, *fitter='gsl_multifit'*, ***fitterargs*)

Nonlinear least-squares fit.

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

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

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

Parameters

- **data** (*dict, array or tuple*) – Data to be fit by *lsqfit.nonlinear_fit* can have any of the following forms:

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

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

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

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

  Setting x=False in the first, third or fourth of these formats implies that the fit function depends only on the fit parameters: that is, fcn(p) instead of fcn(x, p). (This is not

assumed if x=None.)

- **fcn** (*callable*) – The function to be fit to data. It is either a function of the independent data x and the fit parameters p (fcn(x, p)), or a function of just the fit parameters (fcn(p)) when there is no x data or x=False. The parameters are tuned in the fit until the function returns values that agree with the y data to within the ys' errors. The function's return value must have the same layout as the y data (a dictionary or an array). The fit parameters p are either: 1) a dictionary where each p[k] is a single parameter or an array of parameters (any shape); or, 2) a single array of parameters. The layout of the parameters is the same as that of prior prior if it is specified; otherwise, it is inferred from of the starting value p0 for the fit.

- **prior** (*dict, array, str, gvar.GVar or None*) – A dictionary (or array) containing *a priori* estimates for all parameters p used by fit function fcn(x, p) (or fcn(p)). Fit parameters p are stored in a dictionary (or array) with the same keys and structure (or shape) as prior. The default value is None; prior must be defined if p0 is None.

- **p0** (*dict, array, float or None*) – Starting values for fit parameters in fit. *lsqfit.nonlinear_fit* adjusts p0 to make it consistent in shape and structure with prior when the latter is specified: elements missing from p0 are filled in using prior, and elements in p0 that are not in prior are discarded. If p0 is a string, it is taken as a file name and *lsqfit.nonlinear_fit* attempts to read starting values from that file; best-fit parameter values are written out to the same file after the fit (for priming future fits). If p0 is None or the attempt to read the file fails, starting values are extracted from prior. The default value is None; p0 must be defined if prior is None.

- **svdcut** (*float or None*) – If svdcut is nonzero (but not None), SVD cuts are applied to every block-diagonal sub-matrix of the covariance matrix for the data y and prior (if there is a prior). The blocks are first rescaled so that all diagonal elements equal 1 – that is, the blocks are replaced by the correlation matrices for the corresponding subsets of variables. Then, if svdcut > 0, eigenvalues of the rescaled matrices that are smaller than svdcut times the maximum eigenvalue are replaced by svdcut times the maximum eigenvalue. This makes the covariance matrix less singular and less susceptible to roundoff error. When svdcut < 0, eigenvalues smaller than |svdcut| times the maximum eigenvalue are discarded and the corresponding components in y and prior are zeroed out. Default is 1e-12.

- **extend** (*bool*) – Log-normal and sqrt-normal distributions can be used for fit priors when extend=True, provided the parameters are specified by a dictionary (as opposed to an array). To use such a distribution for a parameter 'c' in the fit prior, replace prior['c'] with a prior specifying its logarithm or square root, designated by prior['log(c)'] or prior['sqrt(c)'], respectively. The dictionaries containing parameters generated by *lsqfit.nonlinear_fit* will have entries for both 'c' and 'log(c)' or 'sqrt(c)', so only the prior need be changed to switch to log-normal/sqrt-normal distributions. Setting extend=False (the default) restricts all parameters to Gaussian distributions. Additional distributions can be added using *gvar.add_parameter_distribution()*.

- **udata** (*dict, array or tuple*) – Same as data but instructs the fitter to ignore correlations between different pieces of data. This speeds up the fit, particularly for large amounts of data, but ignores potentially valuable information if the data actually are correlated. Only one of data or udata should be specified. (Default is None.)

- **fitter** (*str or None*) – Fitter code. Options if GSL is installed include: 'gsl_multifit' (default) and 'gsl_v1_multifit' (original fitter). Options if scipy is installed include: 'scipy_least_squares' (default if GSL not

installed). `gsl_multifit` has many options, providing extensive user control. `scipy_least_squares` can be used for fits where the parameters are bounded. (Bounded parameters can also be implemented, for any of the fitters, using non-Gaussian priors — see the tutorial.)

- **tol** (*float or tuple*) – Assigning `tol=(xtol, gtol, ftol)` causes the fit to stop searching for a minimum when any of

  1. `xtol` >= relative change in parameters between iterations
  2. `gtol` >= relative size of gradient of `chi**2` function
  3. `ftol` >= relative change in `chi**2` between iterations

  is satisfied. See the fitter documentation for detailed definitions of these stopping conditions. Typically one sets `xtol=1/10**d` where `d` is the number of digits of precision desired in the result, while `gtol<<1` and `ftol<<1`. Setting `tol=eps` where `eps` is a number is equivalent to setting `tol=(eps,1e-10,1e-10)`. Setting `tol=(eps1,eps2)` is equivalent to setting `tol=(eps1,eps2,1e-10)`. Default is `tol=1e-8`. (Note: the `ftol` option is disabled in some versions of the GSL library.)

- **maxit** (*int*) – Maximum number of algorithm iterations (or function evaluations for some fitters) in search for minimum; default is 1000.

- **debug** (*bool*) – Set to `True` for extra debugging of the fit function and a check for round-off errors. (Default is `False`.)

- **fitterargs** (*dict*) – Dictionary of additional arguments passed through to the underlying fitter. Different fitters offer different parameters; see the documentation for each.

