lsqfit Documentation

Release 4.2

G. P. Lepage

June 08, 2012

CONTENTS

Contents:

CHAPTER

ONE

OVERVIEW AND TUTORIAL

1.1 Introduction

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

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

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

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

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

# do the fit
fit = lsqfit.nonlinear_fit(data=(x, y), prior=prior, fcn=fcn)
print(fit.format(100))      # print standard summary of fit

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

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

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

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

```
>>> print y['data1']
[1.376 +- 0.0685565 2.01 +- 0.236643]
```

shows the means and standard deviations 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. In addition, there is an extra piece of input data, `y["b/a"]`, which indicates that `b/a` is `2.0 +- 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 output from the code sample above is:

```
Least Square Fit:
  chi2/dof [dof] = 0.17 [5]    Q = 0.97    logGBF = -5.2381    itns = 5

Parameters:
              a_    0.252798 +-    0.032          (     0.5 +-      0.5)
              b_    0.448762 +-    0.065          (     0.5 +-      0.5)

Fit:
        key          y_i       f(x_i)        dy_i
-----------------------------------------------
        b/a_            2       1.7752         0.5
      data1_        1.376       1.3467    0.068557
          _         2.01       2.0169     0.23664
      data2_        1.329       1.3467    0.068557
          _        1.582       1.6115     0.11662

Values:
                 a: 0.253(32)
               b/a: 1.775(298)
                 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 `1.775(298)`, which, because of correlations, is slightly more accurate than might be expected from the separate errors for `a` and `b`. The error budget, at the end, for each of these three quantities 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 last section of the code uses Python's `pickle` module to save the best-fit values of the parameters in a file for later use. They are recovered using `pickle` again:

```
>>> import pickle
>>> p = pickle.load(open("outputfile.p", "rb"))
```

```
>>> print(p['a'])
0.252798 +- 0.0323152
>>> print(p['b'])
0.448762 +- 0.0647224
>>> print(p['b']/p['a'])
1.77518 +- 0.298185
```

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

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

```
... as before ...

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

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

... as before ...
```

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

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.

1.2 Making Fake Data

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

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

```
import numpy as np

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

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

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

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

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

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

```
import gvar as gv

def make_data():                        # 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, 0.01)
    c = [gv.gvar(cr(), 0.01) 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)*noise
    return x, y
```

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

```
>>> print(cr)
0 +- 0.01
>>> print(cr())
0.00452180208286
>>> print(cr())
-0.00731564589737
```

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

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

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

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

The diagonal elements of the covariance matrix are the variances of the individual ys; the off-diagonal elements are a measure of the correlations `< (y[i]-<y[i]>) * (y[j]-<y[j]>) >`.

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

1.3 Basic Fits

Now that we have fit data, `x, y = make_data(100)`, we pretend ignorance of the exact functional relationship between `x` and `y` (*i.e.*, `y=f_exact(x)`). Typically we *do* know the functional form and have some *a priori* idea about the parameter values. The point of the fit is to improve our knowledge of the parameter values, beyond our *a priori* impressions, by analyzing the fit data. Here we see how to do this using the `lsqfit` module.

First we need code to represent the fit function. In this case we know that a sum of exponentials is appropriate, so we define the following Python function to represent the relationship between `x` and `y` in our fit:

```
import numpy as np

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

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

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

```
import lsqfit
import gvar as gv

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

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

```
>>> print(prior['a'])
[0.5 +- 0.5 0.5 +- 0.5 0.5 +- 0.5]
>>> print(prior['E'])
[1 +- 0.5 2 +- 0.5 3 +- 0.5]
```

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

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

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

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

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

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

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():                        # 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, 0.01)
    c = [gv.gvar(cr(), 0.01) 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)*noise
    return x, y

