gvar Documentation

Release 4.5.3

G. P. Lepage

January 07, 2014

CONTENTS

Contents:

CHAPTER

ONE

GVAR - GAUSSIAN RANDOM VARIABLES

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

1.2 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)
3.14150(20)
>>> x = gvar.gvar("3.1415(2)")
>>> print(x)
3.14150(20)
```

Function `gvar.asgvar(x)` returns x if it is a `gvar.GVar`; otherwise it returns `gvar.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 any correlations that exist 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 = 0.10(13)     y = 10.0(2.0)
```

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(2.0)
```

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)
1.010(13)
```

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

If the covariance matrix in `gvar.gvar` is diagonal, it can be replaced by an array of standard deviations (square roots of diagonal entries in `cov`). The example above, therefore, is equivalent to:

```
>>> x, y = gvar.gvar([0.1, 10.], [0.125, 2.])
>>> print('x =', x, '   y =', y)
x = 0.10(13)    y = 10.0(2.0)
```

1.3 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. The correlation matrix can be readily obtained as well:

```
>>> print(gvar.evalcorr([x, y, f]))
[[ 1.          0.          0.06237829]
 [ 0.          1.          0.99805258]
 [ 0.06237829  0.99805258  1.        ]]
```

It is often convenient to group related gvar.GVars 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.GVars and/or arbitrary arrays of gvar.GVars:

```
>>> 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]
```

1.4 Random Number Generators

gvar.GVars 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.GVars 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.GVars and have the distribution implied by those random variables (including correlations). For example,

```
>>> a = gvar.gvar(1.0, 1.0)
>>> da = gvar.gvar(0.0, 0.1)
>>> g = [a, a+da]
>>> giter = gvar.raniter(g)
>>> print(next(giter))
[ 1.51874589  1.59987422]
>>> print(next(giter))
[-1.39755111 -1.24780937]
>>> print(next(giter))
[ 0.49840244  0.50643312]
```

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

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

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

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

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

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

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

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

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

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

```
>>> g = gvar.gvar([1.1, 0.8], [[0.01, 0.005], [0.005, 0.01]])
>>> print(g)
[1.10(10) 0.80(10)]
>>> print(gvar.evalcov(g))                  # print covariance matrix
[[ 0.01   0.005]
 [ 0.005  0.01 ]]
>>> gbs_iter = gvar.bootstrap_iter(g)
>>> gbs = next(gbs_iter)                    # bootstrap copy of f
>>> print(gbs)
[1.14(10) 0.90(10)]                         # different means
>>> print(gvar.evalcov(gbs))
[[ 0.01   0.005]                            # same covariance matrix
 [ 0.005  0.01 ]]
```