Objects of type *lsqfit.nonlinear_fit* have the following attributes:

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

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

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

error
str – Error message generated by the underlying fitter when an error occurs. `None` otherwise.

fitter_results
Results returned by the underlying fitter. Refer to the appropriate fitter's documentation for details.

logGBF
float or None – The logarithm of the probability (density) of obtaining the fit data by randomly sampling the parameter model (priors plus fit function) used in the fit — that is, it is `P(data|model)`. This quantity is useful for comparing fits of the same data to different models, with different priors and/or fit functions. The model with the largest value of `fit.logGBF` is the one preferred by the data. The exponential of the difference in `fit.logGBF` between two models is the ratio of probabilities (Bayes factor) for those models. Differences in `fit.logGBF` smaller than 1 are not very significant. Gaussian statistics are assumed when computing `fit.logGBF`.

p
dict, array or gvar.GVar – Best-fit parameters from fit. Depending upon what was used for the prior (or

p0), it is either: a dictionary (gvar.BufferDict) of gvar.GVars and/or arrays of gvar.GVars; or an array (numpy.ndarray) of gvar.GVars. fit.p represents a multi-dimensional Gaussian distribution which, in Bayesian terminology, is the *posterior* probability distribution of the fit parameters.

pmean
dict, array or float – Means of the best-fit parameters from fit.

psdev
dict, array or float – Standard deviations of the best-fit parameters from fit.

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

p0
dict, array or float – The parameter values used to start the fit. This will differ from the input p0 if the latter was incomplete.

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

Q
float or None – The probability that the chi**2 from the fit could have been larger, by chance, assuming the best-fit model is correct. Good fits have Q values larger than 0.1 or so. Also called the *p-value* of the fit.

stopping_criterion
int – Criterion used to stop fit:

0: didn't converge

1: xtol >= relative change in parameters between iterations

2: gtol >= relative size of gradient of chi**2

3: ftol >= relative change in chi**2 between iterations

svdcorrection
gvar.GVar – Sum of all SVD corrections, if any, added to the fit data y or the prior prior.

svdn
int – Number of eigenmodes modified (and/or deleted) by the SVD cut.

time
float – *CPU* time (in secs) taken by fit.

tol
tuple – Tolerance used in fit. This differs from the input tolerance if the latter was incompletely specified.

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

y
dict, array or gvar.GVar – Fit data used in the fit. This may differ from the input data if an SVD cut is

used. It is either a dictionary (`gvar.BufferDict`) or an array (`numpy.ndarray`), depending upon the input.

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

The global defaults used by *lsqfit.nonlinear_fit* can be changed by changing entries in dictionary `lsqfit.nonlinear_fit.DEFAULTS` for keys 'extend', 'svdcut', 'debug', *tol*, 'maxit', and 'fitter'. Additional defaults can be added to that dictionary to be are passed through *lsqfit.nonlinear_fit* to the underlying fitter (via dictionary `fitterargs`).

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

format(*maxline=0*, *pstyle='v'*)
Formats fit output details into a string for printing.

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

Parameters

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

Returns String containing detailed information about fit.

simulated_fit_iter(*n=None*, *pexact=None*, ***kargs*)
Iterator that returns simulation copies of a fit.

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

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

```
...
fit = nonlinear_fit(...)
...
```

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

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

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

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

```
fit = nonlinear_fit(...)

...

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

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

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

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

Parameters

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

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

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

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

Iterator that returns bootstrap copies of a fit.

A bootstrap analysis involves three steps: 1) make a large number of "bootstrap copies" of the original input data and prior that differ from each other by random amounts characteristic of the underlying randomness in the original data; 2) repeat the entire fit analysis for each bootstrap copy of the data, extracting fit results

from each; and 3) use the variation of the fit results from bootstrap copy to bootstrap copy to determine an approximate probability distribution (possibly non-gaussian) for the fit parameters and/or functions of them: the results from each bootstrap fit are samples from that distribution.

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

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

```
fit = nonlinear_fit(...)

...

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

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

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

Parameters

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

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

dump_p (*filename*)

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

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

dump_pmean (*filename*)

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

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

static load_parameters (*filename*)

Load parameters stored in file `filename`.

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

check_roundoff(*rtol=0.25*, *atol=1e-6*)
Check for roundoff errors in fit.p.

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

static set(*clear=False*, ***defaults*)
Set default parameters for *lsqfit.nonlinear_fit*.

Use to set default values for parameters: `extend`, `svdcut`, `debug`, `tol`, `maxit`, and `fitter`. Can also set parameters specific to the fitter specified by the `fitter` argument.

Sample usage:

```
import lsqfit

old_defaults = lsqfit.nonlinear_fit.set(
    fitter='gsl_multifit', alg='subspace2D', solver='cholesky',
    tol=1e-10, debug=True,
    )
```

`nonlinear_fit.set()` without arguments returns a dictionary containing the current defaults.

Parameters

- **clear** (*bool*) – If `True` remove earlier settings, restoring the original defaults, before adding new defaults. The default value is `clear=False`. `nonlinear_fit.set(clear=True)` restores the original defaults.
- **defaults** (*dict*) – Dictionary containing new defaults.

Returns A dictionary containing the old defaults, before they were updated. These can be restored using `nonlinear_fit.set(old_defaults)` where `old_defaults` is the dictionary containint the old defaults.

6.3 Functions

`lsqfit.`**empbayes_fit**(*z0*, *fitargs*, ***minargs*)
Return fit and `z` corresponding to the fit `lsqfit.nonlinear_fit(**fitargs(z))` that maximizes `logGBF`.

This function maximizes the logarithm of the Bayes Factor from fit `lsqfit.nonlinear_fit(**fitargs(z))` by varying `z`, starting at `z0`. The fit is redone for each value of `z` that is tried, in order to determine `logGBF`.

The Bayes Factor is proportional to the probability that the data came from the model (fit function and priors) used in the fit. *empbayes_fit()* finds the model or data that maximizes this probability.