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

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

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

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

```
************************************* nexp = 3
Least Square Fit:
  chi2/dof [dof] = 6.4e+02 [15]    Q = 0    logGBF = -4876    itns = 33

Parameters:
              a_   0.0191246 +-  0.00089           (     0.5 +-      0.5)
               _   0.0237325 +-   0.0011           (     0.5 +-      0.5)
               _   0.0515777 +-   0.0024           (     0.5 +-      0.5)
              E_     1.04066 +-   0.0024           (       1 +-      0.5)
               _     2.06475 +-   0.0024           (       2 +-      0.5)
               _     3.72957 +-   0.0026           (       3 +-      0.5)

E1/E0 = 1.98408 +- 0.0024544   E2/E0 = 3.58385 +- 0.00628162
a1/a0 = 1.24094 +- 0.000263974   a2/a0 = 2.69693 +- 0.00126443

************************************* nexp = 4
Least Square Fit:
  chi2/dof [dof] = 0.57 [15]    Q = 0.9    logGBF = -74.426    itns = 291

Parameters:
              a_    0.401753 +-    0.004           (     0.5 +-      0.5)
               _    0.405533 +-   0.0042           (     0.5 +-      0.5)
               _     0.49513 +-   0.0072           (     0.5 +-      0.5)
               _       1.124 +-    0.012           (     0.5 +-      0.5)
              E_     0.90037 +-  0.00051           (       1 +-      0.5)
               _     1.80235 +-   0.0012           (       2 +-      0.5)
               _     2.77306 +-   0.0085           (       3 +-      0.5)
               _     4.38303 +-     0.02           (       4 +-      0.5)

E1/E0 = 2.00178 +- 0.00117831   E2/E0 = 3.07991 +- 0.00919665
a1/a0 = 1.00941 +- 0.00287022   a2/a0 = 1.23242 +- 0.0128117

************************************* nexp = 5
Least Square Fit:
  chi2/dof [dof] = 0.45 [15]    Q = 0.97    logGBF = -73.627    itns = 6

Parameters:
              a_    0.401829 +-    0.004           (     0.5 +-      0.5)
               _    0.404845 +-   0.0044           (     0.5 +-      0.5)
               _    0.477577 +-    0.026           (     0.5 +-      0.5)
               _    0.626663 +-     0.28           (     0.5 +-      0.5)
               _    0.617964 +-     0.35           (     0.5 +-      0.5)
              E_    0.900363 +-  0.00051           (       1 +-      0.5)
               _     1.80192 +-   0.0014           (       2 +-      0.5)
               _     2.75937 +-    0.022           (       3 +-      0.5)
               _     4.09341 +-     0.26           (       4 +-      0.5)
               _     4.94923 +-     0.48           (       5 +-      0.5)

E1/E0 = 2.00132 +- 0.00139785   E2/E0 = 3.06473 +- 0.0238493
a1/a0 = 1.0075 +- 0.00413287   a2/a0 = 1.18851 +- 0.0629341

************************************* nexp = 6
Least Square Fit:
  chi2/dof [dof] = 0.45 [15]    Q = 0.97    logGBF = -73.771    itns = 6

Parameters:
              a_    0.401835 +-    0.004           (     0.5 +-      0.5)
               _    0.404032 +-   0.0047           (     0.5 +-      0.5)
```

```
                _    0.460419 +-    0.041           (     0.5 +-      0.5)
                _    0.598159 +-     0.24           (     0.5 +-      0.5)
                _    0.471462 +-     0.37           (     0.5 +-      0.5)
                _    0.451949 +-     0.46           (     0.5 +-      0.5)
               E_    0.900353 +-  0.00051           (       1 +-      0.5)
                _     1.80145 +-   0.0017           (       2 +-      0.5)
                _     2.74537 +-    0.034           (       3 +-      0.5)
                _     3.97765 +-     0.32           (       4 +-      0.5)
                _     4.95873 +-     0.49           (       5 +-      0.5)
                _     6.00919 +-      0.5           (       6 +-      0.5)

E1/E0 = 2.00083 +- 0.00166713   E2/E0 = 3.04921 +- 0.0372569
a1/a0 = 1.00547 +- 0.00554293   a2/a0 = 1.14579 +- 0.101026

************************************ nexp = 7
Least Square Fit:
  chi2/dof [dof] = 0.45 [15]    Q = 0.96    logGBF = -73.873    itns = 6

Parameters:
               a_    0.401835 +-    0.004           (     0.5 +-      0.5)
                _    0.403622 +-   0.0048           (     0.5 +-      0.5)
                _    0.452267 +-    0.047           (     0.5 +-      0.5)
                _    0.598425 +-     0.22           (     0.5 +-      0.5)
                _    0.416291 +-     0.37           (     0.5 +-      0.5)
                _    0.417308 +-     0.46           (     0.5 +-      0.5)
                _    0.459911 +-     0.49           (     0.5 +-      0.5)
               E_    0.900348 +-  0.00051           (       1 +-      0.5)
                _     1.80122 +-   0.0018           (       2 +-      0.5)
                _     2.73849 +-    0.039           (       3 +-      0.5)
                _     3.93758 +-     0.33           (       4 +-      0.5)
                _     4.96349 +-     0.49           (       5 +-      0.5)
                _     6.01884 +-      0.5           (       6 +-      0.5)
                _     7.01563 +-      0.5           (       7 +-      0.5)

E1/E0 = 2.00058 +- 0.00179764   E2/E0 = 3.04159 +- 0.0430577
a1/a0 = 1.00445 +- 0.00620982   a2/a0 = 1.1255 +- 0.116229
                                      .
                                      .
                                      .

 ************************************ nexp = 19
 Least Square Fit:
   chi2/dof [dof] = 0.46 [15]    Q = 0.96    logGBF = -73.951    itns = 1

 Parameters:
                a_    0.401835 +-    0.004           (     0.5 +-      0.5)
                 _    0.403323 +-   0.0049           (     0.5 +-      0.5)
                 _    0.446511 +-    0.051           (     0.5 +-      0.5)
                 _    0.600997 +-     0.21           (     0.5 +-      0.5)
                 _    0.380338 +-     0.37           (     0.5 +-      0.5)
                 _    0.395013 +-     0.46           (     0.5 +-      0.5)
                 _    0.450063 +-     0.49           (     0.5 +-      0.5)
                 _    0.479737 +-      0.5           (     0.5 +-      0.5)
                 _     0.49226 +-      0.5           (     0.5 +-      0.5)
                 _    0.497112 +-      0.5           (     0.5 +-      0.5)
                 _    0.498932 +-      0.5           (     0.5 +-      0.5)
                 _    0.499606 +-      0.5           (     0.5 +-      0.5)
                 _    0.499855 +-      0.5           (     0.5 +-      0.5)
```

```
    _    0.499947 +-      0.5           (     0.5 +-      0.5)
    _     0.49998 +-      0.5           (     0.5 +-      0.5)
    _    0.499993 +-      0.5           (     0.5 +-      0.5)
    _    0.499997 +-      0.5           (     0.5 +-      0.5)
    _    0.499999 +-      0.5           (     0.5 +-      0.5)
    _         0.5 +-      0.5           (     0.5 +-      0.5)
   E_    0.900345 +-  0.00051           (       1 +-      0.5)
    _     1.80105 +-   0.0019           (       2 +-      0.5)
    _     2.73354 +-    0.042           (       3 +-      0.5)
    _     3.91278 +-     0.33           (       4 +-      0.5)
    _     4.96687 +-     0.49           (       5 +-      0.5)
    _     6.02418 +-      0.5           (       6 +-      0.5)
    _     7.01928 +-      0.5           (       7 +-      0.5)
    _     8.00922 +-      0.5           (       8 +-      0.5)
    _     9.00374 +-      0.5           (       9 +-      0.5)
    _     10.0014 +-      0.5           (      10 +-      0.5)
    _     11.0005 +-      0.5           (      11 +-      0.5)
    _     12.0002 +-      0.5           (      12 +-      0.5)
    _     13.0001 +-      0.5           (      13 +-      0.5)
    _          14 +-      0.5           (      14 +-      0.5)
    _          15 +-      0.5           (      15 +-      0.5)
    _          16 +-      0.5           (      16 +-      0.5)
    _          17 +-      0.5           (      17 +-      0.5)
    _          18 +-      0.5           (      18 +-      0.5)
    _          19 +-      0.5           (      19 +-      0.5)

E1/E0 = 2.0004 +- 0.0018858   E2/E0 = 3.0361 +- 0.0466706
a1/a0 = 1.0037 +- 0.00663103   a2/a0 = 1.11118 +- 0.125291
```

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 parentheses, on the right). Values for each `a[i]` and `E[i]` are listed in order, starting at the points indicated.

  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.

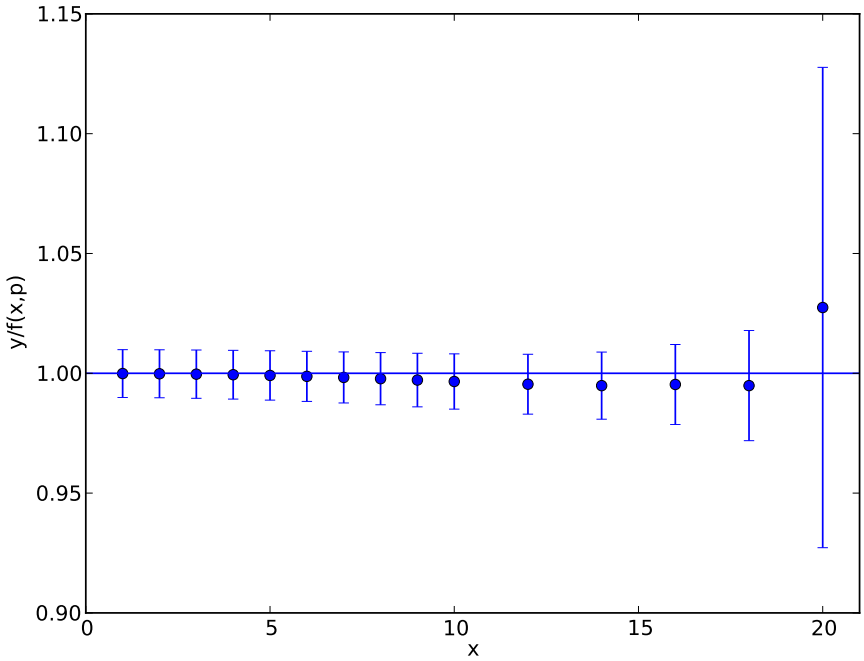
- The Gaussian Bayes Factor (or *posterior probability*, whose logarithm is `logGBF` in the output) is a measure of the likelihood that the actual data being fit could have come from a theory with the prior used in the fit. The larger this number, the more likely it is that prior and data could be related. Here it grows dramatically from the first fit (`nexp=3`) but then more-or-less stops changing around `nexp=6`. The implication is that this data is much more likely to have come from a theory with `nexp>=6` than with `nexp=3` (which we know to be the actual case).
- In the code, results for each fit are captured in a Python object `fit`, which is of type `lsqfit.nonlinear_fit`. A summary of the fit information is obtained by printing `fit`. Also the best-fit results for each fit parameter can be accessed through `fit.p`, as is done here to calculate various ratios of parameters.

