lsqfit Documentation

Release 4.8.4

G. P. Lepage

May 24, 2014

Contents:

CHAPTER

ONE

OVERVIEW AND TUTORIAL

1.1 Introduction

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

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

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

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

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

# do the fit
fit = lsqfit.nonlinear_fit(data=(x, y), prior=prior, fcn=fcn)
print(fit.format(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 ys' errors. The means and covariance matrices for the ys are specified in the gv.gvar(...)s used to create them: for example,

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

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

The output from the code sample above is:

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

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

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

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

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

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

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

as expected given that the `chi**2` per degree of freedom is only 0.17.

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

```
... as before ...

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

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

... as before ...
```

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

There are several things worth noting from this example:

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

What follows is a brief tutorial that demonstrates in greater detail how to use these modules in some standard variations on the data fitting problem. As above, code for the examples is specified completely and so can be copied into a file, and run as is. It can also be modified, allowing for experimentation. At the very end, in an appendix, there is a very simple pedagogical example that illustrates the nature of priors and demonstrates some of the simpler techniques supported by `lsqfit`.

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

```
from __future__ import print_function
```

at the start of your file.

1.2 Gaussian Random Variables and Error Propagation

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

There are three important things to know about `gvar.GVar`s, in addition to knowing how to create them (see *Creating Gaussian Variables*):

1. `gvar.GVar`s describe not only means and standard deviations, but also statistical correlations between different objects. For example, the `gvar.GVar`s 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.     ]]
   ```

2. `gvar.GVar`s can be used in arithmetic expressions or as arguments to pure-Python functions. The results are also `gvar.GVar`s. Covariances are propagated through these expressions following the usual rules, (automat-

ically) preserving information about correlations. For example, the gvar.GVars a and b above could have been created using the following code:

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

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

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

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

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

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

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

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

gvar.BufferDicts were created specifically to handle gvar.GVars, although they can be quite useful

with other data types as well. The values in a pickled `gvar.BufferDict` can be individual `gvar.GVar`s or arbitrary `numpy` arrays of `gvar.GVar`s. See *Storing gvar.GVars for Later Use; gvar.BufferDicts* for more information.

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)`); and 3) some *a priori* idea about the fit parameters' values. The *a priori* information about a parameter could be fairly imprecise — for example, the parameter is order 1. The point of the fit is to improve our knowledge of the parameter values, beyond our *a priori* impressions, by analyzing the fit data. We now show how to do this using the `lsqfit` module.

For this example, we use fake data generated by a function, `make_data()`, that is described at the end of this section. The function call `x,y = make_data()` generates 15 values for `x`, equal to `1,2,3..10,12,14..20`, and 15 values for `y`, where each `y` is obtained by adding random noise to the value of a function of the corresponding `x`. The function of `x` we use is:

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

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

```
>>> print(y)
[0.2752(27) 0.07951(80) ... ]

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

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

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

```
import numpy as np

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

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

Finally we need to define priors that encapsulate our *a priori* knowledge about the fit-parameter values. In practice we almost always have *a priori* knowledge about parameters; it is usually impossible to design a fit function without some sense of the parameter sizes. Given such knowledge it is important (usually essential) to include it in the fit. This is done by designing priors for the fit, which are probability distributions for each parameter that describe the *a priori* uncertainty in that parameter. As discussed in the previous section, we use objects of type `gvar.GVar` to describe

(Gaussian) probability distributions. Let's assume that before the fit we suspect that each `a[i]` is of order 0.5±0.5, while `E[i]` is of order (1+i)±0.5. A prior that represents this information is built using the following code:

```
import lsqfit
import gvar as gv

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

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

```
>>> print(prior['a'])
[0.50(50) 0.50(50) 0.50(50)]
>>> print(prior['E'])
[1.00(50), 2.00(50), 3.00(50)]
```

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

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

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

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

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

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

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

def make_data(nexp=100, eps=0.01): # make x, y fit data
    x = np.array([1.,2.,3.,4.,5.,6.,7.,8.,9.,10.,12.,14.,16.,18.,20.])
    cr = gv.gvar(0.0, eps)
    c = [gv.gvar(cr(), eps) for n in range(100)]
    x_xmax = x/max(x)
    noise = 1+ sum(c[n] * x_xmax ** n for n in range(100))
    y = f_exact(x, nexp) * noise
    return x, y

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

def main():
    gv.ranseed([2009, 2010, 2011, 2012]) # initialize random numbers (opt.)
    x, y = make_data()                  # make fit data
```

```
    p0 = None                       # make larger fits go faster (opt.)
    for nexp in range(3, 20):
        print('************************************ nexp =', nexp)
        prior = make_prior(nexp)
        fit = lsqfit.nonlinear_fit(data=(x, y), fcn=f, prior=prior, p0=p0)
        print(fit)                  # print the fit results
        E = fit.p['E']              # best-fit parameters
        a = fit.p['a']
        print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
        print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
        print()
        if fit.chi2 / fit.dof < 1.:
            p0 = fit.pmean          # starting point for next fit (opt.)

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

We are not sure *a priori* how many exponentials are needed to fit our data. Given that there are only fifteen `y`s, and these are noisy, there may only be information in the data about the first few terms. Consequently we write our code to try fitting with each of `nexp=3,4,5..19` terms. (The pieces of the code involving `p0` are optional; they make the more complicated fits go about 30 times faster since the output from one fit is used as the starting point for the next fit — see the discussion of the `p0` parameter for `lsqfit.nonlinear_fit`.) Running this code produces the following output, which is reproduced here in some detail in order to illustrate a variety of features:

```
************************************ nexp = 3
Least Square Fit:
  chi2/dof [dof] = 6.3e+02 [15]    Q = 0    logGBF = -4465

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

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0    (itns/time = 30/0.0)

E1/E0 = 1.9986(24)   E2/E0 = 3.6255(62)
a1/a0 = 1.23130(47)   a2/a0 = 2.7070(13)

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

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

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0    (itns/time = 231/0.1)
```

```
E1/E0 = 2.0018(12)   E2/E0 = 3.0800(98)
a1/a0 = 1.0094(30)   a2/a0 = 1.233(14)

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

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

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0    (itns/time = 6/0.0)

E1/E0 = 2.0013(14)   E2/E0 = 3.065(24)
a1/a0 = 1.0075(42)   a2/a0 = 1.189(63)

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

Parameters:
            a 0    0.4018 (40)      [  0.50 (50) ]
              1    0.4041 (47)      [  0.50 (50) ]
              2     0.461 (41)      [  0.50 (50) ]
              3      0.60 (24)      [  0.50 (50) ]
              4      0.47 (37)      [  0.50 (50) ]
              5      0.45 (46)      [  0.50 (50) ]
            E 0   0.90035 (51)      [  1.00 (50) ]
              1    1.8015 (17)      [  2.00 (50) ]
              2     2.746 (34)      [  3.00 (50) ]
              3      3.98 (32)      [  4.00 (50) ]
              4      4.96 (49)      [  5.00 (50) ]
              5      6.01 (50)      [  6.00 (50) ]

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0    (itns/time = 6/0.0)

E1/E0 = 2.0008(17)   E2/E0 = 3.049(37)
a1/a0 = 1.0055(56)   a2/a0 = 1.15(10)

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

Parameters:
            a 0    0.4018 (40)      [  0.50 (50) ]
              1    0.4036 (48)      [  0.50 (50) ]
              2     0.452 (47)      [  0.50 (50) ]
              3      0.60 (22)      [  0.50 (50) ]
```

```
              4      0.42 (37)       [  0.50 (50) ]
              5      0.42 (46)       [  0.50 (50) ]
              6      0.46 (49)       [  0.50 (50) ]
            E 0   0.90035 (51)       [  1.00 (50) ]
              1    1.8012 (18)       [  2.00 (50) ]
              2     2.739 (39)       [  3.00 (50) ]
              3      3.94 (33)       [  4.00 (50) ]
              4      4.96 (49)       [  5.00 (50) ]
              5      6.02 (50)       [  6.00 (50) ]
              6      7.02 (50)       [  7.00 (50) ]

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0    (itns/time = 6/0.0)

E1/E0 = 2.0006(18)   E2/E0 = 3.042(43)
a1/a0 = 1.0045(63)   a2/a0 = 1.13(12)
                                    .
                                    .
                                    .

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

Parameters:
            a 0    0.4018 (40)       [  0.50 (50) ]
              1    0.4033 (49)       [  0.50 (50) ]
              2     0.447 (51)       [  0.50 (50) ]
              3      0.60 (21)       [  0.50 (50) ]
              4      0.38 (37)       [  0.50 (50) ]
              5      0.40 (46)       [  0.50 (50) ]
              6      0.45 (49)       [  0.50 (50) ]
              7      0.48 (50)       [  0.50 (50) ]
              8      0.49 (50)       [  0.50 (50) ]
              9      0.50 (50)       [  0.50 (50) ]
             10      0.50 (50)       [  0.50 (50) ]
             11      0.50 (50)       [  0.50 (50) ]
             12      0.50 (50)       [  0.50 (50) ]
             13      0.50 (50)       [  0.50 (50) ]
             14      0.50 (50)       [  0.50 (50) ]
             15      0.50 (50)       [  0.50 (50) ]
             16      0.50 (50)       [  0.50 (50) ]
             17      0.50 (50)       [  0.50 (50) ]
             18      0.50 (50)       [  0.50 (50) ]
            E 0   0.90035 (51)       [  1.00 (50) ]
              1    1.8011 (19)       [  2.00 (50) ]
              2     2.734 (42)       [  3.00 (50) ]
              3      3.91 (33)       [  4.00 (50) ]
              4      4.97 (49)       [  5.00 (50) ]
              5      6.02 (50)       [  6.00 (50) ]
              6      7.02 (50)       [  7.00 (50) ]
              7      8.01 (50)       [  8.00 (50) ]
              8      9.00 (50)       [  9.00 (50) ]
              9     10.00 (50)       [ 10.00 (50) ]
             10     11.00 (50)       [ 11.00 (50) ]
             11     12.00 (50)       [ 12.00 (50) ]
             12     13.00 (50)       [ 13.00 (50) ]
             13     14.00 (50)       [ 14.00 (50) ]
```

```
             14     15.00 (50)      [ 15.00 (50) ]
             15     16.00 (50)      [ 16.00 (50) ]
             16     17.00 (50)      [ 17.00 (50) ]
             17     18.00 (50)      [ 18.00 (50) ]
             18     19.00 (50)      [ 19.00 (50) ]

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

E1/E0 = 2.0004(19)   E2/E0 = 3.036(47)
a1/a0 = 1.0038(67)   a2/a0 = 1.11(13)

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

There are several things to notice here:

- Clearly three exponentials (`nexp=3`) is not enough. The `chi**2` per degree of freedom (`chi2/dof`) is much larger than one. The `chi**2` improves significantly for `nexp=4` exponentials and by `nexp=6` the fit is as good as it is going to get — there is essentially no change when further exponentials are added.
- The best-fit values for each parameter are listed for each of the fits, together with the prior values (in brackets, on the right). Values for each `a[i]` and `E[i]` are listed in order, starting at the points indicated by the labels `a` and `E`. Asterisks are printed at the end of the line if the mean best-fit value differs from the prior's mean by more than one standard deviation; the number of asterisks, up to a maximum of 5, indicates how many standard deviations the difference is. Differences of one or two standard deviations are not uncommon; larger differences could indicate a problem with the prior or the fit.

  Once the fit converges, the best-fit values for the various parameters agree well — that is to within their errors, approximately — with the exact values, which we know since we are using fake data. For example, `a` and `E` for the first exponential are 0.402(4) and 0.9003(5), respectively, from the fit where the exact answers are 0.4 and 0.9; and we get 0.45(5) and 2.73(4) for the third exponential where the exact values are 0.4 and 2.7.
- Note in the `nexp=7` fit how the means and standard deviations for the parameters governing the seventh (and last) exponential are almost identical to the values in the corresponding priors: 0.46(49) from the fit for `a` and 7.0(5) for `E`. This tells us that our fit data has little or no information to add to what we knew *a priori* about these parameters — there isn't enough data and what we have isn't accurate enough.

  This situation is truer still of further terms as they are added in the `nexp=8` and later fits. This is why the fit results stop changing once we have `nexp=6` exponentials. There is no point in including further exponentials, beyond the need to verify that the fit has indeed converged.
- The last fit includes `nexp=19` exponentials and therefore has 38 parameters. This is in a fit to 15 `ys`. Old-fashioned fits, without priors, are impossible when the number of parameters exceeds the number of data points. That is clearly not the case here, where the number of terms and parameters can be made arbitrarily large, eventually (after `nexp=6` terms) with no effect at all on the results.

  The reason is that the prior that we include for each new parameter is, in effect, a new piece of data (the mean and standard deviation of the *a priori* expectation for that parameter); it leads to a new term in the `chi**2` function. We are fitting both the data and our *a priori* expectations for the parameters. So in the `nexp=19` fit, for example, we actually have 53 pieces of data to fit: the 15 `ys` plus the 38 prior values for the 38 parameters.

  The effective number of degrees of freedom (`dof` in the output above) is the number of pieces of data minus the number of fit parameters, or 53-38=15 in this last case. With priors for every parameter, the number of degrees of freedom is always equal to the number of `ys`, irrespective of how many fit parameters there are.
- The Gaussian Bayes Factor (whose logarithm is `logGBF` in the output) is a measure of the likelihood that the actual data being fit could have come from a theory with the prior and fit function used in the fit. The larger this number, the more likely it is that prior/fit-function and data could be related. Here it grows dramatically from the first fit (`nexp=3`) but then more-or-less stops changing around `nexp=5`. The implication is that this data

is much more likely to have come from a theory with `nexp>=5` than with `nexp=3` (which we know to be the actual case).

- In the code, results for each fit are captured in a Python object `fit`, which is of type `lsqfit.nonlinear_fit`. A summary of the fit information is obtained by printing `fit`. Also the best-fit results for each fit parameter can be accessed through `fit.p`, as is done here to calculate various ratios of parameters.

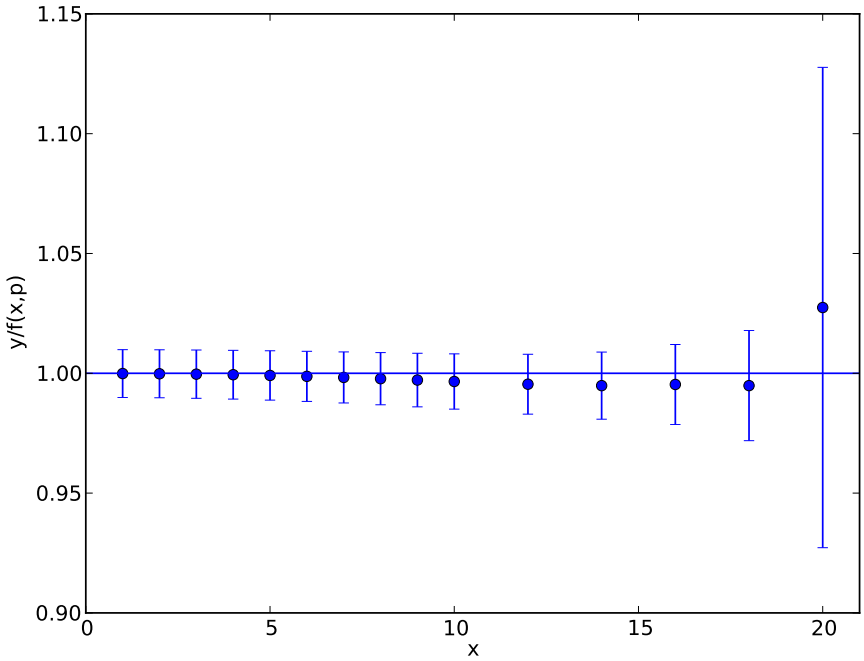
  The errors in these last calculations automatically account for any correlations in the statistical errors for different parameters. This is obvious in the ratio `a1/a0`, which would be 1.004(16) if there was no statistical correlation between our estimates for `a1` and `a0`, but in fact is 1.004(7) in this fit. The (positive) correlation is evident in the covariance matrix:

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

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

```
Fit:
     x[k]               y[k]           f(x[k],p)
-----------------------------------------------
        1         0.2752 (27)         0.2752 (20)
        2        0.07951 (80)        0.07952 (58)
        3        0.02891 (29)        0.02892 (21)
        4        0.01127 (11)       0.011272 (83)
        5       0.004502 (46)       0.004506 (34)
        6       0.001817 (19)       0.001819 (14)
        7      0.0007362 (79)      0.0007375 (57)
        8      0.0002987 (33)      0.0002994 (24)
        9      0.0001213 (14)     0.00012163 (99)
       10     0.00004926 (57)     0.00004943 (41)
       12        8.13(10)e-06        8.164(72)e-06
       14       1.342(19)e-06       1.348(13)e-06
       16       2.217(37)e-07       2.227(23)e-07
       18       3.661(85)e-08       3.679(40)e-08
       20        6.24(61)e-09        6.078(71)e-09
```

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



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

```
import pylab as plt
ratio = y / f(x, fit.pmean)
plt.xlim(0, 21)
plt.xlabel('x')
plt.ylabel('y/f(x,p)')
plt.errorbar(x=x, y=gv.mean(ratio), yerr=gv.sdev(ratio), fmt='ob')
plt.plot([0.0, 21.0], [1.0, 1.0])
plt.show()
```

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

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

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

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

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

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

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

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

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

1.4 Chained Fits

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

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

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

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

newfit = lsqfit.nonlinear_fit(data=data, fcn=ratio, prior=prior)
print(newfit)
```

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

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

Parameters:
            a 0    0.4018 (40)      [  0.4018 (40) ]
              1    0.4018 (40)      [  0.4033 (49) ]
              2     0.421 (20)      [   0.447 (51) ]
              3      0.53 (17)      [    0.60 (21) ]
              4      0.46 (34)      [    0.38 (37) ]
              5      0.50 (42)      [    0.40 (46) ]
              6      0.50 (48)      [    0.45 (49) ]
              7      0.50 (50)      [    0.48 (50) ]
              8      0.50 (50)      [    0.49 (50) ]
              9      0.50 (50)      [    0.50 (50) ]
             10      0.50 (50)      [    0.50 (50) ]
             11      0.50 (50)      [    0.50 (50) ]
             12      0.50 (50)      [    0.50 (50) ]
             13      0.50 (50)      [    0.50 (50) ]
             14      0.50 (50)      [    0.50 (50) ]
             15      0.50 (50)      [    0.50 (50) ]
             16      0.50 (50)      [    0.50 (50) ]
             17      0.50 (50)      [    0.50 (50) ]
             18      0.50 (50)      [    0.50 (50) ]
            E 0   0.90030 (51)      [ 0.90035 (51) ]
```

```
 1   1.80007 (67)       [  1.8011 (19) ]
 2     2.711 (12)       [   2.734 (42) ]
 3      3.76 (18)       [    3.91 (33) ]
 4      5.02 (48)       [    4.97 (49) ]
 5      6.00 (50)       [    6.02 (50) ]
 6      7.00 (50)       [    7.02 (50) ]
 7      8.00 (50)       [    8.01 (50) ]
 8      9.00 (50)       [    9.00 (50) ]
 9     10.00 (50)       [   10.00 (50) ]
10     11.00 (50)       [   11.00 (50) ]
11     12.00 (50)       [   12.00 (50) ]
12     13.00 (50)       [   13.00 (50) ]
13     14.00 (50)       [   14.00 (50) ]
14     15.00 (50)       [   15.00 (50) ]
15     16.00 (50)       [   16.00 (50) ]
16     17.00 (50)       [   17.00 (50) ]
17     18.00 (50)       [   18.00 (50) ]
18     19.00 (50)       [   19.00 (50) ]

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

Parameters `a[0]` and `E[0]` are essentially unchanged by the new information, but `a[i]` and `E[i]` are more precise for `i=1`, `2` and `3`, as is `a[1]/a[0]`, of course. It might seem odd that `E[1]`, for example, is changed at all, since the fit function, `ratio(p)`, makes no mention of it. This is not surprising, however, since `ratio(p)` does depend upon `a[1]`, and `a[1]` is strongly correlated with `E[1]` through the prior. It is important to include all parameters from the first fit as parameters in the new fit in order to capture the impact of the new information on parameters correlated with `a[1]/a[0]`.