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

1.5 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
0.00010(10)
```

Reducing `tiny`, however, leads to problems:

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

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

1.6 Implementation Notes; Derivatives; 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`).

The derivatives stored in a `gvar.GVar` are sometimes useful. Consider, for example, an array `x` each of whose elements was created by a call to `gvar.gvar()`: `x[i] = gvar.gvar(xi_mean,xi_sdev)`. Then derivatives of a function `f(x)` with respect to the `x[i]` can be computed from the `gvar.GVar` `fx = f(x)` using `fx.dotder(x[i].der)`, which equals `df(x)/dx[i]` at the point `x` specified by the means of the `x[i]`s. Note that this trick only works because the `x[i]` are among the underlying (original) `gvar.GVar`s (and not combinations of these).

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 — it used to be more important, in earlier implementations of `gvar.GVar` before sparse matrices were introduced to keep track of covariances. Note that the previous `gvar.gvar()` can be restored using `gvar.restore_gvar()`.

`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. Two `gvar.GVar`s are equal only if their means and derivatives are equal, and their covariance matrices the same. A `gvar.GVar` `x` is defined to equal a non-`gvar.GVar` `y` only if `x.mean == y` and `x.sdev == 0`.

The operators > and < are also defined. These allow `gvar.GVar`s to be ordered, which sometimes simplifies algorithm design. `gvar.GVar` `x` is defined to be greater than `gvar.GVar` `y` if `x.mean > y.mean`. Similarly `gvar.GVar` `x` is defined to be greater than a number `y` if `x.mean > y`. This definition is inconsistent with the definitions of == and != in that, for example, `not (x>y or x<y)` is *not* equivalent to `x==y`. Logically `x>y` for `gvar.GVar`s should evaluate to a boolean-valued random variable, but such variables are beyond the scope of this module. The operators > and < are included only because they facilitate algorithmic design. Operators >= and <= are *not* defined for `gvar.GVar`s.

1.7 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 = 2*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`**(*xdict*)
Returns a dictionary (`BufferDict`) `b` where `b[k] = gvar(xdict[k])` for every key in dictionary `xdict`. The values in `xdict`, therefore, can be strings, tuples or `gvar.GVar`s (see above), or arrays of these.

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

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

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

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

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

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

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

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

`gvar.`**`var`**(*g*)
Extract variances from `gvar.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.`**`fmt`**(*g*, *ndecimal=None*, *sep=''*)
Format `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. Each `gvar.GVar` `gi` in `g` is replaced by the string generated by `gi.fmt(ndecimal,sep)`. Result has same structure as `g`.

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

If `g` is an array of `gvar.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]`.

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

If `g` is an array of `gvar.GVar`s, `evalcorr` returns the correlation 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 `corr[k1,k2]` is the correlation for `g[k1]` and `g[k2]`.

The correlation matrix is related to the covariance matrix by:

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

gvar.**uncorrelated**(*g1*, *g2*)
Return `True` if `gvar.GVar`s in `g1` uncorrelated with those in `g2`.

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

gvar.**chi2**(*g1*, *g2*)
Compute chi**2 of `g1-g2`.

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

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

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

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

`chi**2` is computed from the inverse of the covariance matrix of `g1-g2`. The matrix inversion can be sensitive to roundoff errors. In such cases, *SVD* cuts can be applied by setting parameters `svdcut` and `svdnum`. See the documentation for `gvar.SVD` for information about these parameters.

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

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

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

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

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

Assumes `f` has attributes `chi2`, `dof` and `Q`.

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

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

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

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

gvar.**raniter**(*g*, *n=None*, *svdcut=None*, *svdnum=None*, *rescale=True*)

Return iterator for random samples from distribution g

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

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

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

Parameters

- **g** (*array or dictionary or BufferDict or GVar*) – An array (or dictionary) of objects of type gvar.GVar; or a gvar.GVar.
- **n** – Maximum number of random iterations. Setting n=None (the default) implies there is no maximum number.
- **svdcut** (None or number) – If positive, replace eigenvalues 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 before applying *svd* cuts 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.

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

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

Parameters

- **g** (*array or dictionary or BufferDict*) – An array (or dictionary) of objects of type gvar.GVar.
- **n** – Maximum number of random iterations. Setting n=None (the default) implies there is no maximum number.
- **svdcut** (None or number) – If positive, replace eigenvalues 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` before applying *svd* cuts 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.

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

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

Returns The seed.

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

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

Parameters

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

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

`gvar.`**`fmt_values`**`(`*outputs*, *ndecimal=None*`)`
Tabulate `gvar.GVar`s in `outputs`.

Parameters

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

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

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.GVars (to replace gvar.gvar()).

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

gvar.GVars 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.GVars, 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.GVars or a dictionary containing gvar.GVars and/or arrays of gvar.GVars. rebuild(g) creates a new collection gvar.GVars 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.GVars that replaces gvar.gvar() (the default).

Parameters

- **g** (*array or dictionary*) – gvar.GVars 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.GVars (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=True*)
Apply svd cuts to collection of gvar.GVars in g.

g is an array of gvar.GVars or a dictionary containing gvar.GVars and/or arrays of gvar.GVars. svd(g,...) returns a copy of g whose gvar.GVars 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. (Default is `True`.)
- **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`.

Returns A copy of `g` with the same means but with a covariance matrix modified by *svd* cuts.

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`

1.8 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 correlated with other gvar.GVars, the variance coming from these is included in the result as well. (This last convention is necessary because variances associated with correlated gvar.GVars cannot be disentangled into contributions corresponding to each variable separately.)

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 correlated with other gvar.GVars, the standard deviation coming from these is included in the result as well. (This last convention is necessary because variances associated with correlated gvar.GVars cannot be disentangled into contributions corresponding to each variable separately.)

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.

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

deriv(*x*)
Derivative of self with respest to *independent* gvar.GVar x.

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

All gvar.GVars are constructed from a set of independent gvar.GVars. self.deriv(x) returns the partial derivative of self with respect to independent gvar.GVar x, holding all of the other independent gvar.GVars constant.

Parameters **x** – The independent gvar.GVar.

Returns The derivative of self with respect to x.

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

__str__()
Return string representation of self.

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

fmt(*ndecimal=None*, *sep=''*)
Convert to string with format: mean(sdev).

Leading zeros in the standard deviation are omitted: for example, 25.67 +- 0.02 becomes 25.67(2). Parameter ndecimal specifies how many digits follow the decimal point in the mean.

Parameter `sep` is a string that is inserted between the `mean` and the `(sdev)`. If `ndecimal` is `None` (default), it is set automatically to the larger of `int(2-log10(self.sdev))` or `0`; this will display at least two digits of error. Very large or very small numbers are written with exponential notation when `ndecimal` is `None`.

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

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

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

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

dotder(*v*)
Return the dot product of `self.der` and `v`.

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

class `gvar.`**BufferDict**
Dictionary whose data is packed into a 1-d buffer (numpy.array).

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

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

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

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

or

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

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

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

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

```
 [ 50.  60.]]
>>> b['scalar'] += 1
>>> print(buf)
[  1.  10.  20.  30.  40.  50.  60.]
```

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

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

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

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

dtype
: Data type of buffer array elements.

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

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

The main methods are:

flatten()
: Copy of buffer array.

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

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

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

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

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

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

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

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

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

SVD analysis is handled by the following class:

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

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

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

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

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

Input parameters are:

Parameters

- **mat** (2-d sequence (`numpy.array` or `list` or ...)) – Positive, symmetric matrix.
- **svdcut** (`None` or number `(|svdcut|<=1)`.) – If positive, replace eigenvalues of `mat` with `svdcut*(max eigenvalue)`; if negative, discard eigenmodes with eigenvalues smaller than `svdcut` times the maximum eigenvalue.
- **svdnum** (`None` or int) – If positive, keep only the modes with the largest `svdnum` eigenvalues; ignore if set to `None`.
- **compute_delta** (*boolean*) – Compute `delta` (see below) if `True`; set `delta=None` otherwise.
- **rescale** – Rescale the input matrix to make its diagonal elements equal to 1.0 before diagonalizing.

The results are accessed using:

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

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

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

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

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.

1.9 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

TWO

GVAR.DATASET - RANDOM DATA SETS

2.1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
import yaml

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

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

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

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

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

creates the same dataset as above.

2.2 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(20)
>>> print(avg_data(random_arrays))
[11.3(1.1) 108(13)]
>>> print(avg_data(random_dict))
{'a': array([11.3(1.1), 108(13)], dtype=object),'n': 1.31(20)}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3 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

THREE

INDICES AND TABLES

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

PYTHON MODULE INDEX

g

INDEX

Q

R

S

T

U

V