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

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(15)`):

```
Fit:
          x_i          y_i       f(x_i)         dy_i
-----------------------------------------------------
            1      0.27518      0.27521   0.0027439
            2     0.079505     0.079521  0.00079613
            3     0.028911     0.028921  0.00029149
            4     0.011266     0.011272  0.00011468
            5    0.0045023    0.0045063  4.6409e-05
            6    0.0018171    0.0018194  1.9025e-05
            7  0.00073619   0.00073746  7.8556e-06
            8  0.00029873    0.0002994  3.2608e-06
            9  0.00012129   0.00012163    1.36e-06
           10  4.9257e-05   4.9426e-05  5.7008e-07
           12  8.1264e-06   8.1636e-06    1.02e-07
           14  1.3415e-06   1.3485e-06  1.8887e-08
           16  2.2171e-07   2.2275e-07  3.7159e-09
           18  3.6605e-08   3.6794e-08   8.455e-10
           20  6.2447e-09   6.0779e-09   6.092e-10
```

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 `ys`.



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

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

1.4 `x` has Error Bars

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

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

```
def make_data():                    # 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, 0.01)
    c = [gv.gvar(cr(), 0.01) 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)*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 = -69.734    itns = 6

Parameters:
              a_    0.402497 +-   0.0041          (     0.5 +-      0.5)
               _    0.428721 +-    0.032          (     0.5 +-      0.5)
               _    0.583018 +-     0.23          (     0.5 +-      0.5)
               _     0.40374 +-     0.38          (     0.5 +-      0.5)
               _    0.421848 +-     0.46          (     0.5 +-      0.5)
               _    0.463996 +-     0.49          (     0.5 +-      0.5)
              E_    0.900682 +-   0.0006          (       1 +-      0.5)
               _     1.81758 +-     0.02          (       2 +-      0.5)
               _      2.9487 +-     0.28          (       3 +-      0.5)
               _     3.97546 +-     0.49          (       4 +-      0.5)
```

```
                _      5.02085 +-      0.5           (       5 +-      0.5)
                _      6.01467 +-      0.5           (       6 +-      0.5)
               x_     0.999997 +-     1e-05          (       1 +-     1e-05)
                _      1.99996 +-     2e-05          (       2 +-     2e-05)
                _      3.00001 +-     3e-05          (       3 +-     3e-05)
                _      4.00006 +-   3.6e-05          (       4 +-     4e-05)
                _      5.00005 +-   3.4e-05          (       5 +-     5e-05)
                _      6.00002 +-   3.9e-05          (       6 +-     6e-05)
                _      6.99999 +-     4e-05          (       7 +-     7e-05)
                _      7.99996 +-   4.2e-05          (       8 +-     8e-05)
                _      8.99993 +-     5e-05          (       9 +-     9e-05)
                _      9.99992 +-   5.9e-05          (      10 +-    0.0001)
                _      11.9999 +-   7.9e-05          (      12 +-   0.00012)
                _      13.9999 +-   0.00011          (      14 +-   0.00014)
                _      15.9999 +-   0.00015          (      16 +-   0.00016)
                _      18.0002 +-   0.00018          (      18 +-   0.00018)
                _      20.0002 +-    0.0002          (      20 +-    0.0002)

E1/E0 = 2.01801 +- 0.0219085   E2/E0 = 3.27385 +- 0.307128
a1/a0 = 1.06515 +- 0.0772791   a2/a0 = 1.4485 +- 0.574717
```

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

1.5 Correlated Parameters; Gaussian Bayes Factor

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

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

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

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

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

For `nexp=3`, this implies that

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

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

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

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

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

```
************************************ nexp = 7
Least Square Fit:
  chi2/dof [dof] = 0.44 [15]    Q = 0.97    logGBF = -66.989    itns = 3

Parameters:
              a_    0.401798 +-     0.004          (     0.5 +-      0.5)
               _    0.401633 +-    0.0041          (     0.5 +-      0.5)
               _    0.403819 +-     0.012          (     0.5 +-      0.5)
               _    0.394153 +-     0.045          (     0.5 +-      0.5)
               _    0.398183 +-      0.15          (     0.5 +-      0.5)
               _    0.504394 +-      0.31          (     0.5 +-      0.5)
               _    0.515886 +-      0.42          (     0.5 +-      0.5)
              E_    0.900318 +-   0.00051          (       1 +-      0.5)
               _     1.80009 +-    0.0011          (     1.9 +-      0.5)
               _     2.70085 +-      0.01          (     2.8 +-      0.5)
               _      3.6008 +-     0.014          (     3.7 +-      0.5)
               _     4.50084 +-     0.017          (     4.6 +-      0.5)
               _     5.40084 +-      0.02          (     5.5 +-      0.5)
               _     6.30084 +-     0.022          (     6.4 +-      0.5)

E1/E0 = 1.9994 +- 0.0010494   E2/E0 = 2.99988 +- 0.0110833
a1/a0 = 0.999589 +- 0.00250023   a2/a0 = 1.00503 +- 0.0279927
```

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 = -67.0`) than it was for the uncorrelated prior (`logGBF = -73.9`). If one had 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 significantly more likely to get this data from a theory with the correlated prior than from one with the uncorrelated prior.) This difference is significant despite the fact that the `chi**2`s in the two cases are almost the same.

1.6 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 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 = -60.457    itns = 1

Parameters:
              a_    0.402651 +-    0.004          (     0.5 +-    0.095)
               _    0.402469 +-   0.0041          (     0.5 +-    0.095)
               _    0.407096 +-   0.0079          (     0.5 +-    0.095)
               _    0.385447 +-     0.02          (     0.5 +-    0.095)
               _    0.430817 +-    0.058          (     0.5 +-    0.095)
               _     0.47765 +-    0.074          (     0.5 +-    0.095)
               _    0.493185 +-    0.089          (     0.5 +-    0.095)
              E_    0.900307 +-   0.0005          (       1 +-      0.5)
               _     1.80002 +-    0.001          (     1.9 +-      0.5)
               _     2.70233 +-   0.0085          (     2.8 +-      0.5)
               _     3.60274 +-    0.013          (     3.7 +-      0.5)
               _      4.5033 +-    0.017          (     4.6 +-      0.5)
               _     5.40351 +-    0.019          (     5.5 +-      0.5)
               _     6.30355 +-    0.022          (     6.4 +-      0.5)

E1/E0 = 1.99934 +- 0.00100622   E2/E0 = 3.00156 +- 0.00926136
a1/a0 = 0.999549 +- 0.00245793   a2/a0 = 1.01104 +- 0.0165249
prior['a'] = 0.5 +- 0.0950546
```

Reducing the width of the `prior['a']`s from `0.5` to `0.1` increased `logGBF` from `-67.0` to `-60.5`. 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']`.

1.7 Partial Errors and Error Budgets

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

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

```
>>> E = fit.p['E']
>>> a = fit.p['a']
>>> print(E[1]/E[0])
1.9994 +- 0.0010494
>>> print((E[1]/E[0]).partialsdev(fit.prior['E']))
0.000414032342911
>>> print((E[1]/E[0]).partialsdev(fit.prior['a']))
0.000142408815921
>>> print((E[1]/E[0]).partialsdev(y))
0.000953694015457
```

This shows that the total uncertainty in `E[1]/E[0]`, `0.00105`, is the sum in quadrature of a contribution `0.00041` due to the priors specified by `prior['E']`, `0.00014` 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: 1.999(1)
               a2/a0: 1.005(28)
               a1/a0: 1.000(3)

Partial % Errors:
                          E2/E0     E1/E0     a2/a0     a1/a0
-----------------------------------------------------------
                  a:       0.09      0.01      1.07      0.02
                  y:       0.07      0.05      0.78      0.19
                  E:       0.35      0.02      2.45      0.16
```

```
-------------------------------------------------------
               total:      0.37      0.05      2.79      0.25
```

This table suggests, for example, that reducing the statistical errors in the input `y` data would significantly reduce the final errors in `E1/E0` and `a1/a0`, but would have only a slight impact on errors in `E2/E0` and `a2/a0`. 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 0.36%.