One application is illustrated by the following code:

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

# fit data
x = np.array([1., 2., 3., 4.])
y = np.array([3.4422, 1.2929, 0.4798, 0.1725])
```

```
# prior
prior = gv.gvar(['10(1)', '1.0(1)'])

# fit function
def fcn(x, p):
    return p[0] * gv.exp( - p[1] * x)

# find optimal dy
def fitargs(z):
    dy = y * z
    newy = gv.gvar(y, dy)
    return dict(data=(x, newy), fcn=fcn, prior=prior)

fit, z = lsqfit.empbayes_fit(0.1, fitargs)
print fit.format(True)
```

Here we want to fit data `y` with fit function `fcn` but we don't know the uncertainties in our `y` values. We assume that the relative errors are `x`-independent and uncorrelated. We add the error `dy` that maximizes the Bayes Factor, as this is the most likely choice. This fit gives the following output:

```
Least Square Fit:
  chi2/dof [dof] = 0.58 [4]    Q = 0.67    logGBF = 7.4834

Parameters:
              0      9.44 (18)      [ 10.0 (1.0) ]
              1    0.9979 (69)      [  1.00 (10) ]

Fit:
     x[k]           y[k]       f(x[k],p)
----------------------------------------
        1      3.442 (54)      3.481 (45)
        2      1.293 (20)      1.283 (11)
        3     0.4798 (75)     0.4731 (41)
        4     0.1725 (27)     0.1744 (23)

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 3/0.0)
```

We have, in effect, used the variation in the data relative to the best fit curve to estimate that the uncertainty in each data point is of order 1.6%.

Parameters

- **z0** (*number, array or dict*) – Starting point for search.
- **fitargs** (*callable*) – Function of `z` that returns a dictionary `args` containing the *lsqfit.nonlinear_fit* arguments corresponding to `z`. `z` should have the same layout (number, array or dictionary) as `z0`. `fitargs(z)` can instead return a tuple `(args, plausibility)`, where `args` is again the dictionary for *lsqfit.nonlinear_fit*. `plausibility` is the logarithm of the *a priori* probabilitiy that `z` is sensible. When `plausibility` is provided, *lsqfit.empbayes_fit()* maximizes the sum `logGBF + plausibility`. Specifying `plausibility` is a way of steering selections away from completely implausible values for `z`.
- **minargs** (*dict*) – Optional argument dictionary, passed on to *lsqfit.gsl_multiminex* (or *lsqfit.scipy_multiminex*), which finds the minimum.

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

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

Weighted average of gvar.GVars or arrays/dicts of gvar.GVars.

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

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

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

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

Typical usage is

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

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

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

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

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

Another example is

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

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

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

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

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

Parameters

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

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

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

dof - Effective number of degrees of freedom.

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

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

time - Time required to do average.

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

fit - Fit output from average.

`lsqfit.`**`gammaQ`**`()`

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

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

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

`gvar.`**`add_parameter_distribution`**`()`

Add new parameter distribution for use in fits.

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

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

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

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

Parameters

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

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

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

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

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

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

6.4 Classes for Bayesian Integrals

lsqfit provides support for doing Bayesian integrals, using results from a least-squares fit to optimize the multi-dimensional integral. This is useful for severely non-Gaussian situations. Module `vegas` is used to do the integrals, using an adaptive Monte Carlo algorithm.

The integrator class is:

class lsqfit.**BayesIntegrator**(*fit*, *limit=1e15*, *scale=1*, *pdf=None*, *adapt_to_pdf=True*, *svdcut=1e-15*)
`vegas` integrator for Bayesian fit integrals.

Parameters

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

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

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

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

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

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

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

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

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

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

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

Parameters:
              a    1.61 (90)      [  0.0 (5.0) ]
              s    0.62 (81)      [  0.0 (2.0) ]
```

```
              g    1.2 (1.1)      [  2.0 (2.0) ]

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

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

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

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

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

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

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

__call__ (*f=None*, *pdf=None*, *adapt_to_pdf=None*, ***kargs*)

Estimate expectation value of function `f(p)`.

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

Parameters

- **f** (*callable*) – Function `f(p)` to integrate. Integral is the expectation value of the function with respect to the distribution. The function can return a number, an array of numbers, or a dictionary whose values are numbers or arrays of numbers. Its argument `p` has the same format as `self.fit.pmean` (that is, either a number, an array, or a dictionary). Omitting `f` (or setting it to `None`) implies that only the PDF is integrated.
- **pdf** (*callable*) – Probability density function `pdf(p)` of the fit parameters to use in place of the normal PDF associated with the least-squares fit used to create the integrator. The PDF need not be normalized; vegas will normalize it. Ignored if `pdf=None` (the default).
- **adapt_to_pdf** (*bool*) – `vegas` adapts to the PDF if `True` (default); otherwise it adapts to `f(p)` times the PDF.

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

The results returned are similar to what `vegas` returns but with an extra attribute: `results.norm`,

which contains the `vegas` estimate for the norm of the PDF. This should equal 1 within errors if the PDF is normalized (and so can serve as a check on the integration in those cases).

A class that describes the Bayesian probability distribution associated with a fit is:

class `lsqfit.`**BayesPDF**(*fit*, *svdcut=1e-15*)

Bayesian probability density function corresponding to *nonlinear_fit* `fit`.

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

Parameters

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

__call__(*p*)

Probability density function evaluated at `p`.

logpdf(*p*)

Logarithm of the probability density function evaluated at `p`.

6.5 `lsqfit.MultiFitter` Classes

lsqfit.MultiFitter provides a framework for fitting multiple pieces of data using a set of custom-designed models, derived from *lsqfit.MultiFitterModel*, each of which encapsulates a particular fit function. This framework was developed to support the `corrfitter` module, but is more general. Instances of model classes associate specific subsets of the fit data with specific subsets of the fit parameters. This allows fit problems to be broken down down into more manageable pieces, which are then aggregated by *lsqfit.MultiFitter* into a single fit.

A trivial example of a model would be one that encapsulates a linear fit function:

```
import numpy as np
import lsqfit

class Linear(lsqfit.MultiFitterModel):
    def __init__(self, datatag, x, intercept, slope):
        super(Linear, self).__init__(datatag)
        # the independent variable
        self.x = np.array(x)
        # keys used to find the intercept and slope in a parameter dictionary
        self.intercept = intercept
        self.slope = slope

    def fitfcn(self, p):
        if self.slope in p:
            return p[self.intercept] + p[self.slope] * self.x
        else:
            # slope parameter marginalized
            return len(self.x) * [p[self.intercept]]
```