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

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

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

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

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

gives the following output:

```
chi2/dof = 0.32

E: [0.90030(51) 1.80007(67) 2.711(12) 3.76(18)]
a: [0.4018(41) 0.4018(40) 0.421(20) 0.53(17)]
a1/a0: 1.000000(10)
```

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

1.5 `x` has Error Bars

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

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

```
def make_data(nexp=100, eps=0.01): # make x, y fit data
    x = np.array([1.,2.,3.,4.,5.,6.,7.,8.,9.,10.,12.,14.,16.,18.,20.])
    cr = gv.gvar(0.0, eps)
    c = [gv.gvar(cr(), eps) for n in range(100)]
    x_xmax = x/max(x)
    noise = 1+ sum(c[n] * x_xmax ** n for n in range(100))
    y = f_exact(x, nexp) * noise            # noisy y[i]s
    xfac = gv.gvar(1.0, 0.00001)    # Gaussian distrib'n: 1±0.001%
    x = np.array([xi * gv.gvar(xfac(), xfac.sdev) for xi in x]) # noisy x[i]s
    return x, y
```

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

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

def main():
    gv.ranseed([2009, 2010, 2011, 2012]) # initialize random numbers (opt.)
    x, y = make_data()              # make fit data
    p0 = None                       # make larger fits go faster (opt.)
    for nexp in range(3, 20):
        print('********************************** nexp =', nexp)
        prior = make_prior(nexp, x)
        fit = lsqfit.nonlinear_fit(data=y, fcn=f, prior=prior, p0=p0)
        print(fit)                  # print the fit results
        E = fit.p['E']              # best-fit parameters
        a = fit.p['a']
        print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
        print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
        print()
```

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

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

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

Running the new code gives, for `nexp=6` terms:

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

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

Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0    (itns/time = 6/0.0)

E1/E0 = 2.018(22)   E2/E0 = 3.27(31)
a1/a0 = 1.065(77)   a2/a0 = 1.45(57)
```

This looks quite a bit like what we obtained before, except that now there are 15 more parameters, one for each `x`, and also now all results are a good deal less accurate. Note that one result from this analysis is new values for the `x`s. In some cases (*e.g.*, `x[7]`), the errors on the `x` values have been reduced — by information in the fit data.

1.6 Correlated Parameters; Gaussian Bayes Factor

`gvar.GVar` objects are very useful for handling more complicated priors, including situations where we know *a priori* of correlations between parameters. Returning to the *Basic Fits* example above, imagine a situation where we still have a ± 0.5 uncertainty about the value of any individual `E[i]`, but we know *a priori* that the separations between adjacent `E[i]`s is 0.9 ± 0.01. We want to build the correlation between adjacent `E[i]`s into our prior.

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

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

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

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

For `nexp=3`, this implies that

```
>>> print(E)
[1.00(50) 1.90(50) 2.80(50)]
>>> print(E[1] - E[0], E[2] - E[1])
0.900(10) 0.900(10)
```

which shows that each `E[i]` separately has an uncertainty of ± 0.5 (approximately) but that differences are specified to within ± 0.01.

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

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

Running the code as before, but now with the correlated prior in place, we obtain the following fit with `nexp=7` terms:

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

Parameters:
            a 0    0.4018 (40)      [  0.50 (50) ]
              1    0.4016 (42)      [  0.50 (50) ]
              2     0.404 (12)      [  0.50 (50) ]
              3     0.394 (46)      [  0.50 (50) ]
              4      0.40 (16)      [  0.50 (50) ]
              5      0.51 (31)      [  0.50 (50) ]
              6      0.52 (42)      [  0.50 (50) ]
            E 0   0.90032 (51)      [  1.00 (50) ]
              1    1.8001 (11)      [  1.90 (50) ]
              2     2.701 (10)      [  2.80 (50) ]
              3     3.601 (14)      [  3.70 (50) ]
              4     4.501 (17)      [  4.60 (50) ]
              5     5.401 (20)      [  5.50 (50) ]
```

```
              6     6.301 (22)      [  6.40 (50) ]

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

E1/E0 = 1.9994(11)   E2/E0 = 3.000(11)
a1/a0 = 0.9996(25)   a2/a0 = 1.005(28)
```

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

Note that the Gaussian Bayes Factor (see `logGBF` in the output) is significantly larger with the correlated prior (`logGBF = 227`) than it was for the uncorrelated prior (`logGBF = 221`). Had we been uncertain as to which prior was more appropriate, this difference says that the data prefers the correlated prior. (More precisely, it says that we would be `exp(227-221) = 400` times more likely to get our `x,y` data from a theory with the correlated prior than from one with the uncorrelated prior.) This difference is significant despite the fact that the `chi**2`s in the two cases are almost the same. `chi**2` tests goodness of fit, but there are usually more ways than one to get a good fit. Some are more plausible than others, and the Bayes factor helps sort out which.

1.7 Tuning Priors and the Empirical Bayes Criterion

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

This method is implemented in a driver program

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

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

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

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

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

```
def fitargs(z, nexp=nexp, prior=prior, f=f, data=(x, y), p0=p0):
    z = np.exp(z)
    prior['a'] = [gv.gvar(0.5, 0.5 * z[0]) for i in range(nexp)]
    return dict(prior=prior, data=data, fcn=f, p0=p0)
##
z0 = [0.0]
fit, z = empbayes_fit(z0, fitargs, tol=1e-3)
print(fit)                  # print the optimized fit results
```

```
E = fit.p['E']              # best-fit parameters
a = fit.p['a']
print('E1/E0 =', E[1]  /  E[0], '  E2/E0 =', E[2]  /  E[0])
print('a1/a0 =', a[1]  /  a[0], '  a2/a0 =', a[2]  /  a[0])
print("prior['a'] =", fit.prior['a'][0])
print()
```

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

This code generates the following output when `nexp=7`:

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

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

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

E1/E0 = 1.9993(10)   E2/E0 = 3.0015(94)
a1/a0 = 0.9995(25)   a2/a0 = 1.011(17)
prior['a'] = 0.500(95)
```

Reducing the width of the `prior['a']`s from 0.5 to 0.1 increased `logGBF` from 227 to 234. The error for `a2/a0` is 40% smaller, but the other results are not much affected — suggesting that the details of `prior['a']` are not all that important, which is confirmed by the error budgets generated in the next section. It is not surprising, of course, that the optimal width is 0.1 since the mean values for the `fit.p['a']`s are clustered around 0.4, which is 0.1 below the mean value of the priors `prior['a']`.

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

```
********************************** nexp = 2
Least Square Fit:
  chi2/dof [dof] = 0.47 [15]    Q = 0.96    logGBF = 254.15

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

```
Settings:
  svdcut/n = 1e-15/2    reltol/abstol = 0.0001/0    (itns/time = 6/0.0)


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

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

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

Measured by their `chi**2`s, the two fits are almost equally good. The Bayes factor for the first fit, however, is much larger than that for the second fit. It says that the probability that our fit data comes from an underlying theory with exactly two terms is `exp(254 - 243) = 59,874` times larger than the probability that it comes from a theory with three terms. In fact, the data comes from a theory with only two terms since it was generated using the same code as in the previous section but with `x, y = make_data(2)` instead of `x, y = make_data()` in the main program.

1.8 Partial Errors and Error Budgets

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

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

```
>>> E = fit.p['E']
>>> a = fit.p['a']
>>> print(E[1] / E[0])
1.9994(11)
>>> print((E[1] / E[0]).partialsdev(fit.prior['E']))
0.000419224371208
>>> print((E[1] / E[0]).partialsdev(fit.prior['a']))
0.000158887614987
>>> print((E[1] / E[0]).partialsdev(y))
0.000953276447811
```

This shows that the total uncertainty in `E[1]/E[0]`, 0.00106, is the sum in quadrature of a contribution 0.00042 due to the priors specified by `prior['E']`, 0.00016 due to `prior['a']`, and 0.00095 from the statistical errors in the input data `y`.

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

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

This gives the following output:

```
Values:
              E2/E0: 3.000(11)
              E1/E0: 1.9994(11)
              a2/a0: 1.005(28)
              a1/a0: 0.9996(25)

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

This table shows, for example, that the 0.37% uncertainty in `E2/E0` comes from a 0.09% contribution due to `prior['a']`, a 0.07% contribution due to due to statistical errors in the fit data `y`, and a 0.35% contribution due to `prior['E']`, where, again, the total error is the sum in quadrature of the partial errors. This suggests that reducing the statistical errors in the input `y` data would reduce the error in `E2/E0` only slightly. On the other hand, more accurate `y` data should significantly reduce the errors in `E1/E0` and `a1/a0`, where `y` is the dominant source of uncertainty. In fact a four-fold reduction in the `y` errors reduces the `E1/E0` error to 0.02% (from 0.05%) while leaving the `E2/E0` error at 0.37%.

1.9 `y` has No Error Bars

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

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

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

We know that for some set of parameters

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

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

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

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

It turns out to be quite simple to implement such a strategy using gvar.GVars. We convert our code by first modifying the main program so that it provides prior information to a subroutine that computes ymod. We will vary the number of terms nexp that are kept in the fit, putting the rest into ymod as above (up to a maximum of 20 terms, which is close enough to infinity):

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

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

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

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

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

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

Fit:
     x[k]           y[k]        f(x[k],p)
```

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

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


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

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

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

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


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

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

Fit:
     x[k]             y[k]        f(x[k],p)
```

```
-----------------------------------------
        1       0.253 (19)      0.2557 (54)
      1.2      0.1968 (91)      0.1977 (28)
      1.4      0.1545 (45)      0.1548 (15)
      1.6      0.1224 (22)     0.12256 (76)
      1.8      0.0979 (11)     0.09793 (39)
        2     0.07885 (54)     0.07886 (20)
      2.2     0.06391 (27)     0.06391 (10)
      2.4     0.05206 (13)    0.052065 (52)
      2.6    0.042602 (67)    0.042601 (26)
      2.8    0.034983 (33)    0.034982 (13)

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


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

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

Fit:
     x[k]              y[k]         f(x[k],p)
-----------------------------------------------
        1       0.2656 (78)       0.2666 (22)
      1.2       0.2027 (32)      0.20297 (97)
      1.4       0.1573 (13)      0.15737 (42)
      1.6      0.12378 (54)      0.12381 (18)
      1.8      0.09853 (22)     0.098540 (79)
        2     0.079153 (93)     0.079155 (34)
      2.2     0.064051 (39)     0.064051 (15)
      2.4     0.052134 (16)    0.0521344 (66)
      2.6    0.0426348 (67)    0.0426347 (30)
      2.8    0.0349985 (28)    0.0349985 (14)

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


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

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

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

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

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

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

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

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

1.10 SVD Cuts and Roundoff Error

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

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

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

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

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

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

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

The inverse of the `y-prior` covariance matrix is used in the `chi**2` function that is minimized by `lsqfit.nonlinear_fit`. Given the finite precision of computer hardware, it is impossible to compute this inverse accurately if the matrix is singular or almost singular, and in such situations the reliability of the fit results is in question. The eigenvalues of the covariance matrix in this example (for `nexp=6`) indicate that this is the case: they range from `7.2e-5` down to `4.2e-26`, covering 21 orders of magnitude. This is likely too large a range to be handled with the 16–18 digits of precision available in normal double precision computation. The smallest eigenvalues and their eigenvectors are likely to be quite inaccurate, as is any method for computing the inverse matrix.

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

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

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

The SVD cut is actually applied to the correlation matrix, which is the covariance matrix rescaled by standard deviations so that all diagonal elements equal 1. This helps mitigate problems caused by large scale differences between different variables. Any eigenvalue smaller than `svdcut` times the largest eigenvalue is replaced by `svdcut` times the largest eigenvalue. Thus larger values for `svdcut` affect larger numbers of eigenmodes and increase errors in the final results.

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

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

which gives:

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

Partial % Errors:
                        E2/E0     E1/E0     a2/a0     a1/a0
-------------------------------------------------------------
                 a:      3.73      0.70     11.75      4.35
```

```
                svd:      0.29      0.10       0.13      0.55
                  E:      5.90      1.00      22.43      6.33
-------------------------------------------------------------
              total:      6.99      1.23      25.32      7.70
```

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

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

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

gives a new, non-singular matrix:

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

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

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

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

applies different SVD cuts to the prior and data.

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

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

1.11 Bootstrap Error Analysis

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

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

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

```
Nbs = 40                                    # number of bootstrap copies
outputs = {'E1/E0':[], 'E2/E0':[], 'a1/a0':[], 'a2/a0':[]}   # results
for bsfit in fit.bootstrap_iter(n=Nbs):
    E = bsfit.pmean['E']                    # best-fit parameter values
    a = bsfit.pmean['a']                    #   (ignore errors)
    outputs['E1/E0'].append(E[1] / E[0])      # accumulate results
    outputs['E2/E0'].append(E[2] / E[0])
    outputs['a1/a0'].append(a[1] / a[0])
    outputs['a2/a0'].append(a[2] / a[0])
    outputs['E1'].append(E[1])
    outputs['a1'].append(a[1])
# extract "means" and "standard deviations" from the bootstrap output;
# print using .fmt() to create compact representation of GVars
outputs = gv.dataset.avg_data(outputs, bstrap=True)
print('Bootstrap results:')
print('E1/E0 =', outputs['E1/E0'].fmt(), '  E2/E1 =', outputs['E2/E0'].fmt())
print('a1/a0 =', outputs['a1/a0'].fmt(), '  a2/a0 =', outputs['a2/a0'].fmt())
print('E1 =', outputs['E1'].fmt(), '  a1 =', outputs['a1'].fmt())
```

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

```
Bootstrap results:
E1/E0 = 1.998(18)   E2/E1 = 2.96(16)
a1/a0 = 0.988(60)   a2/a0 = 0.93(22)
E1 = 1.798(16)   a1 = 0.395(25)
```

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

1.12 Testing Fits with Simulated Data

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

```
def main():
    gv.ranseed([2009, 2010, 2011, 2012]) # initialize random numbers (opt.)
    x, y = make_data()                  # make fit data
    p0 = None                           # make larger fits go faster (opt.)
    for nexp in range(3, 20):
        print('************************************* nexp =', nexp)
        prior = make_prior(nexp)
        fit = lsqfit.nonlinear_fit(data=(x, y), fcn=f, prior=prior, p0=p0)
        print(fit)                      # print the fit results
        E = fit.p['E']                  # best-fit parameters
        a = fit.p['a']
        print('E1/E0 =', E[1] / E[0], '  E2/E0 =', E[2] / E[0])
        print('a1/a0 =', a[1] / a[0], '  a2/a0 =', a[2] / a[0])
        print()
        if fit.chi2 / fit.dof < 1.:
            p0 = fit.pmean              # starting point for next fit (opt.)
```

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

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

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

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

```
************************************ simulation
Least Square Fit:
  chi2/dof [dof] = 0.43 [15]    Q = 0.97    logGBF = 227.47

Parameters:
            a 0    0.4064 (40)      [  0.50 (50) ]
              1    0.4049 (42)      [  0.50 (50) ]
              2     0.414 (12)      [  0.50 (50) ]
              3     0.354 (46)      [  0.50 (50) ]
              4      0.57 (16)      [  0.50 (50) ]
              5      0.35 (31)      [  0.50 (50) ]
              6      0.38 (42)      [  0.50 (50) ]
            E 0   0.90123 (50)      [  1.00 (50) ]
              1    1.7997 (11)      [  1.90 (50) ]
              2     2.699 (10)      [  2.80 (50) ]
              3     3.599 (14)      [  3.70 (50) ]
              4     4.499 (17)      [  4.60 (50) ]
              5     5.399 (20)      [  5.50 (50) ]
              6     6.299 (22)      [  6.40 (50) ]
```

```
Settings:
  svdcut/n = None/0    reltol/abstol = 0.0001/0    (itns/time = 40/0.0)

E1/E0 = 1.9969(11)   E2/E0 = 2.995(11)
a1/a0 = 0.9963(26)   a2/a0 = 1.019(28)

Simulated Fit Values - Exact Values:
E1/E0: -0.0025(11)   E2/E0: -0.005(11)
a1/a0: -0.0033(26)   a2/a0: 0.014(28)

Parameter chi2/dof [dof] = 2.12 [4]   Q = 0.1

************************************ simulation
Least Square Fit:
  chi2/dof [dof] = 0.26 [15]    Q = 1    logGBF = 228.77

Parameters:
            a 0    0.4013 (40)      [  0.50 (50) ]
              1    0.3992 (42)      [  0.50 (50) ]
              2     0.410 (12)      [  0.50 (50) ]
              3     0.344 (46)      [  0.50 (50) ]
              4      0.58 (16)      [  0.50 (50) ]
              5      0.33 (31)      [  0.50 (50) ]
              6      0.37 (42)      [  0.50 (50) ]
            E 0   0.90029 (51)      [  1.00 (50) ]
              1    1.7980 (11)      [  1.90 (50) ]
              2     2.696 (10)      [  2.80 (50) ]
              3     3.596 (14)      [  3.70 (50) ]
              4     4.496 (17)      [  4.60 (50) ]
              5     5.396 (20)      [  5.50 (50) ]
              6     6.296 (22)      [  6.40 (50) ]

Settings:
  svdcut/n = None/0    reltol/abstol = 0.0001/0    (itns/time = 50/0.1)

E1/E0 = 1.9971(11)   E2/E0 = 2.995(11)
a1/a0 = 0.9949(26)   a2/a0 = 1.021(28)

Simulated Fit Values - Exact Values:
E1/E0: -0.0023(11)   E2/E0: -0.005(11)
a1/a0: -0.0047(26)   a2/a0: 0.016(28)

Parameter chi2/dof [dof] = 1.73 [4]   Q = 0.1

************************************ simulation
Least Square Fit:
  chi2/dof [dof] = 0.82 [15]    Q = 0.66    logGBF = 224.63

Parameters:
            a 0    0.4016 (40)      [  0.50 (50) ]
              1    0.4028 (41)      [  0.50 (50) ]
              2     0.397 (12)      [  0.50 (50) ]
              3     0.438 (45)      [  0.50 (50) ]
              4      0.24 (15)      [  0.50 (50) ]
              5      0.65 (31)      [  0.50 (50) ]
              6      0.64 (42)      [  0.50 (50) ]
            E 0   0.90021 (51)      [  1.00 (50) ]
              1    1.8015 (11)      [  1.90 (50) ]
```