1.8 `y` has No Error Bars

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

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

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

We know that for some set of parameters

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

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

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

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

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

```
def main():
    gv.ranseed([2009, 2010, 2011, 2012])  # initialize random numbers (opt.)
    max_prior = make_prior(20)          # maximum sized prior
    p0 = None                           # make larger fits go faster (opt.)
    for nexp in range(1, 7):
        print('************************************ nexp =', nexp)
        fit_prior = gv.BufferDict()     # part of max_pior used in fit
        ymod_prior = gv.BufferDict()    # part of max_prior absorbed in ymod
```

```
        for k in max_prior:
            fit_prior[k] = max_prior[k][:nexp]
            ymod_prior[k] = max_prior[k][nexp:]
        x, y = make_data(ymod_prior)    # make fit data
        fit = lsqfit.nonlinear_fit(data=(x, y), fcn=f, prior=fit_prior, p0=p0)
        print(fit.format(10))           # print the fit results
        print()
        if fit.chi2/fit.dof<1.:
            p0 = fit.pmean              # starting point for next fit (opt.)
```

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

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

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

```
************************************ nexp = 1
Least Square Fit (y correlated with prior):
  chi2/dof [dof] = 0.056 [10]    Q = 1    logGBF = -16.24    itns = 5

Parameters:
              a_    0.400845 +-  0.00094           (     0.5 +-      0.5)
              E_    0.900324 +-   0.0004           (       1 +-      0.5)

Fit:
         x_i          y_i       f(x_i)         dy_i
-------------------------------------------------
           1      0.14803      0.16292      0.10692
         1.2      0.12825      0.13607     0.074202
         1.4      0.10957      0.11365     0.051975
         1.6     0.092853     0.094922     0.036625
         1.8     0.078298      0.07928      0.02591
           2     0.065813     0.066216     0.018378
         2.2       0.0552     0.055305     0.013057
         2.4     0.046231     0.046191    0.0092867
         2.6      0.03868      0.03858    0.0066089
         2.8     0.032339     0.032223    0.0047043


************************************ nexp = 2
Least Square Fit (y correlated with prior):
  chi2/dof [dof] = 0.056 [10]    Q = 1    logGBF = -35.133    itns = 4

Parameters:
              a_    0.399968 +-  0.00079           (     0.5 +-      0.5)
               _    0.400415 +-    0.026           (     0.5 +-      0.5)
              E_    0.899986 +-  0.00031           (       1 +-      0.5)
               _     1.79983 +-     0.02           (     1.9 +-     0.54)

Fit:
         x_i          y_i       f(x_i)         dy_i
-------------------------------------------------
           1      0.22281      0.22882     0.044661
```

```
        1.2     0.17939     0.18202    0.025977
        1.4     0.14454     0.14568    0.015244
        1.6     0.11677     0.11725    0.008997
        1.8    0.094655    0.094842   0.0053294
          2    0.076998    0.077061   0.0031644
        2.2    0.062849    0.062861   0.0018817
        2.4    0.051462    0.051455   0.0011199
        2.6    0.042257    0.042246  0.00066679
        2.8    0.034786    0.034776  0.00039704


************************************ nexp = 3
Least Square Fit (y correlated with prior):
  chi2/dof [dof] = 0.058 [10]    Q = 1    logGBF = -50.219    itns = 4

Parameters:
             a_    0.399938 +-  0.00082          (     0.5 +-      0.5)
              _    0.398106 +-    0.034          (     0.5 +-      0.5)
              _    0.401049 +-    0.098          (     0.5 +-      0.5)
             E_    0.899975 +-  0.00032          (       1 +-      0.5)
              _     1.79848 +-    0.024          (     1.9 +-     0.54)
              _     2.69343 +-      0.2          (     2.8 +-     0.57)

Fit:
         x_i          y_i      f(x_i)        dy_i
-----------------------------------------------------
           1     0.25322     0.25564     0.01863
         1.2     0.19676     0.19765   0.0090783
         1.4     0.15446     0.15478   0.0044619
         1.6     0.12244     0.12255   0.0022047
         1.8    0.097892     0.09793   0.0010932
           2    0.078847    0.078859  0.00054319
         2.2    0.063905    0.063908  0.00027026
         2.4    0.052065    0.052065  0.00013456
         2.6    0.042602    0.042601   6.701e-05
         2.8    0.034983    0.034982   3.337e-05


************************************ nexp = 4
Least Square Fit (input data correlated with prior):
  chi2/dof [dof] = 0.057 [10]    Q = 1    logGBF = -67.447    itns = 5

Parameters:
             a_    0.399937 +-  0.00077          (     0.5 +-      0.5)
              _    0.398315 +-    0.032          (     0.5 +-      0.5)
              _    0.401742 +-      0.1          (     0.5 +-      0.5)
              _    0.403269 +-     0.15          (     0.5 +-      0.5)
             E_    0.899975 +-   0.0003          (       1 +-      0.5)
              _     1.79859 +-    0.023          (     1.9 +-     0.54)
              _     2.69522 +-     0.19          (     2.8 +-     0.57)
              _     3.60827 +-     0.28          (     3.7 +-     0.61)

Fit:
         x_i          y_i      f(x_i)        dy_i
-----------------------------------------------------
           1     0.26558      0.2666   0.0077614
         1.2     0.20266     0.20297   0.0031677
         1.4     0.15728     0.15737   0.0013035
```

```
1.6     0.12378     0.12381  0.00053913
1.8    0.098532     0.09854  0.00022369
  2    0.079153    0.079155  9.2995e-05
2.2    0.064051    0.064051  3.8703e-05
2.4    0.052134    0.052134  1.6117e-05
2.6    0.042635    0.042635   6.712e-06
2.8    0.034999    0.034998  2.7948e-06
```

Here we use `fit.format(10)` to print out a table of `x` and `y` (actually `ymod`) values, together with the value of the fit function using the best-fit parameters. There are several things to notice:

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

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

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

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

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

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

1.9 SVD Cuts and Roundoff Error

We did not display values for `E1/E0`, `a1/a0` ... in the example in the previous section. Had we done so a problem would have been immediately apparent: for example,

```
************************************ nexp = 4
Least Square Fit (input data correlated with prior):
  chi2/dof [dof] = 0.057 [10]    Q = 1    logGBF = -67.447    itns = 5

Parameters:
             a_    0.399937 +-  0.00077          (     0.5 +-      0.5)
              _    0.398315 +-    0.032          (     0.5 +-      0.5)
              _    0.401742 +-      0.1          (     0.5 +-      0.5)
              _    0.403269 +-     0.15          (     0.5 +-      0.5)
             E_    0.899975 +-   0.0003          (       1 +-      0.5)
              _     1.79859 +-    0.023          (     1.9 +-     0.54)
              _     2.69522 +-     0.19          (     2.8 +-     0.57)
              _     3.60827 +-     0.28          (     3.7 +-     0.61)
```

```
Fit:
         x_i          y_i       f(x_i)         dy_i
-----------------------------------------------------
           1      0.26558       0.2666    0.0077614
         1.2      0.20266      0.20297    0.0031677
         1.4      0.15728      0.15737    0.0013035
         1.6      0.12378      0.12381   0.00053913
         1.8     0.098532      0.09854   0.00022369
           2     0.079153     0.079155   9.2995e-05
         2.2     0.064051     0.064051   3.8703e-05
         2.4     0.052134     0.052134   1.6117e-05
         2.6     0.042635     0.042635    6.712e-06
         2.8     0.034999     0.034998   2.7948e-06

E1/E0 = 1.99849 +- 0.154988   E2/E0 = 2.99477 +- 1.65242
a1/a0 = 0.995944 +- 0.514388   a2/a0 = 1.00451 +- 2.32754
```