```
    def buildprior(self, prior, mopt=None, extend=False):
        " Extract the model's parameters from prior. "
        newprior = {}
        newprior[self.intercept] = prior[self.intercept]
        if mopt is None:
            # slope parameter marginalized if mopt is not None
            newprior[self.slope] = prior[self.slope]
        return newprior

    def builddata(self, data):
        " Extract the model's fit data from data. "
        return data[self.datatag]
```

Imagine four sets of data, each corresponding to `x=1,2,3,4`, all of which have the same intercept but different slopes:

```
data = gv.gvar(dict(
    d1=['1.154(10)', '2.107(16)', '3.042(22)', '3.978(29)'],
    d2=['0.692(10)', '1.196(16)', '1.657(22)', '2.189(29)'],
    d3=['0.107(10)', '0.030(16)', '-0.027(22)', '-0.149(29)'],
    d4=['0.002(10)', '-0.197(16)', '-0.382(22)', '-0.627(29)'],
    ))
```

To find the common intercept, we define a model for each set of data:

```
models = [
   Linear('d1', x=[1,2,3,4], intercept='a', slope='s1'),
   Linear('d2', x=[1,2,3,4], intercept='a', slope='s2'),
   Linear('d3', x=[1,2,3,4], intercept='a', slope='s3'),
   Linear('d4', x=[1,2,3,4], intercept='a', slope='s4'),
   ]
```

This says that `data['d3']`, for example, should be fit with function `p['a'] + p['s3'] * np.array([1, 2,3,4])` where `p` is a dictionary of fit parameters. The models here all share the same intercept, but have different slopes. Assume that we know *a priori* that the intercept and slopes are all order one:

```
prior = gv.gvar(dict(a='0(1)', s1='0(1)', s2='0(1)', s3='0(1)', s4='0(1)'))
```

Then we can fit all the data to determine the intercept:

```
fitter = lsqfit.MultiFitter(models=models)
fit = fitter.lsqfit(data=data, prior=prior)
print(fit)
print('intercept =', fit.p['a'])
```

The output from this code is:

```
Least Square Fit:
  chi2/dof [dof] = 0.49 [16]    Q = 0.95    logGBF = 18.793

Parameters:
              a    0.2012 (78)       [  0.0 (1.0) ]
             s1    0.9485 (53)       [  0.0 (1.0) ]
             s2    0.4927 (53)       [  0.0 (1.0) ]
             s3   -0.0847 (53)       [  0.0 (1.0) ]
             s4   -0.2001 (53)       [  0.0 (1.0) ]
```

```
Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 5/0.0)

intercept = 0.2012(78)
```

Model class `Linear` is configured to allow marginalization of the slope parameter, if desired. Calling `fitter.lsqfit(data=data, prior=prior, mopt=True)` moves the slope parameters into the data (by subtracting `m.x * prior[m.slope]` from the data for each model `m`), and does a single-parameter fit for the intercept:

```
Least Square Fit:
  chi2/dof [dof] = 0.49 [16]    Q = 0.95    logGBF = 18.793

Parameters:
              a    0.2012 (78)      [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 4/0.0)

intercept = 0.2012(78)
```

Marginalization can be useful when fitting large data sets since it reduces the number of fit parameters and simplifies the fit.

Another variation is to replace the simultaneous fit of the four models by a chained fit, where one model is fit at a time and its results are fed into the next fit through that fit's prior. Replacing the fit code by

```
fitter = lsqfit.MultiFitter(models=models)
fit = fitter.chained_lsqfit(data=data, prior=prior)
print(fit.formatall())
print('slope =', fit.p['a'])
```

gives the following output:

```
========== d1
Least Square Fit:
  chi2/dof [dof] = 0.32 [4]    Q = 0.86    logGBF = 2.0969

Parameters:
              a     0.213 (16)      [  0.0 (1.0) ]
             s1    0.9432 (82)      [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 5/0.0)

========== d2
Least Square Fit:
  chi2/dof [dof] = 0.58 [4]    Q = 0.67    logGBF = 5.3792

Parameters:
              a     0.206 (11)      [  0.213 (16) ]
             s2    0.4904 (64)      [   0.0 (1.0) ]
             s1    0.9462 (64)      [ 0.9432 (82) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 5/0.0)

========== d3
Least Square Fit:
```

```
  chi2/dof [dof] = 0.66 [4]    Q = 0.62    logGBF = 5.3767

Parameters:
              a    0.1995 (90)       [  0.206 (11) ]
             s3   -0.0840 (57)       [   0.0 (1.0) ]
             s1    0.9493 (57)       [ 0.9462 (64) ]
             s2    0.4934 (57)       [ 0.4904 (64) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 4/0.0)

========== d4
Least Square Fit:
  chi2/dof [dof] = 0.41 [4]    Q = 0.81    logGBF = 5.9402

Parameters:
              a    0.2012 (78)       [  0.1995 (90) ]
             s4   -0.2001 (53)       [    0.0 (1.0) ]
             s1    0.9485 (53)       [  0.9493 (57) ]
             s2    0.4927 (53)       [  0.4934 (57) ]
             s3   -0.0847 (53)       [ -0.0840 (57) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 4/0.0)