```
              2     2.703 (10)       [  2.80 (50) ]
              3     3.603 (14)       [  3.70 (50) ]
              4     4.503 (17)       [  4.60 (50) ]
              5     5.403 (20)       [  5.50 (50) ]
              6     6.303 (22)       [  6.40 (50) ]

Settings:
  svdcut/n = None/0    reltol/abstol = 0.0001/0    (itns/time = 36/0.0)

E1/E0 = 2.0012(11)   E2/E0 = 3.003(11)
a1/a0 = 1.0032(25)   a2/a0 = 0.989(28)

Simulated Fit Values - Exact Values:
E1/E0: 0.0018(11)   E2/E0: 0.003(11)
a1/a0: 0.0036(25)   a2/a0: -0.016(28)

Parameter chi2/dof [dof] = 1.02 [4]   Q = 0.4
```

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

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

1.13 Positive Parameters

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

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

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

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

```
@lsqfit.transform_p(prior.keys(), has_x=True)
def fitfcn(x, p):
    ...
```

when the parameter argument is the second argument of the fit function. Consider any parameter `p['XX']` used in `fitfcn`. The prior distribution for that parameter can now be turned into a log-normal distribution by replacing `prior['XX']` with `prior['logXX']` (or `prior['log(XX)']`) when defining the prior, thereby assigning

a Gaussian distribution to `logXX` rather than to `XX`. Nothing need be changed in the fit function, other than adding the decorator. The decorator automatically detects parameters whose keys begin with `'log'` and adds new parameters to the parameter-dictionary for `fitfcn` that are exponentials of those parameters.

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

```
import gvar as gv
import lsqfit

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

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

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

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

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

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

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

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

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

a = 0.004(18)
```

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

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

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

@lsqfit.transform_p(prior.keys(), 0)
def fcn(p, N=len(y)):
   return N * [p['a']]

fit = lsqfit.nonlinear_fit(prior=prior, data=y, fcn=fcn)
print(fit)
print('a =', fit.transformed_p['a'].fmt())     # exp(loga)
```

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

The result from this fit is

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

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

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

a = 0.012(11)
```

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

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

1.14 Troubleshooting

`lsqfit.nonlinear_fit` error messages that come from inside the *gsl* routnines doing the fits are sometimes less than useful. They are usually due to errors in one of the inputs to the fit (that is, the fit data, the prior, or the fit function). Setting `debug=True` in the argument list of `lsqfit.nonlinear_fit` might result in more intelligible error messages. This option also causes the fitter to check for significant roundoff errors in the matrix inversions of the covariance matrices.

Occasionally `lsqfit.nonlinear_fit` appears to go crazy, with gigantic `chi**2`s (*e.g.*, `1e78`). This could be because there is a genuine zero-eigenvalue mode in the covariance matrix of the data or prior. Such a zero mode makes it impossible to invert the covariance matrix when evaluating `chi**2`. One fix is to include SVD cuts in the fit by setting, for example, `svdcut=(1e-14,1e-14)` in the call to `lsqfit.nonlinear_fit`. These cuts will exclude exact or nearly exact zero modes, while leaving important modes mostly unaffected.

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

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

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

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

1.15 Appendix: A Simple Pedagogical Example

1.15.1 A Bad Solution

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

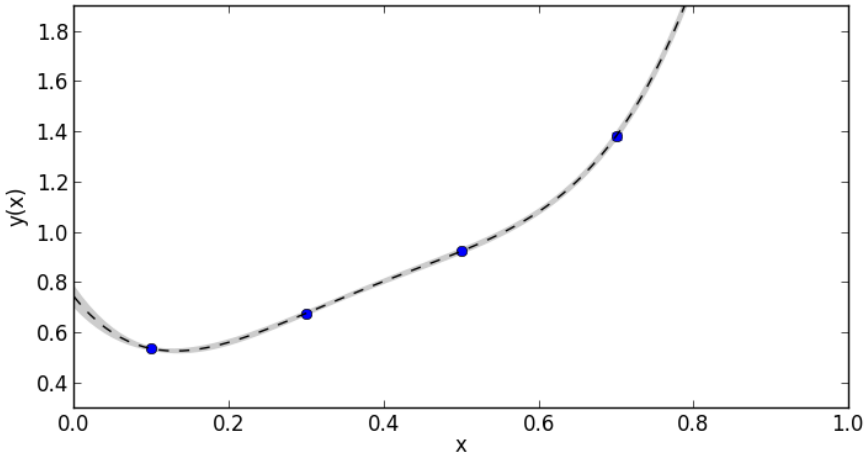
```
x[i]        y(x[i])
---------------------
0.1         0.5351 (54)
0.3         0.6762 (67)
0.5         0.9227 (91)
0.7         1.3803(131)
0.95        4.0145(399)
```

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 extrapolate the data to `x=0`.

One approach that is certainly wrong is to truncate the expansion of `y(x)` after five terms, because there are only five pieces of data. That 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.GVar`s. Running the code gives the following output:

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

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

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

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

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

1.15.2 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
```

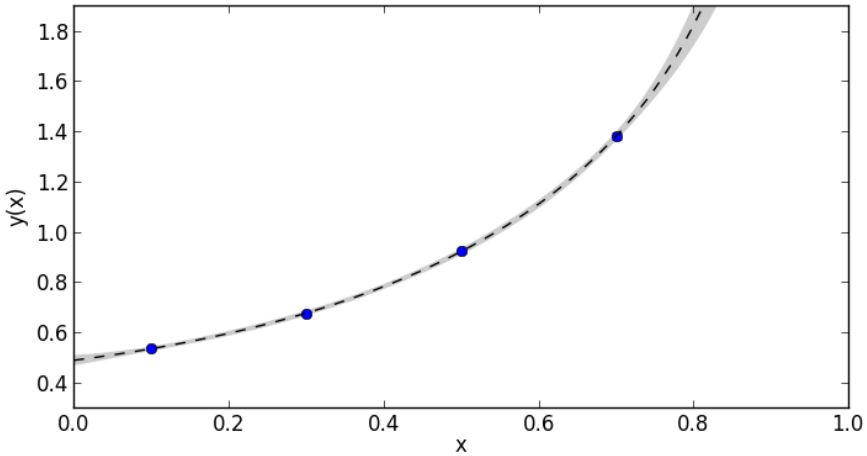
with `N=90`.

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)`. So we are actually fitting 91+5 pieces of data with 91 parameters.

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

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

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

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

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

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

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



The fit code output is:

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

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

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

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

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

1.15.3 Bayes Factor

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

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

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

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

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

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

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

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).

1.15.4 Marginalization

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

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

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

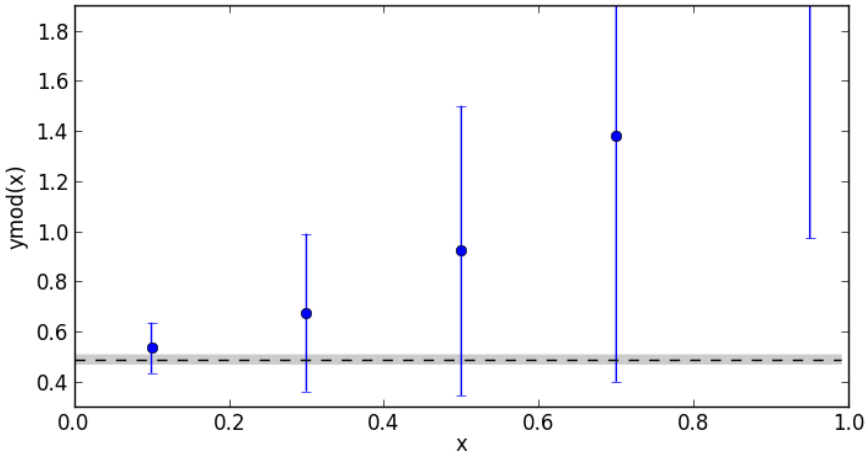
```
import numpy as np
import gvar as gv
import lsqfit

# fit data
```

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

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

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

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

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

Running this code give:

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

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

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

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

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



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

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

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

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

CHAPTER

TWO

GVAR - GAUSSIAN RANDOM VARIABLES

2.1 Introduction

This module provides tools for representing and manipulating Gaussian random variables numerically. It can deal with individual variables or arbitrarily large sets of variables, correlated or uncorrelated. It also supports complicated (Python) functions of Gaussian variables.

A Gaussian variable is a random variable that represents a *typical* random number drawn from a particular Gaussian (or normal) probability distribution; more precisely, it represents the entire probability distribution. A given Gaussian variable `x` is therefore completely characterized by its mean `x.mean` and standard deviation `x.sdev`.

A mathematical function of a Gaussian variable can be defined as the probability distribution of function values obtained by evaluating the function for random numbers drawn from the original distribution. The distribution of function values is itself approximately Gaussian provided the standard deviation of the Gaussian variable is sufficiently small. Thus we can define a function `f` of a Gaussian variable `x` to be a Gaussian variable itself, with

```
f(x).mean = f(x.mean)
f(x).sdev = x.sdev |f'(x.mean)|,
```

which follows from linearizing the `x` dependence of `f(x)` about point `x.mean`. (This obviously fails at an extremum of `f(x)`, where `f'(x)=0`.)

The last formula, together with its multidimensional generalization, leads to a full calculus for Gaussian random variables that assigns Gaussian-variable values to arbitrary arithmetic expressions and functions involving Gaussian variables. A multidimensional collection `x[i]` of Gaussian variables is characterized by the means `x[i].mean` for each variable, together with a covariance matrix `cov[i, j]`. Diagonal elements of `cov` specify the standard deviations of different variables: `x[i].sdev = cov[i, i]**0.5`. Nonzero off-diagonal elements imply correlations between different variables:

```
cov[i, j] = <x[i]*x[j]>  -  <x[i]> * <x[j]>
```

where `<y>` denotes the expectation value or mean for a random variable `y`.

Gaussian variables are represented in `gvar` by objects of type `gvar.GVar`. `gvar.GVar`s are particularly useful for error propagation because the errors (i.e., standard deviations) associated with them propagate through arbitrarily complex Python functions. This is true of errors from individual `gvar.GVar`s, and also of the correlations between different `gvar.GVar`s. In the following sections we demonstrate how to create `gvar.GVar`s and how to manipulate them to solve common problems in error propagation. They are also very useful for creating correlated Gaussian priors for Bayesian analyses.

2.2 Creating Gaussian Variables

Objects of type gvar.GVar are of two types: 1) primary gvar.GVars that are created from means and covariances using gvar.gvar(); and 2) derived gvar.GVars that result from arithmetic expressions or functions involving gvar.GVars (primary or derived). A single (primary) gvar.GVar is created from its mean xmean and standard deviation xsdev using:

```
x = gvar.gvar(xmean, xsdev).
```

This function can also be used to convert strings like "-72.374(22)" or "511.2 +- 0.3" into gvar.GVars: for example,

```
>>> import gvar
>>> x = gvar.gvar(3.1415, 0.0002)
>>> print(x)
3.14150(20)
>>> x = gvar.gvar("3.1415(2)")
>>> print(x)
3.14150(20)
>>> x = gvar.gvar("3.1415 +- 0.0002")
>>> print(x)
3.14150(20)
```

Note that x = gvar.gvar(x) is useful when you are unsure whether x is a gvar.GVar or a string representing a gvar.GVar.

gvar.GVars are usually more interesting when used to describe multidimensional distributions, especially if there are correlations between different variables. Such distributions are represented by collections of gvar.GVars in one of two standard formats: 1) numpy arrays of gvar.GVars (any shape); or, more flexibly, 2) Python dictionaries whose values are gvar.GVars or arrays of gvar.GVars. Most functions in gvar that handle multiple gvar.GVars work with either format, and if they return multidimensional results do so in the same format as the inputs (that is, arrays or dictionaries). Any dictionary is converted internally into a specialized (ordered) dictionary of type gvar.BufferDict, and dictionary-valued results are also gvar.BufferDicts.

To create an array of gvar.GVars with mean values specified by array xmean and covariance matrix xcov, use

```
x = gvar.gvar(xmean, xcov)
```

where array x has the same shape as xmean (and xcov.shape = xmean.shape+xmean.shape). Then each element x[i] of a one-dimensional array, for example, is a gvar.GVar where:

```
x[i].mean = xmean[i]          # mean of x[i]
x[i].val  = xmean[i]          # same as x[i].mean
x[i].sdev = xcov[i, i]**0.5   # std deviation of x[i]
x[i].var  = xcov[i, i]        # variance of x[i]
```

As an example,

```
>>> x, y = gvar.gvar([0.1, 10.], [[0.015625, 0.24], [0.24, 4.]])
>>> print('x =', x, '   y =', y)
x = 0.10(13)     y = 10.0(2.0)
```

makes x and y gvar.GVars with standard deviations sigma_x=0.125 and sigma_y=2, and a fairly strong statistical correlation:

```
>>> print(gvar.evalcov([x, y]))      # covariance matrix
[[ 0.015625  0.24    ]
 [ 0.24      4.      ]]
>>> print(gvar.evalcorr([x, y]))     # correlation matrix
```

```
[[ 1.    0.96]
 [ 0.96  1.  ]]
```

Here functions `gvar.evalcov()` and `gvar.evalcorr()` compute the covariance and correlation matrices, respectively, of the list of `gvar.GVar`s in their arguments.

`gvar.gvar()` can also be used to convert strings or tuples stored in arrays or dictionaries into `gvar.GVar`s: for example,

```
>>> garray = gvar.gvar(['2(1)', '10+-5', (99, 3), gvar.gvar(0, 2)])
>>> print(garray)
[2.0(1.0) 10.0(5.0) 99.0(3.0) 0.0(2.0)]
>>> gdict = gvar.gvar(dict(a='2(1)', b=['10+-5', (99, 3), gvar.gvar(0, 2)]))
>>> print(gdict)
{'a': 2.0(1.0),'b': array([10.0(5.0), 99.0(3.0), 0.0(2.0)], dtype=object)}
```

If the covariance matrix in `gvar.gvar` is diagonal, it can be replaced by an array of standard deviations (square roots of diagonal entries in `cov`). The example above without correlations, therefore, would be:

```
>>> x, y = gvar.gvar([0.1, 10.], [0.125, 2.])
>>> print('x =', x, '   y =', y)
x = 0.10(12)    y = 10.0(2.0)
>>> print(gvar.evalcov([x, y]))      # covariance matrix
[[ 0.015625  0.      ]
 [ 0.        4.      ]]
>>> print(gvar.evalcorr([x, y]))     # correlation matrix
[[ 1.  0.]
 [ 0.  1.]]
```

2.3 `gvar.GVar` Arithmetic and Functions

The `gvar.GVar`s discussed in the previous section are all *primary* `gvar.GVar`s since they were created by specifying their means and covariances explicitly, using `gvar.gvar()`. What makes `gvar.GVar`s particularly useful is that they can be used in arithemtic expressions (and numeric pure-Python functions), just like Python floats. Such expressions result in new, *derived* `gvar.GVar`s whose means, standard deviations, and correlations are determined from the covariance matrix of the primary `gvar.GVar`s. The automatic propagation of correlations through arbitrarily complicated arithmetic is an especially useful feature of `gvar.GVar`s.

As an example, again define

```
>>> x, y = gvar.gvar([0.1, 10.], [0.125, 2.])
```

and set

```
>>> f = x + y
>>> print('f =', f)
f = 10.1(2.0)
```

Then `f` is a (derived) `gvar.GVar` whose variance `f.var` equals

```
df/dx cov[0, 0] df/dx + 2 df/dx cov[0, 1] df/dy + ... = 2.0039**2
```

where `cov` is the original covariance matrix used to define `x` and `y` (in `gvar.gvar`). Note that while `f` and `y` separately have 20% uncertainties in this example, the ratio `f/y` has much smaller errors:

```
>>> print(f / y)
1.010(13)
```

This happens, of course, because the errors in `f` and `y` are highly correlated — the error in `f` comes mostly from `y`. `gvar.GVar`s automatically track correlations even through complicated arithmetic expressions and functions: for example, the following more complicated ratio has a still smaller error, because of stronger correlations between numerator and denominator:

```
>>> print(gvar.sqrt(f**2 + y**2) / f)
1.4072(87)
>>> print(gvar.evalcorr([f, y]))
[[ 1.          0.99805258]
 [ 0.99805258  1.        ]]
>>> print(gvar.evalcorr([gvar.sqrt(f**2 + y**2), f]))
[[ 1.         0.9995188]
 [ 0.9995188  1.       ]]
```

The `gvar` module defines versions of the standard Python mathematical functions that work with `gvar.GVar` arguments. These include: `exp`, `log`, `sqrt`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, `arctanh`. Numeric functions defined entirely in Python (*i.e.*, pure-Python functions) will likely also work with `gvar.GVar`s.

Numeric functions implemented by modules using low-level languages like C will *not* work with `gvar.GVar`s. Such functions must be replaced by equivalent code written directly in Python. In some cases it is possible to construct a `gvar.GVar`-capable function from low-level code for the function and its derivative. For example, the following code defines a new version of the standard Python error function that accepts either floats or `gvar.GVar`s as its argument:

```
import math
import gvar

def erf(x):
    if isinstance(x, gvar.GVar):
        f = math.erf(x.mean)
        dfdx = 2. * math.exp(- x.mean ** 2) / math.sqrt(math.pi)
        return gvar.gvar_function(x, f, dfdx)
    else:
        return math.erf(x)
```

Here function `gvar.gvar_function()` creates the `gvar.GVar` for a function with mean value `f` and derivative `dfdx` at point `x`.

Some sample numerical analysis codes, adapted for use with `gvar.GVar`s, are described in *Numerical Analysis Modules in gvar*.

Arithmetic operators `+ - * / ** == != <> += -= *= /=` are all defined for `gvar.GVar`s. Comparison operators are also supported: `== != > >= < <=`. They are applied to the mean values of `gvar.GVar`s: for example, `gvar.gvar(1,1) == gvar.var(1,2)` is true, as is `gvar.gvar(1,1) > 0`. Logically `x>y` for `gvar.GVar`s should evaluate to a boolean-valued random variable, but such variables are beyond the scope of this module. Comparison operators that act only on the mean values make it easier to implement pure-Python functions that work with either `gvar.GVar`s or `float`s as arguments.

Implementation Notes: Each `gvar.GVar` keeps track of three pieces of information: 1) its mean value; 2) its derivatives with respect to the primary `gvar.GVar`s (created by `gvar.gvar()`); and 3) the location of the covariance matrix for the primary `gvar.GVar`s. The derivatives and covariance matrix allow one to compute the standard deviation of the `gvar.GVar`, as well as correlations between it and any other function of the primary `gvar.GVar`s. The derivatives for derived `gvar.GVar`s are computed automatically, using *automatic differentiation*.

The derivative of a `gvar.GVar` `f` with respect to a primary `gvar.GVar` `x` is obtained from `f.deriv(x)`. A list of derivatives with respect to all primary `gvar.GVar`s is given by `f.der`, where the order of derivatives is the same as the order in which the primary `gvar.GVar`s were created.

A gvar.GVar can be constructed at a very low level by supplying all the three essential pieces of information — for example,

```
f = gvar.gvar(fmean, fder, cov)
```

where fmean is the mean, fder is an array where fder[i] is the derivative of f with respect to the i-th primary gvar.GVar (numbered in the order in which they were created using gvar.gvar()), and cov is the covariance matrix for the primary gvar.GVars (easily obtained from an existing gvar.GVar x using x.cov).