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

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

The standard solution to this common problem in least-squares fitting is to introduce an *svd* cut, here called `svdcut`:

```
fit = nonlinear_fit(data=(x, ymod), fcn=f, prior=prior, p0=p0, svdcut=1e-12)
```

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

Rerunning our fit with `svdcut=1e-12` we obtain

```
********************************** nexp = 4
Least Square Fit (input data correlated with prior):
  chi2/dof [dof] = 0.053 [10]    Q = 1    logGBF = -55.494    itns = 3

Parameters:
              a_    0.400162 +-   0.0013          (     0.5 +-      0.5)
               _    0.404161 +-    0.039          (     0.5 +-      0.5)
               _    0.404572 +-     0.11          (     0.5 +-      0.5)
               _    0.408034 +-     0.16          (     0.5 +-      0.5)
              E_    0.900066 +-  0.00052          (       1 +-      0.5)
               _     1.80348 +-    0.031          (     1.9 +-     0.54)
               _     2.71749 +-     0.21          (     2.8 +-     0.57)
               _     3.62392 +-     0.29          (     3.7 +-     0.61)

Fit:
```

```
         x_i          y_i       f(x_i)         dy_i
-----------------------------------------------------
           1      0.26558      0.26686    0.0077614
         1.2      0.20266      0.20309    0.0031677
         1.4      0.15728      0.15742    0.0013035
         1.6      0.12378      0.12383   0.00053913
         1.8     0.098532      0.09855   0.00022369
           2     0.079153     0.079159   9.2995e-05
         2.2     0.064051     0.064053   3.8703e-05
         2.4     0.052134     0.052135   1.6117e-05
         2.6     0.042635     0.042635    6.712e-06
         2.8     0.034999     0.034999   2.7948e-06

E1/E0 = 2.00372 +- 0.0330005   E2/E0 = 3.01921 +- 0.234244
a1/a0 = 1.00999 +- 0.0955902   a2/a0 = 1.01102 +- 0.269968
```

and consistency has been restored. Note that taking `svdcut=-1e-12` (with a minus sign) causes the problematic modes to be dropped. This is a more conventional implementation of *svd* cuts, but here it results in much less precision than using `svdcut=1e-12` (for example, `2.01972 +- 0.115874` for `E1/E0`, which is almost four times less precise). Dropping modes is equivalent to setting the corresponding variances equal to infinity, which is (obviously) much more conservative and less realistic than setting them equal to the *svd*-cutoff variance.

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

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

which gives:

```
Values:
               E2/E0: 3.019(234)
               E1/E0: 2.004(33)
               a2/a0: 1.011(270)
               a1/a0: 1.010(96)

Partial % Errors:
                         E2/E0     E1/E0     a2/a0     a1/a0
-------------------------------------------------------------
                   a:     2.53      0.66     10.71      3.47
                 svd:     1.30      0.49      1.81      2.46
                   E:     7.22      1.43     24.39      8.45
-------------------------------------------------------------
               total:     7.76      1.65     26.70      9.46
```

Here the contribution from the *svd* cut is rather modest.

The method `lsqfit.nonlinear_fit.check_roundoff()` can be used to check for roundoff errors. It generates a warning if roundoff looks to be a problem.

1.10 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])
# extract means and standard deviations from the bootstrap output
from numpy import mean, std
for k in outputs:
    outputs[k] = gv.gvar(np.mean(outputs[k]), np.std(outputs[k]))
print('Bootstrap results:')
print('E1/E0 =', outputs['E1/E0'], '  E2/E1 =', outputs['E2/E0'])
print('a1/a0 =', outputs['a1/a0'], '  a2/a0 =', outputs['a2/a0'])
```

The results are consistent with the results obtained directly from the fit (when using `svdcut=1e-12`):

```
Bootstrap results:
E1/E0 = 2.00618 +- 0.027411   E2/E1 = 3.05219 +- 0.195792
a1/a0 = 1.01777 +- 0.0755551   a2/a0 = 1.06962 +- 0.275993
```

In particular, the bootstrap analysis confirms our previous error estimates (to within 10-20%, since `Nbs=40`). When `Nbs` is small, it is often safer to use the median instead of the mean as the estimator (this can be done by changing the line `outputs[k] = gv.gvar(np.mean(...` above to `outputs[k] = gvar.dataset.avg_data(outputs[k], bstrap=True)`).

1.11 Troubleshooting

`lsqfit.nonlinear_fit` sometimes gives unintelligible error messages such as:

```
Traceback (most recent call last):
  File "<stdin>", line 10, in <module>
    fit = nonlinear_fit(data=(None, y), prior=prior, fcn=f)
  File "/Users/gpl/Library/Python/2.7/lib/python/site-packages/lsqfit/__init__.py", line 240, in __ir
    fit = multifit(p0, nf, self._chiv, **self.fitterargs)
```

```
  File "_utilities.pyx", line 303, in lsqfit._utilities.multifit.__init__ (src/lsqfit/_utilities.c:26
RuntimeError: Python error in fit function: 33
```

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

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

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

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

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

```
>>> print(prior['a'], prior['b'])
1 +- 1 1 +- 1
>>> print(prior['a']-prior['b'])
0 +- 0
```

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

CHAPTER

TWO

GVAR - GAUSSIAN RANDOM VARIABLES

2.1 Introduction

This module provides tools for representing and manipulating gaussian random variables numerically. 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, and not, for example, a *particular* random number drawn from that 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. This calculus is useful for analyzing the propagation of statistical and other random errors (provided the standard deviations are small enough).

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, in general, `<y>` is the expectation value or mean of random variable `y`.

2.2 Creating Gaussian Variables

An object of type `gvar.GVar` represents a single gaussian variable. Such an object can be created for a single variable, with mean `xmean` and standard deviation `xsdev` (both scalars), 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.GVar`s: for example,

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

A `gvar.GVar` can be converted to a string of this last format using the `GVar.fmt()` method: for example,

```
>>> print(x.fmt(4))
3.1415(2)
>>> print(x.fmt(5))
3.14150(20)
```

where the argument is the number of decimal places retained.

Function `gvar.asgvar(x)` returns x if it is a `gvar.GVar`; otherwise it returns `gvar(x)`.

`gvar.GVar`s are far more interesting when used to describe multidimensional distributions, especially if there are correlations between different variables. Such distributions are represented by collections of `gvar.GVar`s in one of two standard formats: 1) `numpy` type arrays of `gvar.GVar`s (any shape); or, more flexibly, 2) Python dictionaries whose values are `gvar.GVar`s or arrays of `gvar.GVar`s. Most functions in `gvar` that handle multiple `gvar.GVar`s 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.BufferDict`s. `gvar.BufferDict`s are also useful for archiving `gvar.GVar`s, since they may be pickled using Python's `pickle` module; `gvar.GVar`s cannot be pickled otherwise. A pickled `gvar.BufferDict` preserves all of the correlations between the different `gvar.GVar`s in it.

To create an array of `gvar.GVar`s 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]
```

`gvar.GVar`s can be used in arithmetic expressions, just like Python floats. These expressions result in new `gvar.GVar`s whose means and standard deviations are determined from the original covariance matrix. The arithmetic expressions can include calls to standard functions including: `exp`, `log`, `sqrt`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, `arctanh`.