intercept = 0.2012(78)
```

Note how the value for `s1` improves with each fit despite the fact that it appears only in the first fit function. This happens because its value is correlated with that of the intercept `a`, which appears in every fit function.

Chained fits are most useful with very large data sets when it is possible to break the data into smaller, more manageable chunks. There are a variety of options for organizing the chain of fits; these are discussed in the *MultiFitter.chained_lsqfit()* documentation.

class lsqfit.**MultiFitter**(*models*, *mopt=None*, *ratio=False*, *fast=True*, *extend=False*, ***fitterargs*)

Nonlinear least-squares fitter for a collection of models.

Parameters

- **models** – List of models, derived from `modelfitter.MultiFitterModel`, to be fit to the data. Individual models in the list can be replaced by lists of models or tuples of models; see below.
- **mopt** (*object*) – Marginalization options. If not `None`, marginalization is used to reduce the number of fit parameters. Object `mopt` is passed to the models when constructing the prior for a fit; it typically indicates the degree of marginalization (in a model-dependent fashion). Setting `mopt=None` implies no marginalization.
- **ratio** (*bool*) – If `True`, implement marginalization using ratios: `data_marg = data * fitfcn(prior_marg) / fitfcn(prior)`. If `False` (default), implement using differences: `data_marg = data + (fitfcn(prior_marg) - fitfcn(prior))`.
- **fast** (*bool*) – Setting `fast=True` (default) strips any variable not required by the fit from the prior. This speeds fits but loses information about correlations between variables in the fit and those that are not. The information can be restored using `lsqfit.wavg` after the fit.
- **extend** (*bool*) – If `True` supports log-normal and other non-Gaussian priors. See

lsqfit documentation for details. Default is `False`.

- **fitname** (callable or `None`) – Individual fits in a chained fit are assigned default names, constructed from the datatags of the corresponding models, for access and reporting. These names get unwieldy when lots of models are involved. When `fitname` is not `None` (default), each default name `dname` is replaced by `fitname(dname)`.
- **wavg_svdcut** (*float*) – SVD cut used for the weighted averages that combine results from parallel sub-fits in a chained fit (see *MultiFitter.chained_lsqfit()*). Default value is `None` which sets the SVD cut equal to 10x the SVD cut used for other fits (specified by keyword `svdcut`). Weighted averages often need larger SVD cuts than the other fits.
- **fitterargs** – Additional arguments for the *lsqfit.nonlinear_fit* object used to do the fits. These can include `tol`, `maxit`, `svdcut`, `fitter`, etc., as needed.

lsqfit(*data=None*, *prior=None*, *pdata=None*, *p0=None*, ***kargs*)

Compute least-squares fit of models to data.

MultiFitter.lsqfit() fits all of the models together, in a single fit. It returns the *lsqfit.nonlinear_fit* object from the fit.

To see plots of the fit data divided by the fit function with the best-fit parameters use

```
fit.show_plots()
```

Plotting requires module `matplotlib`.

Parameters

- **data** – Input data. One of `data` or `pdata` must be specified but not both. `pdata` is obtained from `data` by collecting the output from `m.builddata(data)` for each model `m` and storing it in a dictionary with key `m.datatag`.
- **pdata** – Input data that has been processed by the models using *MultiFitter.process_data()* or *MultiFitter.process_dataset()*. One of `data` or `pdata` must be specified but not both.
- **prior** – Bayesian prior for fit parameters used by the models.
- **p0** – Dictionary , indexed by parameter labels, containing initial values for the parameters in the fit. Setting `p0=None` implies that initial values are extracted from the prior. Setting `p0="filename"` causes the fitter to look in the file with name `"filename"` for initial values and to write out best-fit parameter values after the fit (for the next call to `self.lsqfit()`).
- **wavg_svdcut** (*float*) – SVD cut used in weighted averages for parallel fits.
- **kargs** – Arguments that override parameters specified when the *MultiFitter* was created. Can also include additional arguments to be passed through to the *lsqfit* fitter.

chained_lsqfit(*data=None*, *pdata=None*, *prior=None*, *p0=None*, ***kargs*)

Compute chained least-squares fit of models to data.

In a chained fit to models `[s1, s2, ...]`, the models are fit one at a time, with the fit output from one being fed into the prior for the next. This can be much faster than fitting the models together, simultaneously. The final result comes from the last fit in the chain, and includes parameters from all of the models.

The most general chain has the structure `[s1, s2, s3 ...]` where each `sn` is one of:

1. a model (derived from `multifitter.MultiFitterModel`);
2. **a tuple `(m1, m2, m3)` of models, to be fit together in** a single fit (*i.e.*, simultaneously);

3. **a list `[p1, p2, p3 ...]` where each `pn` is either** a model or a tuple of models (see #2). The `pn` are fit separately, and independently of each other (*i.e.*, in parallel). Results from the separate fits are averaged at the end to provide a single composite result for the collection of fits.

The final result `fit` returned by `MultiFitter.chained_fit()` has an extra attribute `fit.chained_fits` which is an ordered dictionary containing fit results from each link `sn` in the chain, and keyed by the models' `datatag`s. If any of these involves parallel fits (see #3 above), it will have an extra attribute `fit.chained_fits[fittag].sub_fits` that contains results from the separate parallel fits. To list results from all the chained and parallel fits, use