2.4 Error Budgets from gvar.GVars

It is sometimes useful to know how much of the uncertainty in a derived quantity is due to a particular input uncertainty. Continuing the example above, for example, we might want to know how much of fs standard deviation is due to the standard deviation of x and how much comes from y. This is easily computed:

```
>>> x, y = gvar.gvar([0.1, 10.], [0.125, 2.])
>>> f = x + y
>>> print(f.partialsdev(x))         # uncertainty in f due to x
0.125
>>> print(f.partialsdev(y))         # uncertainty in f due to y
2.0
>>> print(f.partialsdev(x, y))      # uncertainty in f due to x and y
2.00390244274
>>> print(f.sdev)                   # should be the same
2.00390244274
```

This shows, for example, that most (2.0) of the uncertainty in f (2.0039) is from y.

gvar provides a useful tool for compiling an "error budget" for derived gvar.GVars relative to the primary gvar.GVars from which they were constructed: continuing the example above,

```
>>> outputs = {'f':f, 'f/y':f/y}
>>> inputs = {'x':x, 'y':y}
>>> print(gvar.fmt_values(outputs))
Values:
               f/y: 1.010(13)
                 f: 10.1(2.0)

>>> print(gvar.fmt_errorbudget(outputs=outputs, inputs=inputs))
Partial % Errors:
                          f/y         f
-----------------------------------------
                  y:      0.20     19.80
                  x:      1.24      1.24
-----------------------------------------
              total:      1.25     19.84
```

This shows y is responsible for 19.80% of the 19.84% uncertainty in f, but only 0.2% of the 1.25% uncertainty in f/y. The total uncertainty in each case is obtained by adding the x and y contributions in quadrature.

2.5 Storing gvar.GVars for Later Use; gvar.BufferDicts

Storing gvar.GVars in a file for later use is complicated by the need to capture the covariances between different gvar.GVars as well as their means. To pickle an array or dictionary g of gvar.GVars, for example, we might use

```
>>> gtuple = (gvar.mean(g), gvar.evalcov(g))
>>> import pickle
>>> pickle.dump(gtuple, open('outputfile.p', 'wb'))
```

to extract the means and covariance matrix into a tuple which then is saved in file `'output.p'` using Python's standard `pickle` module. To reassemble the `gvar.GVar`s we use:

```
>>> g = gvar.gvar(pickle.load('outputfile.p', 'rb'))
```

where `pickle.load()` reads `gtuple` back in, and `gvar.gvar()` converts it back into a collection of `gvar.GVar`s. The correlations between different `gvar.GVar`s in the original array/dictionary `g` are preserved here, but their correlations with other `gvar.GVar`s are lost. So it is important to include all `gvar.GVar`s of interest in a single array or dictionary before saving them.

This recipe works for `g`s that are: single `gvar.GVar`s, arrays of `gvar.GVar`s (any shape), or dictionaries whose values are `gvar.GVar`s and/or arrays of `gvar.GVar`s. For convenience, it is implemented in functions `gvar.dump()`, `gvar.dumps()`, `gvar.load()`, and `gvar.loads()`.

Pickling is simplified if the `gvar.GVar`s that need saving are all in a `gvar.BufferDict` since these can be serialized and saved to a file again using Python's `pickle` module. So if `g` is a `gvar.BufferDict` containing `gvar.GVar`s (and/or arrays of `gvar.GVar`s),

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

saves the contents of `g` to a file named `outputfile.p`. The `gvar.GVar`s are retrieved using:

```
>>> g = pickle.load(open('outputfile.p', 'rb'))
```

`gvar.BufferDict`s also have methods that allow saving their contents using Python's `json` module rather than `pickle`.

2.6 Random Number Generators

`gvar.GVar`s represent probability distributions. It is possible to use them to generate random numbers from those distributions. For example, in

```
>>> z = gvar.gvar(2.0, 0.5)
>>> print(z())
2.29895701465
>>> print(z())
3.00633184275
>>> print(z())
1.92649199321
```

calls to `z()` generate random numbers from a Gaussian random number generator with mean `z.mean=2.0` and standard deviation `z.sdev=0.5`.

To obtain random arrays from an array `g` of `gvar.GVar`s use `giter=gvar.raniter(g)` (see `gvar.raniter()`) to create a random array generator `giter`. Each call to `next(giter)` generates a new array of random numbers. The random number arrays have the same shape as the array `g` of `gvar.GVar`s and have the distribution implied by those random variables (including correlations). For example,

```
>>> a = gvar.gvar(1.0, 1.0)
>>> da = gvar.gvar(0.0, 0.1)
>>> g = [a, a+da]
>>> giter = gvar.raniter(g)
```

```
>>> print(next(giter))
[ 1.51874589  1.59987422]
>>> print(next(giter))
[-1.39755111 -1.24780937]
>>> print(next(giter))
[ 0.49840244  0.50643312]
```

Note how the two random numbers separately vary over the region 1 ± 1 (approximately), but the separation between the two is rarely more than 0 ± 0.1. This is as expected given the strong correlation between `a` and `a+da`.

`gvar.raniter(g)` also works when `g` is a dictionary (or `gvar.BufferDict`) whose entries `g[k]` are `gvar.GVar`s or arrays of `gvar.GVar`s. In such cases the iterator returns a dictionary with the same layout:

```
>>> g = dict(a=gvar.gvar(0, 1), b=[gvar.gvar(0, 100), gvar.gvar(10, 1e-3)])
>>> print(g)
{'a': 0.0(1.0), 'b': [0(100), 10.0000(10)]}
>>> giter = gvar.raniter(g)
>>> print(next(giter))
{'a': -0.88986130981173306, 'b': array([-67.02994213,   9.99973707])}
>>> print(next(giter))
{'a': 0.21289976681277872, 'b': array([ 29.9351328 ,  10.00008606])}
```

One use for such random number generators is dealing with situations where the standard deviations are too large to justify the linearization assumed in defining functions of Gaussian variables. Consider, for example,

```
>>> x = gvar.gvar(1., 3.)
>>> print(cos(x))
0.5(2.5)
```

The standard deviation for `cos(x)` is obviously wrong since `cos(x)` can never be larger than one. To obtain the real mean and standard deviation, we generate a large number of random numbers `xi` from `x`, compute `cos(xi)` for each, and compute the mean and standard deviation for the resulting distribution (or any other statistical quantity, particularly if the resulting distribution is not Gaussian):

```
# estimate mean,sdev from 1000 random x's
>>> ran_x = numpy.array([x() for in range(1000)])
>>> ran_cos = numpy.cos(ran_x)
>>> print('mean =', ran_cos.mean(), '  std dev =', ran_cos.std())
mean = 0.0350548954142   std dev = 0.718647118869

# check by doing more (and different) random numbers
>>> ran_x = numpy.array([x() for in range(100000)])
>>> ran_cos = numpy.cos(ran_x)
>>> print('mean =', ran_cos.mean(), '  std dev =', ran_cos.std())
mean = 0.00806276057656   std dev = 0.706357174056
```

This procedure generalizes trivially for multidimensional analyses, using arrays or dictionaries with `gvar.raniter()`.

Finally note that *bootstrap* copies of `gvar.GVar`s are easily created. A bootstrap copy of `gvar.GVar` `x ± dx` is another `gvar.GVar` with the same width but where the mean value is replaced by a random number drawn from the original distribution. Bootstrap copies of a data set, described by a collection of `gvar.GVar`s, can be used as new (fake) data sets having the same statistical errors and correlations:

```
>>> g = gvar.gvar([1.1, 0.8], [[0.01, 0.005], [0.005, 0.01]])
>>> print(g)
[1.10(10) 0.80(10)]
>>> print(gvar.evalcov(g))                  # print covariance matrix
[[ 0.01   0.005]
```

```
 [ 0.005  0.01 ]]
>>> gbs_iter = gvar.bootstrap_iter(g)
>>> gbs = next(gbs_iter)                    # bootstrap copy of f
>>> print(gbs)
[1.14(10) 0.90(10)]                         # different means
>>> print(gvar.evalcov(gbs))
[[ 0.01   0.005]                            # same covariance matrix
 [ 0.005  0.01 ]]
```

Such fake data sets are useful for analyzing non-Gaussian behavior, for example, in nonlinear fits.

2.7 Limitations

The most fundamental limitation of this module is that the calculus of Gaussian variables that it assumes is only valid when standard deviations are small (compared to the distances over which the functions of interest change appreciably). One way of dealing with this limitation is described above in the section on *Random Number Generators*.

Another potential issue is roundoff error, which can become problematic if there is a wide range of standard deviations among correlated modes. For example, the following code works as expected:

```
>>> from gvar import gvar, evalcov
>>> tiny = 1e-4
>>> a = gvar(0., 1.)
>>> da = gvar(tiny, tiny)
>>> a, ada = gvar([a.mean, (a+da).mean], evalcov([a, a+da])) # = a,a+da
>>> print(ada-a)    # should be da again
0.00010(10)
```

Reducing `tiny`, however, leads to problems:

```
>>> from gvar import gvar, evalcov
>>> tiny = 1e-8
>>> a = gvar(0., 1.)
>>> da = gvar(tiny, tiny)
>>> a, ada = gvar([a.mean, (a+da).mean], evalcov([a, a+da])) # = a, a+da
>>> print(ada-a)    # should be da again
1(0)e-08
```

Here the call to `gvar.evalcov()` creates a new covariance matrix for `a` and `ada = a+da`, but the matrix does not have enough numerical precision to encode the size of `da`'s variance, which gets set, in effect, to zero. The problem arises here for values of `tiny` less than about 2e-8 (with 64-bit floating point numbers — `tiny**2` is what appears in the covariance matrix).

2.8 Optimizations

When there are lots of primary `gvar.GVar`s, the number of derivatives stored for each derived `gvar.GVar` can become rather large, potentially (though not necessarily) leading to slower calculations. One way to alleviate this problem, should it arise, is to separate the primary variables into groups that are never mixed in calculations and to use different `gvar.gvar()`s when generating the variables in different groups. New versions of `gvar.gvar()` are obtained using `gvar.switch_gvar()`: for example,

```
import gvar
...
x = gvar.gvar(...)
```

```
y = gvar.gvar(...)
z = f(x, y)
... other manipulations involving x and y ...
gvar.switch_gvar()
a = gvar(...)
b = gvar(...)
c = g(a, b)
... other manipulations involving a and b (but not x and y) ...
```

Here the `gvar.gvar()` used to create `a` and `b` is a different function than the one used to create `x` and `y`. A derived quantity, like `c`, knows about its derivatives with respect to `a` and `b`, and about their covariance matrix; but it carries no derivative information about `x` and `y`. Absent the `switch_gvar` line, `c` would have information about its derivatives with respect to `x` and `y` (zero derivative in both cases) and this would make calculations involving `c` slightly slower than with the `switch_gvar` line. Usually the difference is negligible — it used to be more important, in earlier implementations of `gvar.GVar` before sparse matrices were introduced to keep track of covariances. Note that the previous `gvar.gvar()` can be restored using `gvar.restore_gvar()`.

2.9 Utilities

The function used to create Gaussian variable objects is:

`gvar.`**`gvar`**`(...)`
Create one or more new `gvar.GVar`s.

Each of the following creates new `gvar.GVar`s:

`gvar.`**`gvar`**`(`*x*, *xsdev*`)`
Returns a `gvar.GVar` with mean `x` and standard deviation `xsdev`. Returns an array of `gvar.GVar`s if `x` and `xsdev` are arrays with the same shape; the shape of the result is the same as the shape of `x`. Returns a `gvar.BufferDict` if `x` and `xsdev` are dictionaries with the same keys and layout; the result has the same keys and layout as `x`.

`gvar.`**`gvar`**`(`*x*, *xcov*`)`
Returns an array of `gvar.GVar`s with means given by array `x` and a covariance matrix given by array `xcov`, where `xcov.shape = 2*x.shape`; the result has the same shape as `x`. Returns a `gvar.BufferDict` if `x` and `xcov` are dictionaries, where the keys in `xcov` are `(k1,k2)` for any keys `k1` and `k2` in `x`. Returns a single `gvar.GVar` if `x` is a number and `xcov` is a one-by-one matrix. The layout for `xcov` is compatible with that produced by `gvar.evalcov()` for a single `gvar.GVar`, an array of `gvar.GVar`s, or a dictionary whose values are `gvar.GVar`s and/or arrays of `gvar.GVar`s. Therefore `gvar.gvar(gvar.mean(g), gvar.evalcov(g))` creates `gvar.GVar`s with the same means and covariance matrix as the `gvar.GVar`s in `g` provided `g` is a single `gvar.GVar`, or an array or dictionary of `gvar.GVar`s.

`gvar.`**`gvar`**`((`*x*, *xsdev*`))`
Returns a `gvar.GVar` with mean `x` and standard deviation `xsdev`.

`gvar.`**`gvar`**`(`*xstr*`)`
Returns a `gvar.GVar` corresponding to string `xstr` which is either of the form `"xmean +- xsdev"` or `"x(xerr)"` (see `GVar.fmt()`).

`gvar.`**`gvar`**`(`*xgvar*`)`
Returns `gvar.GVar` `xgvar` unchanged.

`gvar.`**`gvar`**`(`*xdict*`)`
Returns a dictionary (`BufferDict`) `b` where `b[k] = gvar(xdict[k])` for every key in dictionary `xdict`. The values in `xdict`, therefore, can be strings, tuples or `gvar.GVar`s (see above), or arrays of these.

gvar.**gvar**(*xarray*)
Returns an array `a` having the same shape as `xarray` where every element `a[i...] = gvar(xarray[i...])`. The values in `xarray`, therefore, can be strings, tuples or `gvar.GVar`s (see above).

`gvar.gvar` is actually an object of type `gvar.GVarFactory`.

The following function is useful for constructing new functions that can accept `gvar.GVar`s as arguments:

gvar.**gvar_function**(*x*, *f*, *dfdx*)
Create a `gvar.GVar` for function f(x) given f and df/dx at x.

This function creates a `gvar.GVar` corresponding to a function of `gvar.GVar`s `x` whose value is `f` and whose derivatives with respect to each `x` are given by `dfdx`. Here `x` can be a single `gvar.GVar`, an array of `gvar.GVar`s (for a multidimensional function), or a dictionary whose values are `gvar.GVar`s or arrays of `gvar.GVar`s, while `dfdx` must be a float, an array of floats, or a dictionary whose values are floats or arrays of floats, respectively.

This function is useful for creating functions that can accept `gvar.GVar`s as arguments. For example,

```
import math
import gvar as gv

def sin(x):
    if isinstance(x, gv.GVar):
        f = math.sin(x.mean)
        dfdx = math.cos(x.mean)
        return gv.gvar_function(x, f, dfdx)
    else:
        return math.sin(x)
```

creates a version of `sin(x)` that works with either floats or `gvar.GVar`s as its argument. This particular function is unnecessary since it is already provided by `gvar`.

Parameters

- **x** (`gvar.GVar`, array of `gvar.GVar`s, or a dictionary of `gvar.GVar`s) – Point at which the function is evaluated.
- **f** (*float*) – Value of function at point `gvar.mean(x)`.
- **dfdx** (*float, array of floats, or a dictionary of floats*) – Derivatives of function with respect to x at point `gvar.mean(x)`.

Returns A `gvar.GVar` representing the function's value at `x`.

Means, standard deviations, variances, formatted strings, covariance matrices and correlation/comparison information can be extracted from arrays (or dictionaries) of `gvar.GVar`s using:

gvar.**mean**(*g*)
Extract means from `gvar.GVar`s in `g`.

`g` can be a `gvar.GVar`, an array of `gvar.GVar`s, or a dictionary containing `gvar.GVar`s or arrays of `gvar.GVar`s. Result has the same layout as `g`.

`g` is returned unchanged if it contains something other than `gvar.GVar`s.

gvar.**sdev**(*g*)
Extract standard deviations from `gvar.GVar`s in `g`.

`g` can be a `gvar.GVar`, an array of `gvar.GVar`s, or a dictionary containing `gvar.GVar`s or arrays of `gvar.GVar`s. Result has the same layout as `g`.

gvar.**var**(*g*)
Extract variances from gvar.GVars in g.

g can be a gvar.GVar, an array of gvar.GVars, or a dictionary containing gvar.GVars or arrays of gvar.GVars. Result has the same layout as g.

gvar.**fmt**(*g*, *ndecimal=None*, *sep=''*)
Format gvar.GVars in g.

g can be a gvar.GVar, an array of gvar.GVars, or a dictionary containing gvar.GVars or arrays of gvar.GVars. Each gvar.GVar gi in g is replaced by the string generated by gi.fmt(ndecimal,sep). Result has same structure as g.

gvar.**evalcov**(*g*)
Compute covariance matrix for elements of array/dictionary g.

If g is an array of gvar.GVars, evalcov returns the covariance matrix as an array with shape g.shape+g.shape. If g is a dictionary whose values are gvar.GVars or arrays of gvar.GVars, the result is a doubly-indexed dictionary where cov[k1,k2] is the covariance for g[k1].flat and g[k2].flat.

gvar.**evalcorr**(*g*)
Compute correlation matrix for elements of array/dictionary g.

If g is an array of gvar.GVars, evalcorr returns the correlation matrix as an array with shape g.shape+g.shape. If g is a dictionary whose values are gvar.GVars or arrays of gvar.GVars, the result is a doubly-indexed dictionary where corr[k1,k2] is the correlation for g[k1] and g[k2].

The correlation matrix is related to the covariance matrix by:

```
corr[i,j] = cov[i,j] / (cov[i,i] * cov[j,j]) ** 0.5
```

gvar.**uncorrelated**(*g1*, *g2*)
Return True if gvar.GVars in g1 uncorrelated with those in g2.

g1 and g2 can be gvar.GVars, arrays of gvar.GVars, or dictionaries containing gvar.GVars or arrays of gvar.GVars. Returns True if either of g1 or g2 is None.

gvar.**chi2**(*g1*, *g2*, *svdcut=1e-15*)
Compute chi**2 of g1-g2.

chi**2 is a measure of whether the multi-dimensional Gaussian distributions g1 and g2 (dictionaries or arrays) agree with each other — that is, do their means agree within errors for corresponding elements. The probability is high if chi2(g1,g2)/chi2.dof is of order 1 or smaller.

Usually g1 and g2 are dictionaries with the same keys, where g1[k] and g2[k] are gvar.GVars or arrays of gvar.GVars having the same shape. Alternatively g1 and g2 can be gvar.GVars, or arrays of gvar.GVars having the same shape.

One of g1 or g2 can contain numbers instead of gvar.GVars, in which case chi**2 is a measure of the likelihood that the numbers came from the distribution specified by the other argument.

One or the other of g1 or g2 can be missing keys, or missing elements from arrays. Only the parts of g1 and g2 that overlap are used. Also setting g2=None is equivalent to replacing its elements by zeros.

chi**2 is computed from the inverse of the covariance matrix of g1-g2. The matrix inversion can be sensitive to roundoff errors. In such cases, SVD cuts can be applied by setting parameters svdcut; see the documentation for gvar.svd(), which is used to apply the cut.

The return value is the chi**2. Extra data is stored in chi2 itself:

chi2.**dof**
Number of degrees of freedom (that is, the number of variables compared).

chi2.**Q**
The probability that the chi**2 could have been larger, by chance, even if g1 and g2 agree. Values smaller than 0.1 or so suggest that they do not agree. Also called the *p-value*.

If argument fmt==True, then a string is returned containing the chi**2 per degree of freedom, the number of degrees of freedom, and Q.

gvar.**fmt_chi2**(*f*)
Return string containing chi**2/dof, dof and Q from f.

Assumes f has attributes chi2, dof and Q. The logarithm of the Bayes factor will also be printed if f has attribute logGBF.

gvar.GVars can be stored (pickled) and retrieved from files (or strings) using:

gvar.**dump**(*g*, *outputfile*)
pickle a collection g of gvar.GVars in file outputfile.

The gvar.GVars are recovered using gvar.load().

Parameters

- **g** – A gvar.GVar, array of gvar.GVars, or dictionary whose values are gvar.GVars and/or arrays of gvar.GVars.
- **outputfile** (*string or file-like object*) – The name of a file or a file object in which the pickled gvar.GVars are stored.

gvar.**dumps**(*g*)
pickle a collection g of gvar.GVars and return as a string.

The gvar.GVars are recovered using gvar.loads().

Parameters **g** – A gvar.GVar, array of gvar.GVars, or dictionary whose values are gvar.GVars and/or arrays of gvar.GVars.

gvar.**load**(*inputfile*)
Load and return pickled gvar.GVars from file inputfile.

This function recovers gvar.GVars pickled with gvar.dump().

Parameters **outputfile** (*string or file-like object*) – The name of the file or a file object in which the pickled gvar.GVars are stored.

Returns The reconstructed gvar.GVar, or array or dictionary of gvar.GVars.

gvar.**loads**(*inputstring*)
Load and return pickled gvar.GVars from string inputstring.

This function recovers gvar.GVars pickled with gvar.dumps().

Parameters **inputstring** – A string containing gvar.GVars pickled using gvar.dumps().

Returns The reconstructed gvar.GVar, or array or dictionary of gvar.GVars.

gvar.GVars contain information about derivatives with respect to the *independent* gvar.GVars from which they were constructed. This information can be extracted using:

gvar.**deriv**(*g*, *x*)
Compute first derivatives wrt x of gvar.GVars in g.

g can be a gvar.GVar, an array of gvar.GVars, or a dictionary containing gvar.GVars or arrays of gvar.GVars. Result has the same layout as g.

`x` must be an *primary* `gvar.GVar`, which is a `gvar.GVar` created by a call to `gvar.gvar()` (*e.g.*, `x = gvar.gvar(xmean, xsdev)`) or a function `f(x)` of such a `gvar.GVar`. (More precisely, `x.der` must have only one nonzero entry.)

The following function creates an iterator that generates random arrays from the distribution defined by array (or dictionary) `g` of `gvar.GVar`s. The random numbers incorporate any correlations implied by the `g`s.

`gvar.`**`raniter`**(*g*, *n=None*, *svdcut=None*)

Return iterator for random samples from distribution `g`

The gaussian variables (`gvar.GVar` objects) in array (or dictionary) `g` collectively define a multidimensional gaussian distribution. The iterator defined by `raniter()` generates an array (or dictionary) containing random numbers drawn from that distribution, with correlations intact.

The layout for the result is the same as for `g`. So an array of the same shape is returned if `g` is an array. When `g` is a dictionary, individual entries `g[k]` may be `gvar.GVar`s or arrays of `gvar.GVar`s, with arbitrary shapes.

`raniter()` also works when `g` is a single `gvar.GVar`, in which case the resulting iterator returns random numbers drawn from the distribution specified by `g`.

Parameters

- **g** (*array or dictionary or BufferDict or GVar*) – An array (or dictionary) of objects of type `gvar.GVar`; or a `gvar.GVar`.
- **n** – Maximum number of random iterations. Setting `n=None` (the default) implies there is no maximum number.
- **svdcut** (`None` or number) – If positive, replace eigenvalues `eig` of `g`'s correlation matrix with `max(eig, svdcut * max_eig)` where `max_eig` is the largest eigenvalue; if negative, discard eigenmodes with eigenvalues smaller than `|svdcut| * max_eig`. Default is `None`.

Returns An iterator that returns random arrays or dictionaries with the same shape as `g` drawn from the gaussian distribution defined by `g`.

`gvar.`**`bootstrap_iter`**(*g*, *n=None*, *svdcut=None*)

Return iterator for bootstrap copies of `g`.

The gaussian variables (`gvar.GVar` objects) in array (or dictionary) `g` collectively define a multidimensional gaussian distribution. The iterator created by `bootstrap_iter()` generates an array (or dictionary) of new `gvar.GVar`s whose covariance matrix is the same as `g`'s but whose means are drawn at random from the original `g` distribution. This is a *bootstrap copy* of the original distribution. Each iteration of the iterator has different means (but the same covariance matrix).

`bootstrap_iter()` also works when `g` is a single `gvar.GVar`, in which case the resulting iterator returns bootstrap copies of the `g`.

Parameters

- **g** (*array or dictionary or BufferDict*) – An array (or dictionary) of objects of type `gvar.GVar`.
- **n** – Maximum number of random iterations. Setting `n=None` (the default) implies there is no maximum number.
- **svdcut** (`None` or number) – If positive, replace eigenvalues `eig` of `g`'s correlation matrix with `max(eig, svdcut * max_eig)` where `max_eig` is the largest eigenvalue; if negative, discard eigenmodes with eigenvalues smaller than `|svdcut| * max_eig`. Default is `None`.

Returns An iterator that returns bootstrap copies of `g`.

gvar.**ranseed**(*a*)
Seed random number generators with tuple seed.

Argument seed is a tuple of integers that is used to seed the random number generators used by numpy and random (and therefore by gvar). Reusing the same seed results in the same set of random numbers.

ranseed generates its own seed when called without an argument or with seed=None. This seed is stored in ranseed.seed and also returned by the function. The seed can be used to regenerate the same set of random numbers at a later time.

Parameters **seed** (*tuple or None*) – A tuple of integers. Generates a random tuple if None.

Returns The seed.

The following two functions that are useful for tabulating results and for analyzing where the errors in a gvar.GVar constructed from other gvar.GVars come from:

gvar.**fmt_errorbudget**(*outputs*, *inputs*, *ndecimal=2*, *percent=True*, *colwidth=10*)
Tabulate error budget for outputs[ko] due to inputs[ki].

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

Parameters

- **outputs** – Dictionary of gvar.GVars for which an error budget is computed.
- **inputs** – Dictionary of: gvar.GVars, arrays/dictionaries of gvar.GVars, or lists of gvar.GVars and/or arrays/dictionaries of gvar.GVars. fmt_errorbudget tabulates the parts of the standard deviations of each outputs[ko] due to each inputs[ki].
- **ndecimal** (int) – Number of decimal places displayed in table.
- **percent** (*boolean*) – Tabulate % errors if percent is True; otherwise tabulate the errors themselves.
- **colwidth** (*positive integer*) – Width of each column.

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

gvar.**fmt_values**(*outputs*, *ndecimal=None*)
Tabulate gvar.GVars in outputs.

Parameters

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

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

The following function applies an SVD cut to the correlation matrix of a set of gvar.GVars:

gvar.**svd**(*g*, *svdcut=1e-15*, *wgts=False*)
Apply svd cuts to collection of gvar.GVars in g.

Standard usage is, for example,

```
svdcut = ...
gmod = svd(g, svdcut=svdcut)
```

where g is an array of gvar.GVars or a dictionary containing gvar.GVars and/or arrays of gvar.GVars. When svdcut>0, gmod is a copy of g whose gvar.GVars have been modified to make their correlation matrix less singular than that of the original g: each eigenvalue eig of the correlation matrix is replaced by max(eig, svdcut * max_eig) where max_eig is the largest eigenvalue. This SVD cut, which is applied separately to each block-diagonal sub-matrix of the correlation matrix, increases the variance of the eigenmodes with eigenvalues smaller than svdcut * max_eig.

When svdcut is negative, eigenmodes of the correlation matrix whose eigenvalues are smaller than |svdcut| * max_eig are dropped from the new matrix and the corresponding components of g are zeroed out (that is, replaced by 0(0)) in gmod.

There is an additional parameter wgts in gvar.svd() whose default value is False. Setting wgts=1 or wgts=-1 instead causes gvar.svd() to return a tuple (gmod, i_wgts) where gmod is the modified copy of g, and i_wgts contains a spectral decomposition of the covariance matrix corresponding to the modified correlation matrix if wgts=1, or a decomposition of its inverse if wgts=-1. The first entry i, wgts = i_wgts[0] specifies the diagonal part of the matrix: i is a list of the indices in gmod.flat corresponding to diagonal elements, and wgts ** 2 gives the corresponding matrix elements. The second and subsequent entries, i, wgts = i_wgts[n] for n > 0, each correspond to block-diagonal sub-matrices, where i is the list of indices corresponding to the block, and wgts[j] are eigenvectors of the sub-matrix rescaled so that

```
numpy.sum(numpy.outer(wi, wi) for wi in wgts[j]
```

is the sub-matrix (wgts=1) or its inverse (wgts=-1).

To compute the inverse of the covariance matrix from i_wgts, for example, one could use code like:

```
gmod, i_wgts = svd(g, svdcut=svdcut, wgts=-1)

inv_cov = numpy.zeros((n, n), float)
i, wgts = i_wgts[0]                         # 1x1 sub-matrices
if len(i) > 0:
    inv_cov[i, i] = numpy.array(wgts) ** 2
for i, wgts in i_wgts[1:]:                  # nxn sub-matrices (n>1)
    for w in wgts:
        inv_cov[i, i[:, None]] += numpy.outer(w, w)
```

This sets inv_cov equal to the inverse of the covariance matrix of the gmods. Similarly, we can compute the expectation value, u.dot(inv_cov.dot(v)), between two vectors (numpy arrays) using:

```
result = 0.0
i, wgts = i_wgts[0]                         # 1x1 sub-matrices
if len(i) > 0:
    result += numpy.sum((u[i] * wgts) * (v[i] * wgts))
for i, wgts in i_wgts[1:]:                  # nxn sub-matrices (n>1)
    result += numpy.sum(wgts.dot(u[i]) * wgts.dot(v[i]))
```

where result is the desired expectation value.

The input parameters are :

Parameters

- **g** – An array of gvar.GVars or a dicitionary whose values are gvar.GVars and/or arrays of gvar.GVars.
- **svdcut** (None or number (|svdcut|<=1).) – If positive, replace eigenvalues eig of the correlation matrix with max(eig, svdcut * max_eig) where max_eig is

the largest eigenvalue; if negative, discard eigenmodes with eigenvalues smaller than `|svdcut| * max_eig`. Default is 1e-15.

- **wgts** – Setting `wgts=1` causes `gvar.svd()` to compute and return a spectral decomposition of the covariance matrix of the modified `gvar.GVar`s, `gmod`. Setting `wgts=-1` results in a decomposition of the inverse of the covariance matrix. The default value is `False`, in which case only `gmod` is returned.

Returns A copy `gmod` of `g` whose correlation matrix is modified by *svd* cuts. If `wgts` is not `False`, a tuple `(g, i_wgts)` is returned where `i_wgts` contains a spectral decomposition of `gmod`'s covariance matrix or its inverse.

Data from the *svd* analysis of `g`'s covariance matrix is stored in `svd` itself:

`svd.`**`dof`**
Number of independent degrees of freedom left after the *svd* cut. This is the same as the number initially unless `svdcut < 0` in which case it may be smaller.

`svd.`**`nmod`**
Number of modes whose eignevalue was modified by the *svd* cut.

`svd.`**`nblocks`**
A dictionary where `svd.nblocks[s]` contains the number of block-diagonal `s`-by-`s` sub-matrices in the correlation matrix.

`svd.`**`eigen_range`**
Ratio of the smallest to largest eigenvalue before *svd* cuts are applied (but after rescaling).

`svd.`**`logdet`**
Logarithm of the determinant of the covariance matrix after *svd* cuts are applied (excluding any omitted modes when `svdcut < 0`).

`svd.`**`correction`**
Array containing the *svd* corrections that were added to `g.flat` to create the modified `g`s.

This function is useful when the correlation matrix is singular or almost singular, and its inverse is needed (as in curve fitting).

The following function can be used to rebuild collections of `gvar.GVar`s, ignoring all correlations with other variables. It can also be used to introduce correlations between uncorrelated variables.

`gvar.`**`rebuild`**(*g*, *gvar=gvar*, *corr=0.0*)
Rebuild `g` stripping correlations with variables not in `g`.

`g` is either an array of `gvar.GVar`s or a dictionary containing `gvar.GVar`s and/or arrays of `gvar.GVar`s. `rebuild(g)` creates a new collection `gvar.GVar`s with the same layout, means and covariance matrix as those in `g`, but discarding all correlations with variables not in `g`.

If `corr` is nonzero, `rebuild` will introduce correlations wherever there aren't any using

```
cov[i,j] -> corr * sqrt(cov[i,i]*cov[j,j])
```

wherever `cov[i,j]==0.0` initially. Positive values for `corr` introduce positive correlations, negative values anti-correlations.

Parameter `gvar` specifies a function for creating new `gvar.GVar`s that replaces `gvar.gvar()` (the default).

Parameters

- **g** (*array or dictionary*) – `gvar.GVar`s to be rebuilt.
- **gvar** (`gvar.GVarFactory` or `None`) – Replacement for `gvar.gvar()` to use in rebuilding. Default is `gvar.gvar()`.

- **corr** (*number*) – Size of correlations to introduce where none exist initially.

Returns Array or dictionary (gvar.BufferDict) of `gvar.GVar`s (same layout as `g`) where all correlations with variables other than those in `g` are erased.

The following functions creates new functions that generate `gvar.GVar`s (to replace `gvar.gvar()`):

`gvar.`**`switch_gvar`**`()`
Switch `gvar.gvar()` to new `gvar.GVarFactory`.

Returns New `gvar.gvar()`.

`gvar.`**`restore_gvar`**`()`
Restore previous `gvar.gvar()`.

Returns Previous `gvar.gvar()`.

`gvar.`**`gvar_factory`**`(`*cov=None*`)`
Return new function for creating `gvar.GVar`s (to replace `gvar.gvar()`).

If `cov` is specified, it is used as the covariance matrix for new `gvar.GVar`s created by the function returned by `gvar_factory(cov)`. Otherwise a new covariance matrix is created internally.

`gvar.GVar`s created by different functions cannot be combined in arithmetic expressions (the error message "Incompatible GVars." results).

2.10 Classes

The fundamental class for representing Gaussian variables is:

class `gvar.`**`GVar`**
The basic attributes are:

`mean`
Mean value.

`sdev`
Standard deviation.

`var`
Variance.

Two methods allow one to isolate the contributions to the variance or standard deviation coming from other `gvar.GVar`s:

`partialvar``(`**args*`)`
Compute partial variance due to `gvar.GVar`s in `args`.

This method computes the part of `self.var` due to the `gvar.GVar`s in `args`. If `args[i]` is correlated with other `gvar.GVar`s, the variance coming from these is included in the result as well. (This last convention is necessary because variances associated with correlated `gvar.GVar`s cannot be disentangled into contributions corresponding to each variable separately.)

Parameters **args[i]** (`gvar.GVar` or array/dictionary of `gvar.GVar`s) – Variables contributing to the partial variance.

Returns Partial variance due to all of `args`.

`partialsdev``(`**args*`)`
Compute partial standard deviation due to `gvar.GVar`s in `args`.

This method computes the part of `self.sdev` due to the `gvar.GVar`s in `args`. If `args[i]` is correlated with other `gvar.GVar`s, the standard deviation coming from these is included in the result as well. (This last convention is necessary because variances associated with correlated `gvar.GVar`s cannot be disentangled into contributions corresponding to each variable separately.)

Parameters **args[i]** (`gvar.GVar` or array/dictionary of `gvar.GVar`s) – Variables contributing to the partial standard deviation.

Returns Partial standard deviation due to `args`.

Partial derivatives of the `gvar.GVar` with respect to the independent `gvar.GVar`s from which it was constructed are given by:

deriv(*x*)

Derivative of `self` with respest to *primary* `gvar.GVar` `x`.

All `gvar.GVar`s are constructed from primary `gvar.GVar`s. `self.deriv(x)` returns the partial derivative of `self` with respect to primary `gvar.GVar` `x`, holding all of the other primary `gvar.GVar`s constant.

Parameters **x** – A primary `gvar.GVar` (or a function of a single primary `gvar.GVar`).

Returns The derivative of `self` with respect to `x`.

There are two methods for converting `self` into a string, for printing:

__str__()

Return string representation of `self`.

The representation is designed to show at least one digit of the mean and two digits of the standard deviation. For cases where mean and standard deviation are not too different in magnitude, the representation is of the form `'mean(sdev)'`. When this is not possible, the string has the form `'mean +- sdev'`.

fmt(*ndecimal=None*, *sep=''*)

Convert to string with format: `mean(sdev)`.

Leading zeros in the standard deviation are omitted: for example, `25.67 +- 0.02` becomes `25.67(2)`. Parameter `ndecimal` specifies how many digits follow the decimal point in the mean. Parameter `sep` is a string that is inserted between the `mean` and the `(sdev)`. If `ndecimal` is `None` (default), it is set automatically to the larger of `int(2-log10(self.sdev))` or `0`; this will display at least two digits of error. Very large or very small numbers are written with exponential notation when `ndecimal` is `None`.

Setting `ndecimal < 0` returns `mean +- sdev`.

Two attributes and a method make reference to the original variables from which `self` is derived:

cov

Underlying covariance matrix (type `gvar.smat`) shared by all `gvar.GVar`s.

der

Array of derivatives with respect to underlying (original) `gvar.GVar`s.

dotder(*v*)

Return the dot product of `self.der` and `v`.

The following class is a specialized form of an ordered dictionary for holding `gvar.GVar`s (or other scalars) and arrays of `gvar.GVar`s (or other scalars) that supports Python pickling:

class `gvar.`**BufferDict**

Dictionary whose data is packed into a 1-d buffer (numpy.array).

A gvar.BufferDict object is a dictionary-like object whose values must either be scalars or arrays (like numpy arrays, with arbitrary shapes). The scalars and arrays are assembled into different parts of a single one-dimensional buffer. The various scalars and arrays are retrieved using keys, as in a dictionary: *e.g.*,

```
>>> a = BufferDict()
>>> a['scalar'] = 0.0
>>> a['vector'] = [1.,2.]
>>> a['tensor'] = [[3.,4.],[5.,6.]]
>>> print(a.flatten())              # print a's buffer
[ 0.  1.  2.  3.  4.  5.  6.]
>>> for k in a:                     # iterate over keys in a
...     print(k,a[k])
scalar 0.0
vector [ 1.  2.]
tensor [[ 3.  4.]
 [ 5.  6.]]
>>> a['vector'] = a['vector']*10    # change the 'vector' part of a
>>> print(a.flatten())
[  0.  10.  20.   3.   4.   5.   6.]
```

The first four lines here could have been collapsed to one statement:

```
a = BufferDict(scalar=0.0,vector=[1.,2.],tensor=[[3.,4.],[5.,6.]])
```

or

```
a = BufferDict([('scalar',0.0),('vector',[1.,2.]),
                ('tensor',[[3.,4.],[5.,6.]])])
```

where in the second case the order of the keys is preserved in a (that is, BufferDict is an ordered dictionary).

The keys and associated shapes in a gvar.BufferDict can be transferred to a different buffer, creating a new gvar.BufferDict: *e.g.*, using a from above,

```
>>> buf = numpy.array([0.,10.,20.,30.,40.,50.,60.])
>>> b = BufferDict(a,buf=buf)       # clone a but with new buffer
>>> print(b['tensor'])
[[ 30.  40.]
 [ 50.  60.]]
>>> b['scalar'] += 1
>>> print(buf)
[  1.  10.  20.  30.  40.  50.  60.]
```

Note how b references buf and can modify it. One can also replace the buffer in the original gvar.BufferDict using, for example, a.buf = buf:

```
>>> a.buf = buf
>>> print(a['tensor'])
[[ 30.  40.]
 [ 50.  60.]]
>>> a['tensor'] *= 10.
>>> print(buf)
[  1.  10.  20.  300.  400.  500.  600.]
```

a.buf is the numpy array used for a's buffer. It can be used to access and change the buffer directly. In a.buf = buf, the new buffer buf must be a numpy array of the correct shape. The buffer can also be accessed through iterator a.flat (in analogy with numpy arrays), and through a.flatten() which returns a copy of the buffer.

When creating a gvar.BufferDict from a dictionary (or another gvar.BufferDict), the keys included and their order can be specified using a list of keys: for example,

```
>>> d = dict(a=0.0,b=[1.,2.],c=[[3.,4.],[5.,6.]],d=None)
>>> print(d)
{'a': 0.0, 'c': [[3.0, 4.0], [5.0, 6.0]], 'b': [1.0, 2.0], 'd': None}
>>> a = BufferDict(d, keys=['d', 'b', 'a'])
>>> for k in a:
...     print(k, a[k])
d None
b [1.0 2.0]
a 0.0
```

A `gvar.BufferDict` functions like a dictionary except: a) items cannot be deleted once inserted; b) all values must be either scalars or arrays of scalars, where the scalars can be any noniterable type that works with `numpy` arrays; and c) any new value assigned to an existing key must have the same size and shape as the original value.

Note that `gvar.BufferDict`s can be pickled and unpickled even when they store `gvar.GVar`s (which themselves cannot be pickled separately).

The main attributes are:

size
Size of buffer array.

flat
Buffer array iterator.

dtype
Data type of buffer array elements.

buf
The (1d) buffer array. Allows direct access to the buffer: for example, `self.buf[i] = new_val` sets the value of the `i-th` element in the buffer to value `new_val`. Setting `self.buf = nbuf` replaces the old buffer by new buffer `nbuf`. This only works if `nbuf` is a one-dimensional `numpy` array having the same length as the old buffer, since `nbuf` itself is used as the new buffer (not a copy).

shape
Always equal to `None`. This attribute is included since `gvar.BufferDict`s share several attributes with `numpy` arrays to simplify coding that might support either type. Being dictionaries they do not have shapes in the sense of `numpy` arrays (hence the shape is `None`).

The main methods are:

flatten()
Copy of buffer array.

slice(*k*)
Return slice/index in `self.flat` corresponding to key `k`.

isscalar(*k*)
Return `True` if `self[k]` is scalar else `False`.

update(*d*)
Add contents of dictionary `d` to `self`.

static load(*fobj*, *use_json=False*)
Load serialized `gvar.BufferDict` from file object `fobj`. Uses `pickle` unless `use_json` is `True`, in which case it uses `json` (obvioulsy).

static loads(*s*, *use_json=False*)
Load serialized `gvar.BufferDict` from string object `s`. Uses `pickle` unless `use_json` is `True`, in which case it uses `json` (obvioulsy).

dump (*fobj*, *use_json=False*)
Serialize `gvar.BufferDict` in file object `fobj`.

Uses `pickle` unless `use_json` is `True`, in which case it uses `json` (obviously). `json` does not handle non-string valued keys very well. This attempts a workaround, but it will only work in simpler cases. Serialization only works when `pickle` (or `json`) knows how to serialize the data type stored in the `gvar.BufferDict`'s buffer (or for `gvar.GVar`s).

dumps (*use_json=False*)
Serialize `gvar.BufferDict` into string.

Uses `pickle` unless `use_json` is `True`, in which case it uses `json` (obviously). `json` does not handle non-string valued keys very well. This attempts a workaround, but it will only work in simpler cases (e.g., integers, tuples of integers, etc.). Serialization only works when `pickle` (or `json`) knows how to serialize the data type stored in the `gvar.BufferDict`'s buffer (or for `gvar.GVar`s).

SVD analysis is handled by the following class:

class `gvar.`**SVD** (*mat*, *svdcut=None*, *svdnum=None*, *compute_delta=False*, *rescale=False*)
SVD decomposition of a pos. sym. matrix.

`SVD` is a function-class that computes the eigenvalues and eigenvectors of a positive symmetric matrix `mat`. Eigenvalues that are small (or negative, because of roundoff) can be eliminated or modified using *svd* cuts. Typical usage is:

```
>>> mat = [[1.,.25],[.25,2.]]
>>> s = SVD(mat)
>>> print(s.val)              # eigenvalues
[ 0.94098301  2.05901699]
>>> print(s.vec[0])           # 1st eigenvector (for s.val[0])
[ 0.97324899 -0.22975292]
>>> print(s.vec[1])           # 2nd eigenvector (for s.val[1])
[ 0.22975292  0.97324899]

>>> s = SVD(mat,svdcut=0.6)   # force s.val[i]>=s.val[-1]*0.6
>>> print(s.val)
[ 1.2354102   2.05901699]
>>> print(s.vec[0])           # eigenvector unchanged
[ 0.97324899 -0.22975292]

>>> s = SVD(mat)
>>> w = s.decomp(-1)          # decomposition of inverse of mat
>>> invmat = sum(numpy.outer(wj,wj) for wj in w)
>>> print(numpy.dot(mat,invmat))     # should be unit matrix
[[  1.00000000e+00   2.77555756e-17]
 [  1.66533454e-16   1.00000000e+00]]
```

Input parameters are:

Parameters

- **mat** (2-d sequence (`numpy.array` or `list` or ...)) – Positive, symmetric matrix.
- **svdcut** (`None` or number `(|svdcut|<=1)`.) – If positive, replace eigenvalues of `mat` with `svdcut*(max eigenvalue)`; if negative, discard eigenmodes with eigenvalues smaller than `svdcut` times the maximum eigenvalue.
- **svdnum** (`None` or int) – If positive, keep only the modes with the largest `svdnum` eigenvalues; ignore if set to `None`.
- **compute_delta** (*boolean*) – Compute `delta` (see below) if `True`; set `delta=None` otherwise.

- **rescale** – Rescale the input matrix to make its diagonal elements equal to 1.0 before diagonalizing.

The results are accessed using:

val
An ordered array containing the eigenvalues or `mat`. Note that `val[i]<=val[i+1]`.

vec
Eigenvectors `vec[i]` corresponding to the eigenvalues `val[i]`.

D
The diagonal matrix used to precondition the input matrix if `rescale==True`. The matrix diagonalized is `D M D` where `M` is the input matrix. `D` is stored as a one-dimensional vector of diagonal elements. `D` is `None` if `rescale==False`.

nmod
The first `nmod` eigenvalues in `self.val` were modified by the SVD cut (equals 0 unless `svdcut > 0`).

eigen_range
Ratio of the smallest to the largest eigenvector in the unconditioned matrix (after rescaling if `rescale=True`)

delta
A vector of `gvars` whose means are zero and whose covariance matrix is what was added to `mat` to condition its eigenvalues. Is `None` if `svdcut<0` or `compute_delta==False`.

decomp(*n*)
Vector decomposition of input matrix raised to power `n`.

Computes vectors `w[i]` such that

mat**n = sum_i numpy.outer(w[i],w[i])

where `mat` is the original input matrix to `svd`. This decomposition cannot be computed if the input matrix was rescaled (`rescale=True`) except for `n=1` and `n=-1`.

Parameters **n** (*number*) – Power of input matrix.

Returns Array `w` of vectors.

2.11 Requirements

`gvar` makes heavy use of `numpy` for array manipulations. It also uses the `numpy` code for implementing elementary functions (*e.g.*, `sin`, `exp` ...) in terms of member functions.

CHAPTER

THREE

GVAR.DATASET - RANDOM DATA SETS

3.1 Introduction

`gvar.dataset` contains a several tools for collecting and analyzing random samples from arbitrary distributions. The random samples are represented by lists of numbers or arrays, where each number/array is a new sample from the underlying distribution. For example, six samples from a one-dimensional gaussian distribution, 1 ± 1, might look like

```
>>> random_numbers = [1.739, 2.682, 2.493, -0.460, 0.603, 0.800]
```

while six samples from a two-dimensional distribution, $[1 \pm 1, 2 \pm 1]$, might be

```
>>> random_arrays = [[ 0.494, 2.734], [ 0.172, 1.400], [ 1.571, 1.304],
...                  [ 1.532, 1.510], [ 0.669, 0.873], [ 1.242, 2.188]]
```

Samples from more complicated multidimensional distributions are represented by dictionaries whose values are lists of numbers or arrays: for example,

```
>>> random_dict = dict(n=random_numbers, a=random_arrays)
```

where list elements `random_dict['n'][i]` and `random_dict['a'][i]` are part of the same multidimensional sample for every `i` — that is, the lists for different keys in the dictionary are synchronized one with the other.

With large samples, we typically want to estimate the mean value of the underlying distribution. This is done using `gvar.dataset.avg_data()`: for example,

```
>>> print(avg_data(random_numbers))
1.31(45)
```

indicates that `1.31(45)` is our best guess, based only upon the samples in `random_numbers`, for the mean of the distribution from which those samples were drawn. Similarly

```
>>> print(avg_data(random_arrays))
[0.95(22) 1.67(25)]
```

indicates that the means for the two-dimensional distribution behind `random_arrays` are `[0.95(22), 1.67(25)]`. `avg_data()` can also be applied to a dictionary whose values are lists of numbers/arrays: for example,

```
>>> print(avg_data(random_dict))
{'a': array([0.95(22), 1.67(25)], dtype=object),'n': 1.31(45)}
```

Class `gvar.dataset.Dataset` can be used to assemble dictionaries containing random samples. For example, imagine that the random samples above were originally written into a file, as they were generated:

```
# file: datafile
n 1.739
a [ 0.494, 2.734]
n 2.682
a [ 0.172, 1.400]
n 2.493
a [ 1.571, 1.304]
n -0.460
a [ 1.532, 1.510]
n 0.603
a [ 0.669, 0.873]
n 0.800
a [ 1.242, 2.188]
```

Here each line is a different random sample, either from the one-dimensional distribution (labeled `n`) or from the two-dimensional distribution (labeled `a`). Assuming the file is called `datafile`, this data can be read into a dictionary, essentially identical to the `data` dictionary above, using:

```
>>> data = Dataset("datafile")
>>> print(data['a'])
[array([ 0.494, 2.734]), array([ 0.172, 1.400]), array([ 1.571, 1.304]) ... ]
>>> print(avg_data(data['n']))
1.31(45)
```

The brackets and commas can be omitted in the input file for one-dimensional arrays: for example, `datafile` (above) could equivalently be written

```
# file: datafile
n 1.739
a 0.494 2.734
n 2.682
a 0.172 1.400
...
```

Other data formats may also be easy to use. For example, a data file written using `yaml` would look like

```
# file: datafile
---
n: 1.739
a: [ 0.494, 2.734]
---
n: 2.682
a: [ 0.172, 1.400]
.
.
.
```

and could be read into a `gvar.dataset.Dataset` using:

```
import yaml

data = Dataset()
with open("datafile", "r") as dfile:
    for d in yaml.load_all(dfile.read()):    # iterate over yaml records
        data.append(d)                       # d is a dictionary
```

Finally note that data can be binned, into bins of size `binsize`, using `gvar.dataset.bin_data()`. For example, `gvar.dataset.bin_data(data, binsize=3)` replaces every three samples in `data` by the average of those samples. This creates a dataset that is `1/3` the size of the original but has the same mean. Binning is use-

ful for making large datasets more manageable, and also for removing sample-to-sample correlations. Over-binning, however, erases statistical information.

Class `gvar.dataset.Dataset` can also be used to build a dataset sample by sample in code: for example,

```
>>> a = Dataset()
>>> a.append(n=1.739, a=[ 0.494, 2.734])
>>> a.append(n=2.682, a=[ 0.172, 1.400])
...
```

creates the same dataset as above.

3.2 Functions

The functions defined in the module are:

`gvar.dataset.`**`avg_data`**(*data*, *spread=False*, *median=False*, *bstrap=False*, *noerror=False*, *warn=True*)

Average random data to estimate mean.

`data` is a list of random numbers, a list of random arrays, or a dictionary of lists of random numbers and/or arrays: for example,

```
>>> random_numbers = [1.60, 0.99, 1.28, 1.30, 0.54, 2.15]
>>> random_arrays = [[12.2,121.3],[13.4,149.2],[11.7,135.3],
...                  [7.2,64.6],[15.2,69.0],[8.3,108.3]]
>>> random_dict = dict(n=random_numbers,a=random_arrays)
```

where in each case there are six random numbers/arrays. `avg_data` estimates the means of the distributions from which the random numbers/arrays are drawn, together with the uncertainties in those estimates. The results are returned as a `gvar.GVar` or an array of `gvar.GVar`s, or a dictionary of `gvar.GVar`s and/or arrays of `gvar.GVar`s:

```
>>> print(avg_data(random_numbers))
1.31(20)
>>> print(avg_data(random_arrays))
[11.3(1.1) 108(13)]
>>> print(avg_data(random_dict))
{'a': array([11.3(1.1), 108(13)], dtype=object),'n': 1.31(20)}
```

The arrays in `random_arrays` are one dimensional; in general, they can have any shape.

`avg_data(data)` also estimates any correlations between different quantities in `data`. When `data` is a dictionary, it does this by assuming that the lists of random numbers/arrays for the different `data[k]`s are synchronized, with the first element in one list corresponding to the first elements in all other lists, and so on. If some lists are shorter than others, the longer lists are truncated to the same length as the shortest list (discarding data samples).

There are four optional arguments. If argument `spread=True` each standard deviation in the results refers to the spread in the data, not the uncertainty in the estimate of the mean. The former is `sqrt(N)` larger where `N` is the number of random numbers (or arrays) being averaged:

```
>>> print(avg_data(random_numbers,spread=True))
1.31(50)
>>> print(avg_data(random_numbers))
1.31(20)
>>> print((0.50 / 0.20) ** 2)    # should be (about) 6
6.25
```

This is useful, for example, when averaging bootstrap data. The default value is `spread=False`.

The second option is triggered by setting `median=True`. This replaces the means in the results by medians, while the standard deviations are approximated by the half-width of the interval, centered around the median, that contains 68% of the data. These estimates are more robust than the mean and standard deviation when averaging over small amounts of data; in particular, they are unaffected by extreme outliers in the data. The default is `median=False`.

The third option is triggered by setting `bstrap=True`. This is shorthand for setting `median=True` and `spread=True`, and overrides any explicit setting for these keyword arguments. This is the typical choice for analyzing bootstrap data — hence its name. The default value is `bstrap=False`.

The fourth option is to omit the error estimates on the averages, which is triggered by setting `noerror=True`. Just the mean values are returned. The default value is `noerror=False`.

The final option `warn` determines whether or not a warning is issued when different components of a dictionary data set have different sample sizes.

`gvar.dataset.`**`autocorr`**(*data*)
Compute autocorrelation in random data.

`data` is a list of random numbers or random arrays, or a dictionary of lists of random numbers/arrays.

When `data` is a list of random numbers, `autocorr(data)` returns an array where `autocorr(data)[i]` is the correlation between elements in `data` that are separated by distance `i` in the list: for example,

```
>>> print(autocorr([2,-2,2,-2,2,-2]))
[ 1. -1.  1. -1.  1. -1.]
```

shows perfect correlation between elements separated by an even interval in the list, and perfect anticorrelation between elements by an odd interval.

`autocorr(data)` returns a list of arrays of autocorrelation coefficients when `data` is a list of random arrays. Again `autocorr(data)[i]` gives the autocorrelations for `data` elements separated by distance `i` in the list. Similarly `autocorr(data)` returns a dictionary when `data` is a dictionary.

`autocorr(data)` uses FFTs to compute the autocorrelations; the cost of computing the autocorrelations should grow roughly linearly with the number of random samples in `data` (up to logarithms).

`gvar.dataset.`**`bin_data`**(*data*, *binsize=2*)
Bin random data.

`data` is a list of random numbers or random arrays, or a dictionary of lists of random numbers/arrays. `bin_data(data,binsize)` replaces consecutive groups of `binsize` numbers/arrays by the average of those numbers/arrays. The result is new data list (or dictionary) with `1/binsize` times as much random data: for example,

```
>>> print(bin_data([1,2,3,4,5,6,7],binsize=2))
[1.5, 3.5, 5.5]
>>> print(bin_data(dict(s=[1,2,3,4,5],v=[[1,2],[3,4],[5,6],[7,8]]),binsize=2))
{'s': [1.5, 3.5], 'v': [array([ 2.,  3.]), array([ 6.,  7.])]}
```

Data is dropped at the end if there is insufficient data to from complete bins. Binning is used to make calculations faster and to reduce measurement-to-measurement correlations, if they exist. Over-binning erases useful information.

`gvar.dataset.`**`bootstrap_iter`**(*data*, *n=None*)
Create iterator that returns bootstrap copies of `data`.

`data` is a list of random numbers or random arrays, or a dictionary of lists of random numbers/arrays. `bootstrap_iter(data,n)` is an iterator that returns `n` bootstrap copies of data. The random num-

bers/arrays in a bootstrap copy are drawn at random (with repetition allowed) from among the samples in `data`: for example,

```
>>> data = [1.1, 2.3, 0.5, 1.9]
>>> data_iter = bootstrap_iter(data)
>>> print(next(data_iter))
[ 1.1  1.1  0.5  1.9]
>>> print(next(data_iter))
[ 0.5  2.3  1.9  0.5]

>>> data = dict(a=[1,2,3,4],b=[1,2,3,4])
>>> data_iter = bootstrap_iter(data)
>>> print(next(data_iter))
{'a': array([3, 3, 1, 2]), 'b': array([3, 3, 1, 2])}
>>> print(next(data_iter))
{'a': array([1, 3, 3, 2]), 'b': array([1, 3, 3, 2])}

>>> data = [[1,2],[3,4],[5,6],[7,8]]
>>> data_iter = bootstrap_iter(data)
>>> print(next(data_iter))
[[ 7.  8.]
 [ 1.  2.]
 [ 1.  2.]
 [ 7.  8.]]
>>> print(next(data_iter))
[[ 3.  4.]
 [ 7.  8.]
 [ 3.  4.]
 [ 1.  2.]]
```

The distribution of bootstrap copies is an approximation to the distribution from which `data` was drawn. Consequently means, variances and correlations for bootstrap copies should be similar to those in `data`. Analyzing variations from bootstrap copy to copy is often useful when dealing with non-gaussian behavior or complicated correlations between different quantities.

Parameter `n` specifies the maximum number of copies; there is no maximum if `n is None`.

3.3 Classes

`gvar.dataset.Dataset` is used to assemble random samples from multidimensional distributions:

class `gvar.dataset.`**`Dataset`**

Dictionary for collecting random data.

This dictionary class simplifies the collection of random data. The random data are stored in a dictionary, with each piece of random data being a number or an array of numbers. For example, consider a situation where there are four random values for a scalar `s` and four random values for vector `v`. These can be collected as follows:

```
>>> data = Dataset()
>>> data.append(s=1.1,v=[12.2,20.6])
>>> data.append(s=0.8,v=[14.1,19.2])
>>> data.append(s=0.95,v=[10.3,19.7])
>>> data.append(s=0.91,v=[8.2,21.0])
>>> print(data['s'])       # 4 random values of s
[ 1.1, 0.8, 0.95, 0.91]
>>> print(data['v'])       # 4 random vector-values of v
[array([ 12.2,  20.6]), array([ 14.1,  19.2]), array([ 10.3,  19.7]), array([  8.2,  21. ])]
```

The argument to `data.append()` could be a dictionary: for example, `dd = dict(s=1.1,v=[12.2,20.6]); data.append(dd)` is equivalent to the first `append` statement above. This is useful, for example, if the data comes from a function (that returns a dictionary).

One can also append data key-by-key: for example, `data.append('s',1.1); data.append('v',[12.2,20.6])` is equivalent to the first `append` in the example above. One could also achieve this with, for example, `data['s'].append(1.1); data['v'].append([12.2,20.6])`, since each dictionary value is a list, but `gvar.Dataset`'s `append` checks for consistency between the new data and data already collected and so is preferable.

Use `extend` in place of `append` to add data in batches: for example,

```
>>> data = Dataset()
>>> data.extend(s=[1.1,0.8],v=[[12.2,20.6],[14.1,19.2]])
>>> data.extend(s=[0.95,0.91],v=[[10.3,19.7],[8.2,21.0]])
>>> print(data['s'])        # 4 random values of s
[ 1.1, 0.8, 0.95, 0.91]
```

gives the same dataset as the first example above.

A `Dataset` can also be created from a file where every line is a new random sample. The data in the first example above could have been stored in a file with the following content:

```
# file: datafile
s 1.1
v [12.2,20.6]
s 0.8
v [14.1,19.2]
s 0.95
v [10.3,19.7]
s 0.91
v [8.2,21.0]
```

Lines that begin with # are ignored. Assuming the file is called `datafile`, we create a dataset identical to that above using the code:

```
>>> data = Dataset('datafile')
>>> print(data['s'])
[ 1.1, 0.8, 0.95, 0.91]
```

Data can be binned while reading it in, which might be useful if there the data set is huge. To bin the data contained in file `datafile` in bins of binsize 2 we use:

```
>>> data = Dataset('datafile',binsize=2)
>>> print(data['s'])
[0.95, 0.93]
```

The keys read from a data file are restricted to those listed in keyword `keys` and those that are matched (or partially matched) by regular expression `grep` if one or other of these is specified: for example,

```
>>> data = Dataset('datafile')
>>> print([k for k in a])
['s', 'v']
>>> data = Dataset('datafile',keys=['v'])
>>> print([k for k in a])
['v']
>>> data = Dataset('datafile',grep='[^v]')
>>> print([k for k in a])
['s']
>>> data = Dataset('datafile',keys=['v'],grep='[^v]')
```

```
>>> print([k for k in a])
[]
```

`Dataset`s can also be constructed from dictionaries, other `Dataset`s, or lists of key-data tuples. For example,

```
>>> data = Dataset('datafile')
>>> data_binned = Dataset(data, binsize=2)
>>> data_v = Dataset(data, keys=['v'])
```

reads data from file `'datafile'` into `Dataset` `data`, and then creates a new `Dataset` with the data binned (`data_binned`), and another that only containes the data with key `'v'` (`data_v`).

The main attributes and methods are:

samplesize
Smallest number of samples for any key.

append(**args*, ***kargs*)
Append data to dataset.

There are three equivalent ways of adding data to a dataset `data`: for example, each of

```
data.append(n=1.739,a=[0.494,2.734])          # method 1

data.append(n,1.739)                          # method 2
data.append(a,[0.494,2.734])

dd = dict(n=1.739,a=[0.494,2.734])            # method 3
data.append(dd)
```

adds one new random number to `data['n']`, and a new vector to `data['a']`.

extend(**args*, ***kargs*)
Add batched data to dataset.

There are three equivalent ways of adding batched data, containing multiple samples for each quantity, to a dataset `data`: for example, each of

```
data.extend(n=[1.739,2.682],
            a=[[0.494,2.734],[ 0.172, 1.400]])   # method 1

data.extend(n,[1.739,2.682])                      # method 2
data.extend(a,[[0.494,2.734],[ 0.172, 1.400]])

dd = dict(n=[1.739,2.682],
            a=[[0.494,2.734],[ 0.172, 1.400]])   # method 3
data.extend(dd)
```

adds two new random numbers to `data['n']`, and two new random vectors to `data['a']`.

This method can be used to merge two datasets, whether or not they share keys: for example,

```
data = Dataset("file1")
data_extra = Dataset("file2")
data.extend(data_extra)    # data now contains all of data_extra
```

grep(*rexp*)
Create new dataset containing items whose keys match `rexp`.

Returns a new `gvar.dataset.Dataset`‘ containing only the items `self[k]` whose keys `k` match regular expression `rexp` (a string) according to Python module `re`:

```
>>> a = Dataset()
>>> a.append(xx=1.,xy=[10.,100.])
>>> a.append(xx=2.,xy=[20.,200.])
>>> print(a.grep('y'))
{'yy': [array([  10.,  100.]), array([  20.,  200.])]}
>>> print(a.grep('x'))
{'xx': [1.0, 2.0], 'xy': [array([  10.,  100.]), array([  20.,  200.])]}
>>> print(a.grep('x|y'))
{'xx': [1.0, 2.0], 'xy': [array([  10.,  100.]), array([  20.,  200.])]}
>>> print a.grep('[^y][^x]')
{'xy': [array([  10.,  100.]), array([  20.,  200.])]}
```

Items are retained even if `rexp` matches only part of the item's key.

slice(*sl*)
Create new dataset with `self[k] -> self[k][sl]`.

Parameter `sl` is a slice object that is applied to every item in the dataset to produce a new `gvar.Dataset`. Setting `sl = slice(0,None,2)`, for example, discards every other sample for each quantity in the dataset. Setting `sl = slice(100,None)` discards the first 100 samples for each quantity.

arrayzip(*template*)
Merge lists of random data according to `template`.

`template` is an array of keys in the dataset, where the shapes of `self[k]` are the same for all keys `k` in `template`. `self.arrayzip(template)` merges the lists of random numbers/arrays associated with these keys to create a new list of (merged) random arrays whose layout is specified by `template`: for example,

```
>>> d = Dataset()
>>> d.append(a=1,b=10)
>>> d.append(a=2,b=20)
>>> d.append(a=3,b=30)
>>> print(d)            # three random samples each for a and b
{'a': [1.0, 2.0, 3.0], 'b': [10.0, 20.0, 30.0]}
>>> # merge into list of 2-vectors:
>>> print(d.arrayzip(['a','b']))
[[  1.  10.]
 [  2.  20.]
 [  3.  30.]]
>>> # merge into list of (symmetric) 2x2 matrices:
>>> print(d.arrayzip([['b','a'],['a','b']]))
[[[ 10.   1.]
  [  1.  10.]]

 [[ 20.   2.]
  [  2.  20.]]

 [[ 30.   3.]
  [  3.  30.]]]
```

The number of samples in each merged result is the same as the number samples for each key (here 3). The keys used in this example represent scalar quantities; in general, they could be either scalars or arrays (of any shape, so long as all have the same shape).

trim()
Create new dataset where all entries have same sample size.

toarray()
Copy `self` but with `self[k]` as numpy arrays.

CHAPTER

FOUR

NUMERICAL ANALYSIS MODULES IN GVAR

gvar.GVars can be used in many numerical algorithms, to propagates errors through the algorithm. A code that is written in pure Python is likely to work well with gvar.GVars, perhaps with minor modifications. Here we describe some sample numerical codes, included in gvar, that have been adapted to work with gvar.GVars, as well as with floats. More examples will follow with time.

4.1 Cubic Splines

The module gvar.cspline implements a class for smoothing and/or interpolating one-dimensional data using cubic splines:

class gvar.cspline.**CSpline**(*x*, *y*, *deriv=(None*, *None)*, *warn=True*)

Cubic spline resentation of a function.

Given N values of a function y[i] at N points x[i] for i=0..N-1 (the 'knots' of the spline), the code

```
from gvar.cspline import CSpline

f = CSpline(x, y)
```

defines a function f such that: a) f(x[i]) = y[i] for all i; and b) f(x) is continuous, as are its first and second derivatives. Function f(x) is a cubic polynomial between the knots x[i].

CSpline(x,y) creates a *natural spline*, which has zero second derivative at the end points, x[0] and x[-1]. More generally one can specify the derivatives of f(x) at one or both of the endpoints:

```
f = CSpline(x, y, deriv=[dydx_i, dydx_f])
```

where dydx_i is the derivative at x[0] and dydx_f is the derivative at x[-1]. Replacing either (or both) of these with None results in a derivative corresponding to zero second derivative at that boundary (i.e., a *natural* boundary).

Derivatives and integrals of the spline function can also be evaluated:

f.D(x) — first derivative at x;

f.D2(x) — second derivative at x;

f.integ(x) — integral from x[0] to x.

Parameters

- **x** (*1-d sequence of numbers*) – The knots of the spline, where the function values are specified.
- **y** (*1-d sequence*) – Function values at the locations specified by x[i].

- **deriv** (*2-component sequence*) – Derivatives at initial and final boundaries of the region specified by `x[i]`. Default value is `None` for each boundary.
- **warn** – If `True`, warnings are generated when the spline function is called for `x` values that fall outside of the original range of `xs` used to define the spline. Default value is `True`; out-of-range warnings are suppressed if set to `False`.

4.2 Ordinary Differential Equations

The module `gvar.ode` implements two classes for integrating systems of first-order differential equations using an adaptive Runge-Kutta algorithm. One integrates scalar- or array-valued equations, while the other integrates dictionary-valued equations:

class `gvar.ode.`**`Integrator`**(*deriv*, *tol=1e-05*, *h=None*, *hmin=None*, *analyzer=None*)

Integrate `dy/dx = deriv(x,y)`.

An `Integrator` object `odeint` integrates `dy/dx = f(x,y)` to obtain `y(x1)` from `y0 = y(x0)`. `y` and `f(x,y)` can be scalars or `numpy` arrays. Typical usage is illustrated by the following code for integrating `dy/dx = y`:

```
from gvar.ode import Integrator

def f(x, y):
    return y

odeint = Integrator(deriv=f,  tol=1e-8)
y0 = 1.
y1 = odeint(y0, interval=(0, 1.))
y2 = odeint(y1, interval=(1., 2.))
...
```

Here the first call to `odeint` integrates the differential equation from `x=0` to `x=1` starting with `y=y0` at `x=0`; the result is `y1=exp(1)`, of course. Similarly the second call to `odeint` continues the integration from `x=1` to `x=2`, giving `y2=exp(2)`.

An alternative interface creates a new function which is the solution of the differential equation for specific initial conditions. The code above could be rewritten:

```
x0 = 0.          # initial conditions
y0 = 1.
y = Integrator(deriv=f, tol=1e-8).solution(x0, y0)
y1 = y(1)
y2 = y(2)
...
```

Here method `Integrator.solution()` returns a function `y(x)` where: a) `y(x0) = y0`; and b) `y(x)` uses the integator to integrate the differential equation to point `x` starting from the last point at which `y` was evaluated (or from `x0` for the first call to `y(x)`).

The integrator uses an adaptive Runge-Kutta algorithm that adjusts the integrator's step size to obtain relative accuracy `tol` in the solution. An initial step size can be set in the `Integrator` by specifying parameter `h`. A minimum step size `hmin` can also be specified; the `Integrator` raises an exception if the step size becomes smaller than `hmin`. The `Integrator` keeps track of the number of good steps, where `h` is increased, and the number of bad steps, where `h` is decreased and the step is repeated: `odeint.ngood` and `odeint.nbad`, respectively.

A custom criterion for step-size changes can be implemented by specifying a function for parameter delta. This is a function `delta(yerr, y, delta_y)` — of the estimated error `yerr` after a given step, the proposed

value for `y`, and the proposed change `delta_y` in `y` — that returns a number to compare with tolerance `tol`. The step size is decreased and the step repeated if `delta(yerr, y, delta_y) > tol`; otherwise the step is accepted and the step size increased. The default definition of `delta` is roughly equivalent to:

```
import numpy as np
import gvar as gv

def delta(yerr, y, delta_y):
    return np.max(
        np.fabs(yerr) / (np.fabs(y) + np.fabs(delta_y) + gv.ode.TINY)
        )
```

A custom definition can be used to allow an `Integrator` to work with data types other than floats or `numpy` arrays of floats. All that is required of the data type is that it support ordinary arithmetic. Therefore, for example, defining `delta(yerr, y, delta_y)` with `np.abs()` instead of `np.fabs()` allows `y` to be complex valued. (Actually the default `delta` allows this as well.)

An analyzer `analyzer(x,y)` can be specified using parameter `analyzer`. This function is called after every full step of the integration, with the current values of `x` and `y`. Objects of type `gvar.ode.Solution` are examples of (simple) analyzers.

Parameters

- **deriv** – Function of `x` and `y` that returns `dy/dx`. The return value should have the same shape as `y` if arrays are used.
- **tol** (*float*) – Relative accuracy in `y` relative to `|y| + h|dy/dx|` for each step in the integration. Any integration step that achieves less precision is repeated with a smaller step size. The step size is increased if precision is higher than needed. Default is 1e-5.
- **h** (*float or None*) – Absolute value of initial step size. The default value equals the entire width of the integration interval.
- **hmin** (*float or None*) – Smallest step size allowed. An exception is raised if a smaller step size is needed. This is mostly useful for preventing infinite loops caused by programming errors. The default value is zero (which does *not* prevent infinite loops).
- **delta** – Function `delta(yerr, y, delta_y)` that returns a number to be compared with `tol` at each integration step: if it is larger than `tol`, the step is repeated with a smaller step size; if it is smaller the step is accepted and a larger step size used for the subsequent step. Here `yerr` is an estimate of the error in `y` on the last step; `y` is the proposed value; and `delta_y` is the change in `y` over the last step.
- **analyzer** – Function of `x` and `y` that is called after each step of the integration. This can be used to analyze intermediate results.

class `gvar.ode.`**`DictIntegrator`**(*deriv*, *tol=1e-05*, *h=None*, *hmin=None*, *analyzer=None*)

Integrate `dy/dx = deriv(x,y)` where `y` is a dictionary.

An `DictIntegrator` object `odeint` integrates `dy/dx = f(x,y)` to obtain `y(x1)` from `y0 = y(x0)`. `y` and `f(x,y)` are dictionary types having the same keys, and containing scalars and/or `numpy` arrays as values. Typical usage is:

```
from gvar.ode import DictIntegrator

def f(x, y):
    ...

odeint = DictIntegrator(deriv=f,  tol=1e-8)
y1 = odeint(y0, interval=(x0, x1))
```

```
y2 = odeint(y1, interval=(x1, x2))
...
```

The first call to `odeint` integrates from `x=x0` to `x=x1`, returning `y1=y(x1)`. The second call continues the integration to `x=x2`, returning `y2=y(x2)`. Any of the initial parameters can be reset in the calls to `odeint`: for example,

```
y2 = odeint(y1, interval=(x1, x2), tol=1e-10)
```

The integrator uses an adaptive Runge-Kutta algorithm that adjusts the integrator's step size to obtain relative accuracy `tol` in the solution. An initial step size can be set in the `DictIntegrator` by specifying parameter `h`. A minimum ste psize `hmin` can also be specified; the `Integrator` raises an exception if the step size becomes smaller than `hmin`. The `DictIntegrator` keeps track of the number of good steps, where `h` is increased, and the number of bad steps, where `h` is decreases and the step is repeated: `odeint.ngood` and `odeint.nbad`, respectively.

An analyzer `analyzer(x,y)` can be specified using parameter `analyzer`. This function is called after every full step of the integration with the current values of `x` and `y`. Objects of type `gvar.ode.Solution` are examples of (simple) analyzers.

Parameters

- **deriv** – Function of `x` and `y` that returns `dy/dx`. The return value should be a dictionary with the same keys as `y`, and values that have the same shape as the corresponding values in `y`.
- **tol** (*float*) – Relative accuracy in `y` relative to `|y| + h|dy/dx|` for each step in the integration. Any integration step that achieves less precision is repeated with a smaller step size. The step size is increased if precision is higher than needed.
- **h** (*float*) – Absolute value of initial step size. The default value equals the entire width of the integration interval.
- **hmin** (*float*) – Smallest step size allowed. An exception is raised if a smaller step size is needed. This is mostly useful for preventing infinite loops caused by programming errors. The default value is zero (which does *not* prevent infinite loops).
- **analyzer** – Function of `x` and `y` that is called after each step of the integration. This can be used to analyze intermediate results.

A simple analyzer class is:

class `gvar.ode.`**`Solution`**

ODE analyzer for storing intermediate values.

Usage: eg, given

```
odeint = Integrator(...)
soln = Solution()
y0 = ...
y = odeint(y0, interval=(x0, x), analyzer=soln)
```

then the `soln.x[i]` are the points at which the integrator evaluated the solution, and `soln.y[i]` is the solution of the differential equation at that point.

4.3 Power Series

This module provides tools for manipulating power series approximations of functions. A function's power series is specified by the coefficients in its Taylor expansion with respect to an independent variable, say `x`:

```
f(x) = f(0) + f'(0)*x + (f''(0)/2)*x**2 + (f'''(0)/6)*x**3 + ...
     = f0 + f1*x + f2*x**2 + f3*x**3 + ...
```

In practice a power series is different from a polynomial because power series, while infinite order in principle, are truncated at some finite order in numerical applications. The `order` of a power series is the highest power of `x` that is retained in the approximation; coefficients for still higher-order terms are assumed to be unknown (as opposed to zero).

Taylor's theorem can be used to generate power series for functions of power series:

```
g(f(x)) = g(f0) + g'(f0)*(f(x)-f0) + (g''(f0)/2)*(f(x)-f0)**2 + ...
        = g0 + g1*x + g2*x**2 + ...
```

This allows us to define a full calculus for power series, where arithmetic expressions and (sufficiently differentiable) functions of power series return new power series.

4.3.1 Power series arithmetic

Class `PowerSeries` provides a numerical implementation of the power series calculus. `PowerSeries([f0,f1,f2,f3...])` is a numerical representation of a power series with coefficients `f0, f1, f2, f3...` (as in `f(x)` above). Thus, for example, we can define a 4th-order power series approximation `f` to `exp(x)=1+x+x**2/2+...` using

```
>>> from gvar.powerseries import *
>>> f = PowerSeries([1., 1., 1/2., 1/6., 1/24.])
>>> print f              # print the coefficients
[ 1.          1.          0.5         0.16666667  0.04166667]
```

Arithmetic expressions involving instances of class `PowerSeries` are themselves `PowerSeries` as in, for example,

```
>>> print 1/f            # power series for exp(-x)
[ 1.         -1.          0.5        -0.16666667  0.04166667]
>>> print log(f)         # power series for x
[ 0.  1.  0. -0.  0.]
>>> print f/f            # power series for 1
[ 1.  0.  0.  0.  0.]
```

The standard arithmetic operators (`+,-,*,/,=,**`) are supported, as are the usual elementary functions (`exp, log, sin, cos, tan ...`). Different `PowerSeries` can be combined arithmetically to create new `PowerSeries`; the order of the result is that of the operand with the lowest order.

`PowerSeries` can be differentiated and integrated:

```
>>> print f.deriv()      # derivative of exp(x)
[ 1.          1.          0.5         0.16666667]
>>> print f.integ()      # integral of exp(x)  (from x=0)
[ 0.          1.          0.5         0.16666667  0.04166667  0.00833333]
```

Each `PowerSeries` represents a function. The `PowerSeries` for a function of a function is easily obtained. For example, assume `f` represents function `f(x)=exp(x)`, as above, and `g` represents `g(x)=log(1+x)`:

```
>>> g = PowerSeries([0, 1, -1/2., 1/3., -1/4.])
```

Then `f(g)` gives the `PowerSeries` for `exp(log(1+x)) = 1 + x`:

```
>>> print f(g)
[  1.0000e+00   1.0000e+00   0.0000e+00  -2.7755e-17 -7.6327e-17]
```

Individual coefficients from the powerseries can be accessed using array-element notation: for example,

```
>>> print f[0], f[1], f[2], f[3]
1.0 1.0 0.5 0.166666666667
>>> f[0] = f[0] - 1.
>>> print f              # f is now the power series for exp(x)-1
[ 0.          1.          0.5         0.16666667  0.04166667]
```

4.3.2 Numerical evaluation of power series

The power series can also be evaluated for a particular numerical value of x: continuing the example,

```
>>> x = 0.01
>>> print f(x)           # should be exp(0.01)-1 approximately
0.0100501670833

>>> print exp(x)-1       # verify that it is
0.0100501670842
```

The independent variable `x` could be of any arithmetic type (it need not be a `float`).

4.3.3 Taylor expansions of Python functions

`PowerSeries` can be used to compute Taylor series for more-or-less arbitrary pure-Python functions provided the functions are locally analytic (or at least sufficiently differentiable). To compute the `N`-th order expansion of a Python function `g(x)`, first create a `N`-th order `PowerSeries` variable that represents the expansion parameter: say, `x = PowerSeries([0.,1.],order=N)`. The Taylor series for function `g` is then given by `g_taylor = g(x)` which is a `PowerSeries` instance. For example, consider:

```
>>> from gvar.powerseries import *
>>> def g(x):            # an example of a Python function
...     return 0.5/sqrt(1+x) + 0.5/sqrt(1-x)
...
>>> x = PowerSeries([0.,1.],order=5)     # Taylor series for x
>>> print x
[ 0.  1.  0.  0.  0.  0.]
>>> g_taylor = g(x)      # Taylor series for g(x) about x=0
>>> print g_taylor
[ 1.          0.          0.375       0.          0.2734375   0.        ]
>>> exp_taylor = exp(x) # Taylor series for exp(x) about x=0
>>> print exp_taylor
[ 1.          1.          0.5         0.16666667  0.04166667  0.00833333]
```

class `gvar.powerseries.`**`PowerSeries`**(*c=None*, *order=None*)

Power series representation of a function.

The power series created by `PowerSeries(c)` corresponds to:

```
c[0] + c[1]*x + c[2]*x**2 + ... .
```

The order of the power series is normally determined by the length of the input list `c`. This can be overridden by specifying the order of the power series using the `order` parameter. The list of `c[i]`s is then padded with zeros if `c` is too short, or truncated if it is too long. Omitting `c` altogether results in a power series all of whose coefficients are zero. Individual series coefficients are accessed using array/list notation: for example, the 3rd-order coefficient of `PowerSeries p` is `p[3]`. The order of `p` is `p.order`. `PowerSeries` should work for coefficients of any data type that supports ordinary arithmetic.

Arithmetic expressions of `PowerSeries` variables yield new `PowerSeries` results that represent the power series expansion of the expression. Expressions can include the standard mathematical functions (`log, exp, sqrt, sin, cos, tan...`). `PowerSeries` can also be differentiated (`p.deriv()`) and integrated (`p.integ()`).

Parameters

- **c** (*list or array*) – Power series coefficients (optional if parameter *order* specified).
- **order** (*integer*) – Highest power in power series (optional if parameter *c* specified).

deriv(*n=1*)

Compute *n*-th derivative of `self`.

Parameters **n** (*positive integer*) – Number of derivatives.

Returns *n*-th derivative of `self`.

integ(*n=1*, *x0=None*)

Compute *n*-th indefinite integral of `self`.

If *x0* is specified, then the definite integral, integrating from point *x0*, is returned.

Parameters

- **n** (*integer*) – Number of integrations.
- **x0** – Starting point for definite integral (optional).

Returns *n*-th integral of `self`.

order

Highest power in power series.

CHAPTER

FIVE

LSQFIT - NONLINEAR LEAST SQUARES FITTING

5.1 Introduction

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

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

A typical code sequence has the structure:

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

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

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

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

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

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

The `lsqfit` tutorial contains extended explanations and examples.

5.2 Formal Background

The formal structure structure of a least-squares problem involves fitting input data y_i with functions $f_i(p)$ by adjusting fit parameters p_a to minimize

$$\begin{aligned}\chi^2 &\equiv \sum_{ij} \Delta y(p)_i \, (\mathrm{cov}_y^{-1})_{ij} \, \Delta y(p)_j \\ &\equiv (\Delta y(p))^{\mathrm{T}} \cdot \mathrm{cov}_y^{-1} \cdot \Delta y(p)\end{aligned}$$

where cov_y is the covariance matrix for the input data and

$$\Delta y(p)_i \equiv f_i(p) - y_i.$$

There are generally two types of input data — actual data and prior information for each fit parameter — but we lump these together here since they enter in the same way (that is, the sums over i and j are over all data and priors).

The best-fit values $\overline{p}_a$ for the fit parameters are those that minimize χ^2:

$$(\partial_a \Delta y(\overline{p}))^{\mathrm{T}} \cdot \mathrm{cov}_y^{-1} \cdot \Delta y(\overline{p}) = 0$$

where the derivatives are $\partial_a = \partial/\partial\overline{p}_a$. The covariance matrix cov_p for these is obtained (approximately) from

$$(\mathrm{cov}_{\mathrm{p}}^{-1})_{ab} \equiv (\partial_a \Delta y(\overline{p}))^{\mathrm{T}} \cdot \mathrm{cov}_y^{-1} \cdot (\partial_b \Delta y(\overline{p})).$$

Consequently the variance for any function $g(\overline{p})$ of the best-fit parameters is given by (approximately)

$$\sigma_g^2 = (\partial g(\overline{p}))^{\mathrm{T}} \cdot \mathrm{cov}_p \cdot \partial g(\overline{p})$$

The definition of the covariance matrix implies that it and any variance σ_g^2 derived from it depend linearly (approximately) on the elements of the input data covariance matrix cov_y, at least when errors are small:

$$\sigma_g^2 \approx \sum_{ij} c(\overline{p})_{ij} \, (\mathrm{cov}_y)_{ij}$$

This allows us to associate different portions of the output error σ_g^2 with different parts of the input error cov_y, creating an "error budget" for $g(\overline{p})$. Such information helps pinpoint the input errors that most affect the output errors for any particular quantity $g(\overline{p})$, and also indicates how those output errors might change for a given change in input error.

The relationship between the input and output errors is only approximately linear because the coefficients in the expansion depend upon the best-fit values for the parameters, and these depend upon the input errors — but only weakly when errors are small. Neglecting such variation in the parameters, the error budget for any quantity is easily computed using

$$\frac{\partial(\mathrm{cov}_p)_{ab}}{\partial(\mathrm{cov}_y)_{ij}} = D_{ai} \, D_{bj}$$

where

$$D_{ai} \equiv (\mathrm{cov}_p \cdot \partial\Delta y \cdot \mathrm{cov}_y^{-1})_{ai}$$

and, trivially, $\mathrm{cov}_p = D \cdot \mathrm{cov}_y \cdot D^{\mathrm{T}}$.

This last formula suggests that

$$\frac{\partial\overline{p}_a}{\partial y_i} = D_{ai}.$$

This relationship is true in the limit of small errors, as is easily derived from the minimum condition for the fit, which defines (implicitly) $\overline{p}_a(y)$: Differentiating with respect to y_i we obtain

$$(\partial_a \Delta y(\overline{p}))^{\mathrm{T}} \cdot \mathrm{cov}_y^{-1} \cdot \frac{\partial \Delta y(\overline{p})}{\partial y_i} = 0$$

where we have ignored terms suppressed by a factor of $\Delta y(p)$. This leads immediately to the relationship above.

The data's covariance matrix cov_y is sometimes rather singular, making it difficult to invert. This problem is dealt with using an SVD cut: the covariance matrix is diagonalized, some number of the smallest (and therefore least-well determined) eigenvalues and their eigenvectors are discarded, and the inverse matrix is reconstituted from the eigenmodes that remain. (Instead of discarding modes one can replace their eigenvalues by the smallest eigenvalue that is retained; this is less conservative and usually leads to more accurate results.)

5.3 nonlinear_fit Objects

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

Nonlinear least-squares fit.

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

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

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

Parameters

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

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

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

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

  `data = x, ymean, ysdev` `x` is the independent data that is passed to the fit function with the fit parameters: `fcn(x, p)`. `ymean` is an array containing the mean values of the fit data. `ysdev` is an array containing the standard deviations

of the fit data; `ysdev.shape` equals `ymean.shape`. The data are assumed to be uncorrelated. The fit function must return an array having the same shape as `ymean`.

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

- **fcn** (*function*) – The function to be fit to `data`. It is either a function of the independent data `x` and the fit parameters `p` (`fcn(x, p)`), or a function of just the fit parameters (`fcn(p)`) when there is no `x` data or `x=False`. The parameters are tuned in the fit until the function returns values that agree with the `y` data to within the `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** (dictionary, array, or `None`) – A dictionary (or array) containing *a priori* estimates for all parameters `p` used by fit function `fcn(x, p)` (or `fcn(p)`). Fit parameters `p` are stored in a dictionary (or array) with the same keys and structure (or shape) as `prior`. The default value is `None`; `prior` must be defined if `p0` is `None`.
- **p0** (dictionary, array, string or `None`) – Starting values for fit parameters in fit. `lsqfit.nonlinear_fit` adjusts `p0` to make it consistent in shape and structure with `prior` when the latter is specified: elements missing from `p0` are filled in using `prior`, and elements in `p0` that are not in `prior` are discarded. If `p0` is a string, it is taken as a file name and `lsqfit.nonlinear_fit` attempts to read starting values from that file; best-fit parameter values are written out to the same file after the fit (for priming future fits). If `p0` is `None` or the attempt to read the file fails, starting values are extracted from `prior`. The default value is `None`; `p0` must be defined if `prior` is `None`.
- **svdcut** (`None` or `float`) – If `svdcut` is nonzero (not `None`), *svd* cuts are applied to every block-diagonal sub-matrix of the covariance matrix for the data `y` and `prior` (if there is a prior). The blocks are first rescaled so that all diagonal elements equal 1 – that is, the blocks are replaced by the correlation matrices for the corresponding subsets of variables. Then, if `svdcut > 0`, eigenvalues of the rescaled matrices that are smaller than `svdcut` times the maximum eigenvalue are replaced by `svdcut` times the maximum eigenvalue. This makes the covariance matrix less singular and less susceptible to roundoff error. When `svdcut < 0`, eigenvalues smaller than `|svdcut|` times the maximum eigenvalue are discarded and the corresponding components in `y` and `prior` are zeroed out.
- **debug** (*boolean*) – Set to `True` for extra debugging of the fit function and a check for roundoff errors. (Default is `False`.)
- **fitterargs** – Dictionary of arguments passed on to `lsqfit.multifit`, which does the fitting.

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

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

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

dof

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

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

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

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

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

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

transformed_p
Same as `fit.p` but augmented to include the transforms of any log-normal or other parameter implemented using decorator `lsqfit.transform_p`. In the case of a log-normal variable `fit.p['logXX']`, for example, `fit.transformed_p['XX']` is defined equal to `exp(fit.p['logXX'])`.

p0
The parameter values used to start the fit.

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

svdcorrection
An array containing the (flattened) SVD corrections, if any, added to the fit data `y` and the prior `prior`.

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

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

time
CPU time (in secs) taken by fit.

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

prior
Prior used in the fit. This may differ from the input prior if an SVD cut is used. It is either a dictionary

(gvar.BufferDict) or an array (numpy.ndarray), depending upon the input. Equals None if no prior was specified.

x

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

y

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

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

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

Formats fit output details into a string for printing.

The output tabulates the chi**2 per degree of freedom of the fit (chi2/dof), the number of degrees of freedom, the logarithm of the Gaussian Bayes Factor for the fit (logGBF), and the number of fit-algorithm iterations needed by the fit. Optionally, it will also list the best-fit values for the fit parameters together with the prior for each (in [] on each line). It can also list all of the data and the corresponding values from the fit. At the end it lists the SVD cut, the number of eigenmodes modified by the SVD cut, the relative and absolute tolerances used in the fit, and the time in seconds needed to do the fit.

Parameters

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

Returns String containing detailed information about fit.

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

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

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

Parameters

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

- **percent** (*boolean*) – Tabulate % errors if `percent is True`; otherwise tabulate the errors themselves.
- **colwidth** (*positive integer*) – Width of each column.

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

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

Tabulate `gvar.GVar`s in `outputs`.

Parameters

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

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

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

Iterator that returns simulation copies of a fit.

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

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

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

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

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

5.4 Functions

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

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

Parameters

- **z0** (*array*) – Starting point for search.
- **fitargs** (*function*) – Function of array `z` that determines which fit parameters to use. The function returns these as an argument dictionary for `lsqfit.nonlinear_fit()`.
- **minargs** (*dictionary*) – Optional argument dictionary, passed on to `lsqfit.multiminex`, which finds the minimum.

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

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

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

The following function attributes are also set:

wavg.**chi2**
chi**2 for weighted average.

wavg.**dof**
Effective number of degrees of freedom.

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

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

wavg.**time**
Time required to do average.

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

wavg.**fit**
Fit output from average.

These same attributes are also attached to the output gvar.GVar, array or dictionary from gvar.wavg().

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.

5.5 Utility Classes

class lsqfit.**transform_p**(*priorkeys*, *has_x=False*)
Decorate fit function to allow log/sqrt-normal priors.

This decorator can be applied to fit functions whose parameters are stored in a dictionary-like object. It searches the parameter keys for string-valued keys of the form "log(XX)", "logXX", "sqrt(XX)", or "sqrtXX" where "XX" is an arbitrary string. For each such key it adds a new entry to the parameter dictionary with key "XX" where:

```
p["XX"] = exp(p[k])    for k = "log(XX)" or "logXX"
```

or

```
p["XX"] = p[k] ** 2    for k = "sqrt(XX)" or "sqrtXX"
```

This means that the fit function can be expressed entirely in terms of `p["XX"]` even if the actual fit parameter is the logarithm or square root of that quantity. Since fit parameters have gaussian/normal priors, `p["XX"]` has a log-normal or "sqrt-normal" distribution in the first or second cases above, respectively. In either case `p["XX"]` is guaranteed to be postiive.

This is a convenience function. It allows for the rapid replacement of a fit parameter by its logarithm or square root without having to rewrite the fit function — only the prior need be changed. The decorator needs to be told if the fit function has an `x` as its first argument, followed by the parameters `p`:

```
@lsqfit.transform_p(prior.keys(), has_x=True)
def fitfcn(x, p):
    ...
```

versus

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

A list of the specific keys that need transforming can be used instead of the list of all keys (`prior.keys()`). The decorator assigns a copy of itself to the function as an attribute: `fitfcn.transform_p`.

Parameters

- **priorkeys** (*sequence*) – The keys in the prior that are to be tranformed. Other keys can be in `priorkeys` provided they do not begin with `'log'` or `'sqrt'` — they are ignored.
- **has_x** – Set equal to `True` if the fit function is a function of `x` and parameters `p` (*i.e.*, `f(x,p)`). Set equal to `False` if the fit function is a function only of the parameters (*i.e.*, `f(p)`). Default is `False`.
- **pkey** (*string or None*) – Name of the parameters-variable in the argument keyword dictionary of the fit function. Default value is `None`; one of `pkey` or `pindex` must be specified (i.e., `not None`), unless the fit function has only a single argument.

transform(*p*)

Create transformed copy of dictionary `p`.

Create a copy of parameter-dictionary `p` that includes new entries for each `"logXX"`, etc entry corresponding to `"XX"`. The values in `p` can be any type that supports logarithms, exponentials, and arithmetic.

untransform(*p*)

Undo `self.transform(p)`.

Reconstruct `p0` where `p == self.transform(p0)`; that is remove entries for keys `"XX"` that were added by by `transform_p.transform()` (because `"logXX"` or `"sqrtXX"` or ... appeared in `p0`).

static paramkey(*k*)

Return parameter key corresponding to prior-key `k`.

Strip off any `"log"` or `"sqrt"` prefix.

static priorkey(*prior*, *k*)

Return key in `prior` corresponding to `k`.

Add in `"log"` or `"sqrt"` as needed to find a key in `prior`.

class lsqfit.**multifit**(*x0*, *n*, *f*, *reltol=1e-4*, *abstol=0*, *maxit=1000*, *alg='lmsder'*, *analyzer=None*)

Fitter for nonlinear least-squares multidimensional fits.

Parameters

- **x0** (`numpy` array of floats) – Starting point for minimization.
- **n** (*positive integer*) – Length of vector returned by the fit function `f(x)`.
- **f** (*function*) – Fit function: `multifit` minimizes `sum_i f_i(x)**2` by varying parameters `x`. The parameters are a 1-d `numpy` array of either numbers or `gvar.GVar`s.
- **reltol** (*float*) – The fit stops when `|dx_i| < abstol + reltol * |x_i|`; default value is `1e-4`.
- **abstol** (*float*) – The fit stops when `|dx_i| < abstol + reltol * |x_i|`; default value is `0.0`.
- **maxit** (*integer*) – Maximum number of iterations in search for minimum; default is 1000.
- **alg** (*string*) – *GSL* algorithm to use for minimization. Two options are currently available: `"lmsder"`, the scaled *LMDER* algorithm (default); and `"lmder"`, the unscaled *LMDER* algorithm.
- **analyzer** (*function*) – Optional function of `x, [...f_i(x)...], [[..df_ij(x)..]]` which is called after each iteration. This can be used to inspect intermediate steps in the minimization, if needed.

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

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

cov
Covariance matrix at the minimum point.

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

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

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

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

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

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

Parameters

- **x0** (`numpy` array of floats) – Starting point for minimization search.
- **f** (*function*) – Function `f(x)` to be minimized by varying vector `x`.
- **tol** (*float*) – Minimization stops when `x` has converged to with tolerance `tol`; default is `1e-4`.
- **maxit** (*integer*) – Maximum number of iterations in search for minimum; default is 1000.
- **step** (*number*) – Initial step size to use in varying components of `x`; default is 1.
- **alg** (*string*) – *GSL* algorithm to use for minimization. Three options are currently available: `"nmsimplex"`, Nelder Mead Simplex algorithm; `"nmsimplex2"`, an improved version

of "nmsimplex" (default); and "nmsimplex2rand", a version of "nmsimplex2" with random shifts in the start position.

- **analyzer** (*function*) – Optional function of `x, f(x), it`, where `it` is the iteration number, which is called after each iteration. This can be used to inspect intermediate steps in the minimization, if needed.

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

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

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

nit
Number of iterations required to find most recent minimum.

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

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

5.6 Requirements

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

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

g

l

Symbols

A

B

C

D

E

F

G

Y