As an example,

```
>>> x, y = gvar.gvar([0.1, 10.], [[0.015625, 0.], [0., 4.]])
>>> print('x =', x, '   y =', y)
x = 0.1 +- 0.125    y = 10 +- 2
```

makes `x` and `y` `gvar.GVar`s with standard deviations `sigma_x=0.125` and `sigma_y=2`, and, in this case, no correlation between `x` and `y` (since `cov[i, j]=0` when `i!=j`). If now we set, for example,

```
>>> f = x+y
>>> print('f =', f)
f = 10.1 +- 2.0039
```

then `f` is a `gvar.GVar` with

```
f.var = df/dx cov[0, 0] df/dx + 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)
1.01 +- 0.012659
```

This happens, of course, because the errors in `f` and `y` are highly correlated (since the error in `f` comes mostly from `y`).

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

```
>>> 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
```

`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 +- 1 10 +- 5 99 +- 3 0 +- 2]
>>> gdict = gvar.gvar(dict(a="2(1)", b=["10+-5", (99, 3), gvar.gvar(0, 2)]))
>>> print(gdict)
{'a': 2 +- 1, 'b': array([10 +- 5, 99 +- 3, 0 +- 2], 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, therefore, is equivalent to:

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

2.3 Computing Covariance Matrices

The covariance matrix for a set of `gvar.GVar`s, `g0 g1 ...`, can be computed using

```
gvar.evalcov([g0, g1...]) -> cov_g
```

where `cov_g[i, j]` gives the covariance between `gi` and `gj`. Instead of a list or array of `g`s, one can also give a dictionary `g` where `g[k]` is a `gvar.GVar`. In this case `gvar.evalcov()` returns a doubly-indexed dictionary `cov_g[k1][k2]` where keys `k1, k2` are in `g`.

Using the example from the previous section, the code

```
>>> x, y = gvar.gvar([0.1, 10.], [[0.015625, 0.], [0., 4.]])
>>> f = x+y
>>> print(gvar.evalcov([x, y, f]))
[[ 0.015625  0.        0.015625]
 [ 0.        4.        4.      ]
 [ 0.015625  4.        4.015625]]
```

confirms that `x` and `y` are uncorrelated with each other, but strongly correlated with `f`.

It is often convenient to group related `gvar.GVar`s together in a dictionary rather than an array since dictionaries are far more flexible. `gvar.evalcov` can be used to evaluate the covariance matrix for a dictionary containing `gvar.GVar`s and/or arbitrary arrays of `gvar.GVar`s:

```
>>> d = dict(x=x, y=y, g=[x+y, x-y])
>>> cov = gvar.evalcov(d)
>>> print(cov['x', 'x'])
0.015625
>>> print(cov['x', 'y'])
0.0
>>> print(cov['x', 'g'])
[ 0.015625  0.015625]
```

2.4 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(f)
>>> 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 +- 1, 'b': [0 +- 100, 10 +- 0.001]}
>>> 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))
0.540302 +- 2.52441
```

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)
[1.1 +- 0.1 0.8 +- 0.1]
>>> 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)
[1.13881 +- 0.1 0.896066 +- 0.1]            # 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.

2.5 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
0.0001 +- 0.0001
```

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
1e-8 +- 0
```

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).

2.6 Implementation Notes; Optimizations

There are two types of `gvar.GVar`: the underlying independent variables, created with calls to `gvar.gvar()`; and variables which are obtained from functions of the underlying variables. Each `gvar.GVar` must keep track of three pieces of information: 1) its mean value; 2) its derivatives with respect to the underlying variables; and 3) the covariance matrix for the underlying variables. 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 underlying variables. A `gvar.GVar` can be constructed at a very low level by supplying all three 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 underlying variable (numbered in the order in which they were created using `gvar.gvar()`), and `cov` is the covariance matrix for the underlying variables (easily obtained from an existing `gvar.GVar` `x` using `x.cov`).

When there are lots of underlying variables, the number of derivatives 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 underlying 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, but in cases with tens of thousands of underlying variables redefining `gvar` might make a difference. Note that the previous `gvar.gvar()` can be restored using `gvar.restore_gvar()`.

`gvar.GVar`s are designed to work well with `numpy` arrays. They can be combined in arithmetic expressions with arrays of numbers or of `gvar.GVar`s; the results in both cases are arrays of `gvar.GVar`s.

Arithmetic operators `+ - * / ** == != <> += -= *= /=` are all defined. `gvar.GVar`s are not ordered so `> >= < <=` are not defined.

2.7 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`.

`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 = x.shape+x.shape`. The result has the same shape as `x`.

`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`**`(`*dict(x=xstr, y=(x, xsdev)...)*`)`
Returns a dictionary (`BufferDict`) `b` where `b['x'] = gvar(xstr)`, `b['y']=gvar(x,xsdev)` ...

`gvar.`**`gvar`**`(`*[[(x1,x1sdev)...]...]*`)`
Returns an array `numpy.array([[gvar(x1,x1sdev)...]...])`. Works for arrays of any shape.

gvar.**gvar**(*[[x1str...]...]*)
Returns an array `numpy.array([[gvar(xstr1)...]...])` where `x1str...` are strings. Works for arrays of any shape.

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

Means, standard deviations, variances, and covariance matrices 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`.

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

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

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*, *svdnum=None*, *rescale=True*)
Return iterator for random samples from distribution `g`

The gaussian deviates (`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.

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 of the covariance matrix of `g` with `svdcut*(max eigenvalue)`; if negative, discards eigenmodes with eigenvalues smaller than `svdcut*(max eigenvalue)`; ignore if set to `None`.
- **svdnum** (`None` or positive `int`) – If positive, keep only the modes with the largest `svdnum` eigenvalues in the covariance matrix for `g`; ignore if set to `None` or negative.
- **rescale** (*bool*) – Covariance matrix is rescaled so that diagonal elements equal `1` if `rescale=True`.

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*, *svdnum=None*, *rescale=True*)
Return iterator for bootstrap copies of `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 of the covariance matrix of `g` with `svdcut*(max eigenvalue)`; if negative, discards eigenmodes with eigenvalues smaller than `svdcut*(max eigenvalue)`; ignore if set to `None`.
- **svdnum** (`None` or positive `int`) – If positive, keep only the modes with the largest `svdnum` eigenvalues in the covariance matrix for `g`; ignore if set to `None` or negative.
- **rescale** (*bool*) – Covariance matrix is rescaled so that diagonal elements equal `1` if `rescale=True`.

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.

Parameters **seed** (*tuple*) – A tuple of integers.

Two functions that are useful for tabulating results and for analyzing where the errors in a `gvar.GVar` constructed from other `gvar.GVar`s come from:

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

Parameters

- **outputs** – Dictionary of `gvar.GVar`s for which partial standard deviations are computed.
- **inputs** – Dictionary of lists of: `gvar.GVar`s or arrays/dictionaries of `gvar.GVar`s. The partial standard deviation due to the input quantities in each list is tabulated for each output quantity in `outputs`.
- **ndigit** (`int`) – Number of decimal places displayed in table.
- **percent** (*boolean*) – Tabulate % errors if `percent is True`; otherwise tabulate the errors themselves.

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*, *ndigit=3*)
Tabulate `gvar.GVar`s in `outputs`.

Parameters

- **outputs** – A dictionary of `gvar.GVar` objects.
- **ndigit** (`int` or `None`) – Number of digits displayed in table; if `None`, use `str(outputs[k])` instead.

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

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).

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.

Finally there is a utility function and a class for implementing an *svd* analysis of a covariance or other symmetric, positive matrix:

`gvar.`**`svd`**`(`*g*, *svdcut=None*, *svdnum=None*, *compute_delta=False*, *rescale=False*`)`
Apply svd cuts to collection of `gvar.GVar`s in `g`.

`g` is an array of `gvar.GVar`s or a dictionary containing `gvar.GVar`s and/or arrays of `gvar.GVar`s. `svd(g,...)` returns a copy of `g` whose `gvar.GVar`s have been modified so that their covariance matrix is less singular than for the original `g` (the `gvar.GVar` means are unchanged). This is done using an *svd* algorithm which is controlled by three parameters: `svdcut`, `svdnum` and `rescale` (see `gvar.SVD` for more

details). *svd* cuts are not applied when the covariance matrix is diagonal (that is, when there are no correlations between different elements of `g`).

The input parameters are :

Parameters

- **g** – An array of `gvar.GVar`s or a dicitionary whose values are `gvar.GVar`s and/or arrays of `gvar.GVar`s.
- **svdcut** (`None` or number `(|svdcut|<=1).`) – If positive, replace eigenvalues of the covariance matrix with `svdcut*(max eigenvalue)`; if negative, discard eigenmodes with eigenvalues smaller than `svdcut` times the maximum eigenvalue. Default is `None`.
- **svdnum** (`None` or int) – If positive, keep only the modes with the largest `svdnum` eigenvalues; ignore if set to `None`. Default is `None`.
- **rescale** – Rescale the input matrix to make its diagonal elements equal to 1.0 before applying *svd* cuts.
- **compute_inv** – Compute representation of inverse of covariance matrix if `True`; the result is stored in `svd.inv_wgt` (see below). Default value is `False`.

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

- `svd.val`: eigenvalues of the covariance matrix after *svd* cuts (and after rescaling if `rescale=True`); the eigenvalues are ordered, with the smallest first;
- `svd.vec`: eigenvectors of the covariance matrix after *svd* cuts (and after rescaling if `rescale=True`), where `svd.vec[i]` is the vector corresponding to `svd.val[i]`;
- `svd.eigen_range`: ratio of the smallest to largest eigenvalue before *svd* cuts are applied (but after rescaling if `rescale=True`);
- `svd.D`: diagonal of matrix used to rescale the covariance matrix before applying *svd* cuts (cuts are applied to `D cov D`) if `rescale=True`; `svd.D` is `None` if `rescale=False`;
- `svd.logdet`: logarithm of the determinant of the covariance matrix after *svd* cuts are applied;
- `svd.correction`: vector of the *svd* corrections to `g.flat`;
- `svd.inv_wgt`: the sum of the outer product of vectors `inv_wgt[i]` with themselves equals the inverse of the covariance matrix after *svd* cuts. Only computed if `compute_inv=True`. The order of the vectors is reversed relative to `svd.val` and `svd.vec`

2.8 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.GVars:

partialvar(**args*)
Compute partial variance due to gvar.GVars in args.

This method computes the part of self.var due to the gvar.GVars in args. If args[i] is derived from other gvar.GVars, the variance coming from these is included in the result.

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

Returns Partial variance due to all of args.

partialsdev(**args*)
Compute partial standard deviation due to gvar.GVars in args.

This method computes the part of self.sdev due to the gvar.GVars in args. If args[i] is derived from other gvar.GVars, the standard deviation coming from these is included in the result.

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

Returns Partial standard deviation due to args.

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

__str__()
Convert to string with format: mean +- std-dev.

fmt(*d=4*, *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 d 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 d is None, it is set automatically to the larger of int(1-log(self.sdev)/log(10)) or 0; this will display the smallest number of digits needed to expose the error.

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

cov

der
Array of derivatives with respect to underlying (original) gvar.GVars.

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.GVars (or other scalars) and arrays of gvar.GVars (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.flat = buf`:

```
>>> a.flat = buf
>>> print(a['tensor'])
[[ 30.  40.]
 [ 50.  60.]]
```

`a.flat` is an iterator for the `numpy` array used for `a`'s buffer. It can be used to access and change the buffer directly. `a.flatten()` is a copy of the buffer.

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

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 `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`.

add(*k*, *v=None*)
Augment buffer with data `v`, indexed by key `k`.

`v` is either a scalar or a `numpy` array (or a list or other data type that can be changed into a numpy.array). If `v` is a `numpy` array, it can have any shape.

Same as `self[k] = v` except: 1) when `v is None`, in which case k is assumed to be a dictionary and each entry in it is added; and 2) when `k` is already used in `self`, in which case a `ValueError` is raised.

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.

kappa
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.

2.9 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+-1, 2+-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))
1.3095 +- 0.452117
```

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))
[0.946667 +- 0.217418 1.66817 +- 0.251002]
```

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([0.946667 +- 0.217418, 1.66817 +- 0.251002], dtype=object),
 'n': 1.3095 +- 0.452117}
```

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']))
1.3095 +- 0.452117
```

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*, *median=False*, *spread=False*, *bstrap=False*)

Average random data to estimate mean.

`data` is a list of random numbers or random arrays, or a dictionary of lists of random numbers/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 +- 0.203169
>>> print(avg_data(random_arrays))
[11.3333 +- 1.13521 107.95 +- 12.936]
>>> print(avg_data(random_dict))
{'n': 1.31 +- 0.203169, 'a': array([11.3333 +- 1.13521, 107.95 +- 12.936], dtype=object)}
```

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 three 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 +- 0.497661
>>> print(avg_data(random_numbers))
1.31 +- 0.203169
>>> >>> print((0.497661/0.203169)**2)    # should be 6
6.00001491255
```

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`.

`gvar.dataset.`**`autocorr`**(*data*, *ncorr=None*)
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,ncorr)` returns an array of length `ncorr` where `autocorr(data,ncorr)[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. Correlations are computed only for `i < ncorr` unless `ncorr` is `None`, in which case all possible separations are analyzed.

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

The computational cost of running `autocorr` grows linearly with `ncorr` and, obviously, with the size of `data`.

`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 numbers/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]
```