```
print(fit.formatall())
```

To see plots of the fit data divided by the fit function with the best-fit parameters use

fit.show_plots()

Plotting requires module `matplotlib`.

Parameters

- **`data`** – Input data. One of `data` or `pdata` must be specified but not both. `pdata` is obtained from `data` by collecting the output from `m.builddata(data)` for each model `m` and storing it in a dictionary with key `m.datatag`.
- **`pdata`** – Input data that has been processed by the models using *`MultiFitter.process_data()`* or *`MultiFitter.process_dataset()`*. One of `data` or `pdata` must be specified but not both.
- **`prior`** – Bayesian prior for fit parameters used by the models.
- **`p0`** – Dictionary , indexed by parameter labels, containing initial values for the parameters in the fit. Setting `p0=None` implies that initial values are extracted from the prior. Setting `p0="filename"` causes the fitter to look in the file with name `"filename"` for initial values and to write out best-fit parameter values after the fit (for the next call to `self.lsqfit()`).
- **`kargs`** – Arguments that override parameters specified when the *`MultiFitter`* was created. Can also include additional arguments to be passed through to the *`lsqfit`* fitter.

static process_data(*data*, *models*)

Convert `data` to processed data using `models`.

Data from dictionary `data` is processed by each model in list `models`, and the results collected into a new dictionary `pdata` for use in *`MultiFitter.lsqfit()`* and `MultiFitter.chained_lsqft()`.

static process_dataset(*dataset*, *models*, ***kargs*)

Convert `dataset` to processed data using `models`.

`gvar.dataset.Dataset` (or similar dictionary) object `dataset` is processed by each model in list `models`, and the results collected into a new dictionary `pdata` for use in *`MultiFitter.lsqfit()`* and `MultiFitter.chained_lsqft()`. Assumes that the models have defined method *`MultiFitterModel.builddataset()`*. Keyword arguments `kargs` are passed on to `gvar.dataset.avg_data()` when averaging the data.

static show_plots(*fitdata*, *fitval*, *x=None*, *save=False*)

Show plots of `fitdata[k]/fitval[k]` for each key `k` in `fitval`.

Assumes `matplotlib` is installed (to make the plots). Plots are shown for one correlator at a time. Press key `n` to see the next correlator; press key `p` to see the previous one; press key `q` to quit the plot and return control to the calling program; press a digit to go directly to one of the first ten plots. Zoom, pan and save using the window controls.

Copies of the plots that are viewed can be saved by setting parameter `save=prefix` where `prefix` is a string used to create file names: the file name for the plot corresponding to key `k` is `prefix.format(k)`. It is important that the filename end with a suffix indicating the type of plot file desired: e.g., `prefix='plot-{}.pdf'`.

static flatten_models(*models*)
Create 1d-array containing all disctinct models from `models`.

lsqfit.MultiFitter models are derived from the following class. Methods `buildprior`, `builddata`, `fitfcn`, and `builddataset` are not implemented in this base class. They need to be overwritten by the derived class (except for `builddataset` which is optional).

class lsqfit.**MultiFitterModel**(*datatag*, *ncg=1*)
Base class for MultiFitter models.

Derived classes must define methods `fitfcn`, `buildprior`, and `builddata`, all of which are described below. In addition they have attributes:

datatag
lsqfit.MultiFitter builds fit data for the correlator by extracting the data labelled by `datatag` (eg, a string) from an input data set (eg, a dictionary). This label is stored in the `MultiFitterModel` and must be passed to its constructor. It must be a hashable quantity, like a string or number or tuple of strings and numbers.

ncg
When `ncg>1`, fit data and functions are coarse-grained by breaking them up into bins of of `ncg` values and replacing each bin by its average. This can increase the fitting speed, because their is less data, without much loss of precision if the data elements within a bin are highly correlated.

Parameters

- **datatag** – Label used to identify model's data.
- **ncg** (*int*) – Size of bins for coarse graining (default is `ncg=1`).

buildprior(*prior*, *mopt=None*, *extend=False*)
Extract fit prior from `prior`.

Returns a dictionary containing the part of dictionary `prior` that is relevant to this model's fit. The code could be as simple as collecting the appropriate pieces: e.g.,

```
def buildprior(self, prior, mopt=None, extend=False):
    mprior = gv.BufferDict()
    model_keys = [...]
    for k in model_keys:
        mprior[k] = prior[k]
    return mprior
```

where `model_keys` is a list of keys corresponding to the model's parameters. Supporting the `extend` option requires a slight modification: e.g.,

```
def buildprior(self, prior, mopt=None, extend=False):
    mprior = gv.BufferDict()
    model_keys = [...]
    for k in self.get_prior_keys(prior, model_keys, extend):
        mprior[k] = prior[k]
    return mprior
```

Marginalization involves omitting some of the fit parameters from the model's prior. `mopt=None` implies no marginalization. Otherwise `mopt` will typically contain information about what and how much to marginalize.

Parameters

- **prior** – Dictionary containing *a priori* estimates of all fit parameters.
- **mopt** (*object*) – Marginalization options. Ignore if `None`. Otherwise marginalize fit parameters as specified by `mopt`. `mopt` can be any type of Python object; it is used only in `buildprior` and is passed through to it unchanged.
- **extend** (*bool*) – If `True` supports log-normal and other non-Gaussian priors. See *lsqfit* documentation for details.

builddata(*data*)

Extract fit data corresponding to this model from data set `data`.

The fit data is returned in a 1-dimensional array; the fitfcn must return arrays of the same length.

Parameters **data** – Data set containing the fit data for all models. This is typically a dictionary, whose keys are the `datatag`s of the models.

fitfcn(*p*)

Compute fit function fit for parameters `p`.

Results are returned in a 1-dimensional array the same length as (and corresponding to) the fit data returned by `self.builddata(data)`.

If marginalization is supported, `fitfcn` must work with or without the marginalized parameters.

Parameters **p** – Dictionary of parameter values.

builddataset(*dataset*)

Extract fit dataset from `gvar.dataset.Dataset` `dataset`.

The code

```
import gvar as gv

data = gv.dataset.avg_data(m.builddataset(dataset))
```

that builds data for model `m` should be functionally equivalent to

```
import gvar as gv

data = m.builddata(gv.dataset.avg_data(dataset))
```

This method is optional. It is used only by *MultiFitter.process_dataset()*.

Parameters **dataset** – `gvar.dataset.Dataset` (or similar dictionary) dataset containing the fit data for all models. This is typically a dictionary, whose keys are the `datatag`s of the models.

static get_prior_keys(*prior*, *keys*, *extend=False*)

Return list of keys in dictionary `prior` for keys in list `keys`.

List `keys` is returned if `extend=False`. Otherwise the keys returned may differ from those in `keys`. For example, a prior that has a key `log(x)` would return that key in place of a key `x` in list `keys`. This support non-Gaussian priors as discussed in the *lsqfit* documentation.

static prior_key()

Find base key in `prior` corresponding to `k`.

6.6 Requirements

lsqfit relies heavily on the `gvar`, and `numpy` modules. Also the fitting and minimization routines are from the Gnu Scientific Library (GSL) and/or the Python `scipy` module.

CHAPTER

SEVEN

GSL ROUTINES

7.1 Fitters

lsqfit uses routines from the GSL C-library provided it is installed; GSL is the open-source Gnu Scientific Library. There are two fitters that are available for use by *lsqfit.nonlinear_fit*.

class lsqfit.**gsl_multifit**

GSL fitter for nonlinear least-squares multidimensional fits.

`gsl_multifit` is a function-class whose constructor does a least-squares fit by minimizing `sum_i f_i(x)**2` as a function of vector `x`.

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

Parameters

- **x0** (*array of floats*) – Starting point for minimization.
- **n** (*positive int*) – Length of vector returned by the fit function `f(x)`.
- **f** (*array-valued function*) – `sum_i f_i(x)**2` is minimized by varying parameters `x`. The parameters are a 1-d `numpy` array of either numbers or `gvar.GVar`s.
- **tol** (*float or tuple*) – Assigning `tol=(xtol, gtol, ftol)` causes the fit to stop searching for a minimum when any of

  `xtol` >= relative change in parameters between iterations

  `gtol` >= relative size of gradient of `chi**2`

  `ftol` >= relative change in `chi**2` between iterations

  is statisfied. See the GSL documentation for detailed definitions of the stopping conditions. Typically one sets `xtol=1/10**d` where `d` is the number of digits of precision desired in the result, while `gtol<<1` and `ftol<<1`. Setting `tol=eps` where `eps` is a number is equivalent to setting `tol=(eps,1e-10,1e-10)`. Setting `tol=(eps1,eps2)` is equivlent to setting `tol=(eps1,eps2,1e-10)`. Default is `tol=1e-5`. (Note: `ftol` option is disabled in some versions of the GSL library.)
- **maxit** (*int*) – Maximum number of iterations in search for minimum; default is 1000.
- **alg** (*str*) – GSL algorithm to use for minimization. The following options are supported (see GSL documentation for more information):

  'lm' Levenberg-Marquardt algorithm (default).

  'lmaccel' Levenberg-Marquardt algorithm with geodesic acceleration. Can be faster than `'lm'` but less stable. Stability is controlled by damping parameter `avmax`; setting it to zero turns acceleration off.

'subspace2D' 2D generalization of dogleg algorithm. This can be substantially faster than the two `'lm'` algorithms.

'dogleg' dogleg algorithm.

'ddogleg' double dogleg algorithm.

- **scaler** (*str*) – Scaling method used in minimization. The following options are supported (see GSL documentation for more information):

  'more' More rescaling, which makes the problem scale invariant. Default.

  'levenberg' Levenberg rescaling, which is not scale invariant but may be more efficient in certain problems.

  'marquardt' Marquardt rescaling. Probably not as good as the other two options.

- **solver** (*str*) – Method use to solve the linear equations for the solution from a given step. The following options are supported (see GSL documentation for more information):

  'qr' QR decomposition of the Jacobian. Default.

  'cholesky' Cholesky decomposition of the Jacobian. Can be substantially faster than `'qr'` but not as reliable for singular Jacobians.

  'svd' SVD decomposition. The most robust for singular situations, but also the slowest.

- **factor_up** (*float*) – Factor by which search region is increased when a search step is accepted. Values that are too large destablize the search; values that are too small slow down the search. Default is `factor_up=3`.

- **factor_down** (*float*) – Factor by which search region is decreased when a search step is rejected. Values that are too large destablize the search; values that are too small slow down the search. Default is `factor_up=2`.

- **avmax** (*float*) – Damping parameter for geodesic acceleration. It is the maximum allowed value for the acceleration divided by the velocity. Smaller values imply less acceleration. Default is `avmax=0.75`.

lsqfit.gsl_multifit objects have the following attributes.

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

cov
array – Covariance matrix at the minimum point.

description
str – Short description of internal fitter settings.

f
array – Fit function value `f(x)` at the minimum in the most recent fit.

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

nit
int – Number of function evaluations used in last fit to find the minimum.

stopping_criterion
int – Criterion used to stop fit:

0. didn't converge

1. `xtol` >= relative change in parameters between iterations
2. `gtol` >= relative size of gradient of `chi**2`
3. `ftol` >= relative change in `chi**2` between iterations

error
str or None – `None` if fit successful; an error message otherwise.

class lsqfit.**gsl_v1_multifit**
Fitter for nonlinear least-squares multidimensional fits. (GSL v1.)

`gsl_v1_ multifit` is a function-class whose constructor does a least-squares fit by minimizing `sum_i f_i(x)**2` as a function of vector `x`.

`gsl_v1_multifit` is a wrapper for the (older, v1) `multifit` GSL routine (see `nlin.h`). This package was used in earlier versions of *lsqfit* (<9.0).

Parameters

- **x0** (*array of floats*) – Starting point for minimization.
- **n** (*positive int*) – Length of vector returned by the fit function `f(x)`.
- **f** (*array-valued function*) – `sum_i f_i(x)**2` is minimized by varying parameters `x`. The parameters are a 1-d `numpy` array of either numbers or `gvar.GVar`s.
- **tol** (*float or tuple*) – Assigning `tol=(xtol, gtol, ftol)` causes the fit to stop searching for a minimum when any of

  `xtol` >= relative change in parameters between iterations

  `gtol` >= relative size of gradient of `chi**2`

  `ftol` >= relative change in `chi**2` between iterations

  is statisfied. See the GSL documentation for detailed definitions of the stopping conditions. Typically one sets `xtol=1/10**d` where `d` is the number of digits of precision desired in the result, while `gtol<<1` and `ftol<<1`. Setting `tol=eps` where `eps` is a number is equivalent to setting `tol=(eps,1e-10,1e-10)`. Setting `tol=(eps1,eps2)` is equivlent to setting `tol=(eps1,eps2,1e-10)`. Default is `tol=1e-5`. (Note: the `ftol` option is disabled in some versions of the GSL library.)
- **maxit** (*int*) – Maximum number of iterations in search for minimum; default is 1000.
- **alg** (*str*) – GSL algorithm to use for minimization. Two options are currently available: `"lmsder"`, the scaled LMDER algorithm (default); and `"lmder"`, the unscaled LMDER algorithm. With version 2 of the GSL library, another option is `"lmniel"`, which can be useful when there is much more data than parameters.
- **analyzer** – Optional function of `x, [...f_i(x)...], [[..df_ij(x)..]]` which is called after each iteration. This can be used to inspect intermediate steps in the minimization, if needed.

lsqfit.gsl_v1_multifit objects have the following attributes.

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

cov
array – Covariance matrix at the minimum point.

f
callable – Fit function value `f(x)` at the minimum in the most recent fit.

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

nit
int – Number of function evaluations used in last fit to find the minimum.

stopping_criterion
int – Criterion used to stop fit:

0. didn't converge
1. `xtol` >= relative change in parameters between iterations
2. `gtol` >= relative size of gradient of `chi**2`
3. `ftol` >= relative change in `chi**2` between iterations

error
str or None – `None` if fit successful; an error message otherwise.

7.2 Minimizer

The *lsqfit.empbayes_fit()* uses a minimizer from the GSL library to minimize `logGBF`.

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

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

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

Parameters

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

lsqfit.gsl_multiminex objects have the following attributes.

x
array – Location of the minimum.

f
float – Value of function `f(x)` at the minimum.

nit
int – Number of iterations required to find the minimum.

error
None or str – `None` if minimization successful; an error message otherwise.

CHAPTER

EIGHT

SCIPY ROUTINES

8.1 Fitter

lsqfit uses routines from the open-source `scipy` Python module provided it is installed. These routines are used in place of GSL routines if the latter are not installed. There is one fitter available for use by *lsqfit.nonlinear_fit*.

class lsqfit.**scipy_least_squares**(*x0*, *n*, *f*, *tol=(1e-08, 1e-08, 1e-08)*, *maxit=1000*, ***extra_args*)

`scipy` fitter for nonlinear least-squares multidimensional fits.

`scipy_least_squares` is a function-class whose constructor does a least-squares fit by minimizing `sum_i f_i(x)**2` as a function of vector `x`.

`scipy_least_squares` is a wrapper for the `scipy.optimize.least_squares`.

Parameters

- **x0** (*array of floats*) – Starting point for minimization.
- **n** (*positive int*) – Length of vector returned by the fit function `f(x)`.
- **f** (*array-valued function*) – `sum_i f_i(x)**2` is minimized by varying parameters `x`. The parameters are a 1-d `numpy` array of either numbers or `gvar.GVars`.
- **tol** (*float or tuple*) – Assigning `tol=(xtol, gtol, ftol)` causes the fit to stop searching for a minimum when any of

  `xtol` >= relative change in parameters between iterations

  `gtol` >= relative size of gradient of `chi**2`

  `ftol` >= relative change in `chi**2` between iterations

  is statisfied. See the `scipy.optimize.least_squares` documentation detailed definitions of the stopping conditions. Typically one sets `xtol=1/10**d` where `d` is the number of digits of precision desired in the result, while `gtol<<1` and `ftol<<1`. Setting `tol=eps` where `eps` is a number is equivalent to setting `tol=(eps,1e-10,1e-10)`. Setting `tol=(eps1,eps2)` is equivlent to setting `tol=(eps1,eps2,1e-10)`. Default is `tol=1e-5`.
- **method** (*str or None*) – Minimization algorithm. Options include:

  'trf' Trusted Region Reflective algorithm (default). Best choice with bounded parameters.

  'dogbox' dogleg algorithm adapted for bounded parameters.

'lm' Levenberg-Marquardt algorithm as implemented in MINPACK. Best for smaller problems. Does not work with bounded parameters (bounds are ignored).

Setting `method=None` implies the default `'trf'`.

- **maxit** (*int*) – Maximum number of function evaluations in search for minimum; default is 1000.

Other arguments include: `x_jac`, `loss`, `tr_solver`, `f_scale`, `tr_options`, `bounds`. See the documentation for `scipy.optimize.least_squares` for information about these and other options.

lsqfit.scipy_least_squares objects have the following attributes.

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

cov
array – Covariance matrix at the minimum point.

description
str – Short description of internal fitter settings.

f
array – Fit function value `f(x)` at the minimum in the most recent fit.

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

nit
int – Number of function evaluations used in last fit to find the minimum.

stopping_criterion
int – Criterion used to stop fit:

0. didn't converge
1. `xtol` >= relative change in parameters between iterations
2. `gtol` >= relative size of gradient of `chi**2`
3. `ftol` >= relative change in `chi**2` between iterations

error
str or None – `None` if fit successful; an error message otherwise.

results
dict – Results returned by `scipy.optimize.least_squares`.

8.2 Minimizer

The *lsqfit.empbayes_fit()* uses a minimizer from the `scipy` module to minimize `logGBF`.

class lsqfit.**scipy_multiminex**(*x0*, *f*, *tol=1e-4*, *maxit=1000*, *step=1*, *alg='nmsimplex2'*, *analyzer=None*)

`scipy` minimizer for multidimensional functions.

`scipy_multiminex` is a function-class whose constructor minimizes a multidimensional function `f(x)` by varying vector `x`. This routine does *not* use user-supplied information about the gradient of `f(x)`.

`scipy_multiminex` is a wrapper for the `minimize` `scipy` function. It gives access to only part of that function.

Parameters

- **x0** (*array of floats*) – Starting point for minimization search.
- **f** – Function `f(x)` to be minimized by varying vector `x`.
- **tol** (*float*) – Minimization stops when `x` has converged to with tolerance `tol`; default is `1e-4`.
- **maxit** (*positive int*) – Maximum number of iterations in search for minimum; default is 1000.
- **analyzer** (*function*) – Optional function of the current `x`. This can be used to inspect intermediate steps in the minimization, if needed.

lsqfit.scipy_multiminex objects have the following attributes.

x
array – Location of the minimum.

f
float – Value of function `f(x)` at the minimum.

nit
int – Number of iterations required to find the minimum.

error
Noe or str – `None` if fit successful; an error message otherwise.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

P

Q

R

S

T

W

X

Y