Finally 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 the 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])
[]
```

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 (or array) to `data['n']` (or `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 (or arrays) to `data['n']` (or `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

LSQFIT - NONLINEAR LEAST SQUARES FITTING

4.1 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.

4.2 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

$$\begin{aligned}\chi^2 &\equiv \sum_{ij} \Delta y(p)_i \, (\mathrm{cov}_y^{-1})_{ij} \, \Delta y(p)_j \\ &\equiv \Delta y(p) \cdot \mathrm{cov}_y^{-1} \cdot \Delta y(p)\end{aligned}$$

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)

$$\sigma_g^2 = \partial g(\overline{p}) \cdot \mathrm{cov}_p \cdot \partial g(\overline{p})$$

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

$$\frac{\partial p_a}{\partial y_i} = D_{ai}.$$

This relationship is true in the limit of small errors, as is easily derived from the minimum condition for the fit: Differentiating with respect to y_i we obtain

$$(\partial_a \Delta y(p))^{\mathrm{T}} \cdot \mathrm{cov}_y^{-1} \cdot \frac{\partial \Delta y(p)}{\partial y_i} = 0$$

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 sometimes leads to more accurate results.) Note that the covariance matrix has at most N non-zero eigenvalues when it is estimated from N random samples; zero-modes should always be discarded.

4.3 nonlinear_fit Objects

class `lsqfit.`**`nonlinear_fit`**(*data=None*, *fcn=None*, *prior=None*, *p0=None*, *svdcut=None*, *svdnum=None*, *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 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.

Parameters

- **data** (2-tuple or 3-tuple or `y`) – Fit data consisting of `(x, y)` where `x` is the independent data that is passed to the fit function, and `y` is a dictionary whose values are `gvar.GVar`s or arrays of `gvar.GVar`s specifying the means and covariance matrix for the dependent data (*i.e.*, the data being fit). `y` could instead be an array of `gvar.GVar`s, rather than a dictionary. Another format for `data` is the 3-tuples `(x, ymean, ycov)` (or `(x, ymean, ysdev)`) where `ymean` is an array containing the mean `y` values, and `ycov` is the corresponding covariance matrix (or `ysdev` the corresponding array of standard deviations, if there are no correlations). In this second case, `ycov.shape` must equal `ymean.shape+ymean.shape`. A final option is `data=y` in which case the fit function is assumed to be independent of `x`.
- **fcn** (*function*) – Fit function `fcn(x, p)` of the independent data `x` and the parameters `p`. The function should return approximations to the `y` data in the same format used for `y` in `data=(x, y)` (*i.e.*, a dictionary or array). Fit parameters are stored in `p`, which is either a dictionary, where `p[k]` is a single parameter or an array of parameters (any shape), or an array of parameters. When `data=y` or `x=False`, the fit function should depend only upon the parameters: `fcn(p)`.
- **prior** (dictionary, array, or `None`) – A dictionary (or array) containing *a priori* estimates for all parameters `p` used by fit function `fcn(x, 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. p0 should be a dictionary with the same keys and structure as `prior` (or an array of the same shape if `prior` is an array). 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 the prior. The default value is `None`; `p0` must be defined if `prior` is `None`.
- **svdcut** (`None` or `float` or 2-tuple) – If positive, eigenvalues of the (rescaled) `y` covariance matrix that are smaller than `svdcut` times the maximum eigenvalue are replaced by `svdcut` times the maximum eigenvalue. If negative, eigenmodes with eigenvalues smaller than `|svdcut|` times the largest eigenvalue are discarded. If zero or `None`, the covariance matrix is left unchanged. If `svdcut` is a 2-tuple, the first entry is `svdcut` for the `y` covariance matrix and the second entry is `svdcut` for the prior's covariance matrix.
- **svdnum** (`None` or `int` or 2-tuple) – If positive, at most `svdnum` eigenmodes of the (rescaled) `y` covariance matrix are retained; the modes with the smallest eigenvalues are discarded. `svdnum` is ignored if set to `None`. If `svdnum` is a 2-tuple, the first entry is `svdnum` for the `y` covariance matrix and the second entry is `svdnum` for the prior's covariance matrix.
- **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 `chi**2` for the last fit.

cov
Covariance matrix from fit.

dof
Number of degrees of freedom in fit.

logGBF
Logarithm of Gaussian Bayes Factor for last fit (larger is better). The exponential of `fit.logGBF` is proportional to the Bayesian posterior probability of the fit in a linearized (that is, Gaussian) approximation. Larger values imply larger posterior probabilities.

p
Best-fit parameters from last 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.

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

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

palt
Same as `fit.p` except that the errors are computed directly from `fit.cov`. This is faster but means that no information about correlations with the input data is retained (unlike in `fit.p`); and, therefore,

`fit.palt` cannot be used to generate error budgets. `fit.p` and `fit.palt` give the same means and normally give the same errors for each parameter. They differ only when the input data's covariance matrix is too singular to invert accurately (because of roundoff error), in which case an *svd* cut is advisable.

p0
The parameter values used to start the fit.

Q
Quality factor for last fit (should be >0.1 for good fits).

svdcorrection
A list where `svdcorrection[0]` is the *svd* correction to the fit data: that is, `fit.y` is obtained by adding `svdcorrection[0]` to the (flattened) input data. Similarly, `svdcorrection[1]` is the correction for the prior: `fit.prior` is obtained by adding `svdcorrection[1]` to the (flattened) input prior. Each `svdcorrection[i]` is either a flattened array, or `None`, if there was no *svd* correction. `svdcorrection` is mostly used as an input variable when creating error budgets, to assess how much of the error in a final result is caused by the *svd* cut.

time
CPU time (in secs) taken by last fit.

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

prior
Prior used in the fit. This may be differ from the input prior if an *svd* cut is used (`svdcut>0`). 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 be differ from the input data if an *svd* cut is used (`svdcut>0`). 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, 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*)
Formats fit output details into a string for printing.

The best-fit values for the fitting function are tabulated together with the input data if argument `maxline>0`.

The format of the output is controlled by the following format strings:

- `nonlinear_fit.fmt_label` - parameter labels
- `nonlinear_fit.fmt_parameter` - parameters
- `nonlinear_fit.fmt_prior` - priors
- `nonlinear_fit.fmt_table_header` - header for data vs fit
- `nonlinear_fit.fmt_table_line` - line in data vs fit
- `nonlinear_fit.alt_fmt_table_line` - alt line in data vs fit

Parameters **maxline** (*integer*) – 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. Default is `maxline=0`.

Returns String containing detailed information about last fit.

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

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

Parameters

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

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*, *ndigit=3*)

Tabulate `gvar.GVar`s in `outputs`.

Parameters

- **outputs** – A dictionary of `gvar.GVar` objects.
- **ndigit** (`int` or `None`) – Number of digits displayed in table; if `None`, use `str(outputs[k])` instead.

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

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 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.

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).

Typical usage is:

```
...
fit = lsqfit.nonlinear_fit(...)
...
for bsfit in fit.bootstrap_iter(n=100, datalist=datalist):
    ... analyze fit parameters in bsfit.pmean ...
```

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 an `AssertionError` if they do not (in which case an *svd* cut might be advisable).

4.4 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**(*xa*, *svdcut=None*, *svdnum=None*, *rescale=True*, *covfac=None*)

Weighted average of 1d-sequence of gvar.GVars or arrays 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 p[0]`. 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.

Typical usage is:

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

In this example, `x1` and `x2` could be replaced by arrays of gvar.GVars, in which case `xavg` is an array as well: for example,

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

Parameters

- **xa** – The gvar.GVars to be averaged. `xa` is a one-dimensional sequence of gvar.GVars or of arrays of gvar.GVars, all of the same shape.
- **svdcut** (`None` or `float`) – If positive, eigenvalues of the `xa` covariance matrix that are smaller than `svdcut` times the maximum eigenvalue are replaced by `svdcut` times the maximum eigenvalue. If negative, eigenmodes with eigenvalues smaller than `|svdcut|` times the largest eigenvalue are discarded. If zero or `None`, the covariance matrix is left unchanged.
- **svdnum** (`None` or `int`) – If positive, at most `svdnum` eigenmodes of the `xa` covariance matrix are retained; the modes with the smallest eigenvalues are discarded. `svdnum` is ignored if set to `None`.
- **rescale** (`bool`) – If `True`, rescale covariance matrix so diagonal elements all equal 1 before applying *svd* cuts. (Default is `True`.)
- **covfac** (`None` or number) – The covariance matrix (or matrices) of `xa` is multiplied by `covfac` if `covfac` is not `None`.

Returns Weighted average of the `xa` elements. The result has the same type and shape as each element of `xa` (that is, either a gvar.GVar or an array of gvar.GVars.)

The following function attributes are also set:

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

wavg.**dof**
Effective number of degrees of freedom.

wavg.**Q**
Quality factor *Q* for fit.

4.5 Utility Classes

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.

4.6 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

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX