gvar Documentation

Release 7.0.2

G. P. Lepage

June 17, 2015

Contents:

CHAPTER

ONE

OVERVIEW AND TUTORIAL

1.1 Introduction

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

A Gaussian variable `x` represents a Gaussian probability distribution, and is therefore completely characterized by its mean `x.mean` and standard deviation `x.sdev`. They are used to represent quantities whose values are uncertain: for example, the mass, 125.7±0.4 GeV, of the recently discovered Higgs boson from particle physics. The following code illustrates a (very) simple application of *gvar*; it calculates the Higgs boson's energy when it carries momentum 50±0.15 GeV.

```
>>> import gvar as gv
>>> m = gv.gvar(125.7, 0.4)              # Higgs boson mass
>>> p = gv.gvar(50, 0.15)                # Higgs boson momentum
>>> E = (p ** 2 +  m ** 2) ** 0.5        # Higgs boson energy
>>> print(m, E)
125.70(40) 135.28(38)
>>> print(E.mean, '+-', E.sdev)
135.279303665 +- 0.375787639425
```

Here method *gvar.gvar()* creates objects `m` and `p` of type *gvar.GVar* that represent Gaussian random variables for the Higgs mass and momentum, respectively. The energy `E` computed from the mass and momentum must, like them, be uncertain and so is also an object of type *gvar.GVar* — with mean `E.mean=135.28` and standard deviation `E.sdev=0.038`. (Note that *gvar* uses the compact notation 135.28(38) to represent a Gaussian variable, where the number in parentheses is the uncertainty in the corresponding rightmost digits of the quoted mean value.)

A highly nontrivial feature of *gvar.GVar*s is that they *automatically* track statistical correlations between different Gaussian variables. In the Higgs boson code above, for example, the uncertainty in the energy is due mostly to the initial uncertainty in the boson's mass. Consequently statistical fluctuations in the energy are strongly correlated with those in the mass, and largely cancel, for example, in the ratio:

```
>>> print(E / m)
1.07621(64)
```

The ratio is 4–5 times more accurate than the either the mass or energy separately.

The correlation between `m` and `E` is obvious from their covariance and correlation matrices, both of which have large off-diagonal elements:

```
>>> print gv.evalcov([m, E])             # covariance matrix
[[ 0.16        0.14867019]
 [ 0.14867019  0.14121635]]
```

```
>>> print gv.evalcorr([m, E])           # correlation matrix
[[ 1.          0.98905722]
 [ 0.98905722  1.        ]]
```

The correlation matrix shows that there is a 98.9% statistical correlation between the mass and energy.

A extreme example of correlation arises if we reconstruct the Higgs boson's mass from its energy and momentum:

```
>>> print((E ** 2 - m ** 2) / m ** 2)
1 +- 1.4e-18
```

The numerator and denominator are completely correlated, indeed identical to machine precision, as they should be. This works only because *gvar.GVar* object `E` knows that its uncertainty comes from the uncertainties associated with variables `m` and `p`.

We can verify that the uncertainty in the Higgs boson's energy comes mostly from its mass by creating an *error budget* for the Higgs energy (and for its energy to mass ratio):

```
>>> inputs = {'m':m, 'p':p}             # sources of uncertainty
>>> outputs = {'E':E, 'E/m':E/m}        # derived quantities
>>> print(gv.fmt_errorbudget(outputs=outputs, inputs=inputs))
Partial % Errors:
                     E        E/m
-------------------------------
        p:       0.04       0.04
        m:       0.27       0.04
-------------------------------
    total:       0.28       0.06
```

For each output (`E` and `E/m`), the error budget lists the contribution to the total uncertainty coming from each of the inputs (`m` and `p`). The total uncertainty in the energy is $\pm 0.28\%$, and almost all of that comes from the mass — only $\pm 0.04\%$ comes from the uncertainty in the momentum. The two sources of uncertainty contribute equally, however, to the ratio `E/m`, which has a total uncertainty of only 0.06%.

This example is relatively simple. Module *gvar*, however, can easily handle thousands of Gaussian random variables and all of their correlations. These can be combined in arbitrary arithmetic expressions and/or fed through complicated (pure) Python functions, while the *gvar.GVar*s automatically track uncertainties and correlations for and between all of these variables. The code for tracking correlations is the most complex part of the module's design, particularly since this is done automatically, behind the scenes.

What follows is a tutorial showing how to create *gvar.GVar*s and manipulate them to solve common problems in error propagation. Another way to learn about *gvar* is to look at the case studies later in the documentation. Each focuses on a single problem, and includes the full code and data, to allow for further experimentation.

gvar was originally written for use by the `lsqfit` module, which does multidimensional (Bayesian) least-squares fitting. It used to be distributed as part of `lsqfit`, but is now distributed separately because it is used by other modules (*e.g.*, `vegas` for multidimensional Monte Carlo integration).

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

```
from __future__ import print_function
```

at the start of your file.

1.2 Gaussian Random Variables

The Higgs boson mass (125.7±0.4 GeV) from the previous section is an example of a Gaussian random variable. As discussed above, such variables `x` represent Gaussian probability distributions, and therefore are completely characterized by their mean `x.mean` and standard deviation `x.sdev`. A mathematical function `f(x)` of a Gaussian variable is 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 `x.sdev` 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 formula, together with its multidimensional generalization, lead to a full calculus for Gaussian random variables that assigns Gaussian- variable values to arbitrary arithmetic expressions and functions involving Gaussian variables. This calculus, which is built into *gvar*, provides the rules for standard error propagation — an important application of Gaussian random variables and of the *gvar* module.

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 (or anti-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`.

1.3 Creating Gaussian Variables

Objects of type *gvar.GVar* are of two types: 1) primary *gvar.GVar*s that are created from means and covariances using *gvar.gvar()*; and 2) derived *gvar.GVar*s that result from arithmetic expressions or functions involving *gvar.GVar*s. The primary *gvar.GVar*s are the primordial sources of all uncertainties in a *gvar* code. A single (primary) *gvar.GVar* is created from its mean `xmean` and standard deviation `xsdev` using:

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

This function can also be used to convert strings like `"-72.374(22)"` or `"511.2 +- 0.3"` into *gvar.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)
>>> x = gvar.gvar("3.1415 +- 0.0002")
>>> print(x)
3.14150(20)
```

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

*gvar.GVar*s are usually more interesting when used to describe multidimensional distributions, especially if there are correlations between different variables. Such distributions are represented by collections of *gvar.GVar*s in

one of two standard formats: 1) `numpy` 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.

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]
```

As an example,

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

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

```
>>> print(gvar.evalcov([x, y]))     # covariance matrix
[[ 0.015625  0.24    ]
 [ 0.24      4.      ]]
>>> print(gvar.evalcorr([x, y]))    # correlation matrix
[[ 1.    0.96]
 [ 0.96  1.  ]]
```

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

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

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

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

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

1.4 gvar.GVar Arithmetic and Functions

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

As an example, again define

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

and set

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

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

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

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

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

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

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

The *gvar* module defines versions of the standard Python mathematical functions that work with *gvar.GVar* arguments. These include: `exp`, `log`, `sqrt`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `arctan2`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, `arctanh`. Numeric functions defined entirely in Python (*i.e.*, pure-Python functions) will likely also work with *gvar.GVar*s.

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

```
import math
import gvar

def erf(x):
    if isinstance(x, gvar.GVar):
```

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

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

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

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

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

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

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

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

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

1.5 Error Budgets from `gvar.GVar`s

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

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

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

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

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

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

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

1.6 Storing `gvar.GVar`s for Later Use; `gvar.BufferDict`s

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

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

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

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

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

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

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

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

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

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

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

1.7 Random Number Generators and Simulations

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

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

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

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

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

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

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

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

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

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

The standard deviation for `cos(x)` is obviously wrong since `cos(x)` can never be larger than one. We can estimate the the real mean and standard deviation using a simulation. To do this, we: 1) generate a large number of random numbers `xi` from `x`; 2) compute `cos(xi)` for each; and 3) 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()*.

Note finally 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.

1.8 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 to use simulations, as discussed in *Random Number Generators and Simulations*.

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).

1.9 Optimizations

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

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

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

CHAPTER

TWO

GVAR - GAUSSIAN RANDOM VARIABLES

2.1 Introduction

Objects of type *gvar.GVar* represent gaussian random variables, which are specified by a mean and standard deviation. They are created using *gvar.gvar()*: for example,

```
>>> x = gvar.gvar(10,3)           # 0 +- 3
>>> y = gvar.gvar(12,4)           # 2 +- 4
>>> z = x+y                      # 2 +- 5
>>> print(z)
22.0(5.0)
>>> print(z.mean)
22.0
>>> print(z.sdev)
5.0
```

This module contains a variety of tools for creating and manipulating gaussian random variables, including:

- *mean*(g) — extract means.
- *sdev*(g) — extract standard deviations.
- *var*(g) — extract variances.
- *chi2*(g1, g2) — chi**2 of g1-g2.
- *equivalent*(g1, g2) — *gvar.GVar*s the same in g1 and g2?
- *evalcov*(g) — compute covariance matrix.
- *evalcorr*(g) — compute correlation matrix.
- *fmt_values*(g) — create table of values for printing.
- *fmt_errorbudget*(g) — create error-budget table for printing.
- *fmt_chi2*(f) — format chi**2 information in f as string for printing.
- class *BufferDict* — ordered dictionary with data buffer.
- *dump*(g, outputfile) — pickle a collection of *gvar.GVar*s in file.
- *dumps*(g) — pickle a collection of *gvar.GVar*s in a string.
- *load*(inputfile) — reconstitute a collection of *gvar.GVar*s from a file.
- *loads*(inputstr) — reconstitute a collection of *gvar.GVar*s from a string.
- *raniter*(g,N) — iterator for random numbers.
- *bootstrap_iter*(g,N) — bootstrap iterator.

- *svd*(g) — SVD modification of correlation matrix.
- *dataset.bin_data*(data) — bin random sample data.
- *dataset.avg_data*(data) — estimate means of random sample data.
- *dataset.bootstrap_iter*(data,N) — bootstrap random sample data.
- class *dataset.Dataset* — class for collecting random sample data.

2.2 Functions

The function used to create Gaussian variable objects is:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
import math
import gvar as gv

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

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

Parameters

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

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

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

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

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

Elements of `g` that are not *gvar.GVar*s are left unchanged.

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`.

The deviation is set to 0.0 for elements of `g` that are not *gvar.GVar*s.

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`.

The variance is set to 0.0 for elements of `g` that are not *gvar.GVar*s.

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].flat and g[k2].flat.

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*, *svdcut=1e-15*, *fmt=False*)
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; see the documentation for *gvar.svd()*, which is used to apply the cut.

The return value is the chi**2. Extra attributes attached to this value give additional information:

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

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

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

*gvar.GVar*s are compared by:

gvar.**equivalent**(*g1*, *g2*)
Determine whether `g1` and `g2` contain equivalent *gvar.GVar*s.

This compares summs and differences of *gvar.GVar*s stored in `g1` and `g2` to see if they agree with tolerances. Operationally, agreement means that:

```
abs(diff) < abs(summ) / 2 * rtol + atol
```

where `diff` and `summ`` are the difference and sum of the mean values (``g.mean`) or derivatives (`g.der`) associated with each pair of *gvar.GVar*s.

*gvar.GVar*s that are equivalent are effectively interchangeable both respect to their means and also with respect to their covariances with any other *gvar.GVar* (including ones not in `g1` and `g2`).

`g1` and `g2` can be individual *gvar.GVar*s or arrays of *gvar.GVar*s or dictionaries whose values are *gvar.GVar*s and/or arrays of *gvar.GVar*s. Comparisons are made only for shared keys when they are dictionaries. Array dimensions must match between `g1` and `g2`, but the shapes can be different; comparisons are made for the parts of the arrays that overlap in shape.

Parameters

- **g1** – A *gvar.GVar* or an array of *gvar.GVar*s or a dictionary of *gvar.GVar*s and/or arrays of *gvar.GVar*s.
- **g2** – A *gvar.GVar* or an array of *gvar.GVar*s or a dictionary of *gvar.GVar*s and/or arrays of *gvar.GVar*s.
- **rtol** – Relative tolerance with which mean values and derivatives must agree with each other. Default is `1e-10`.
- **atol** – Absolute tolerance within which mean values and derivatives must agree with each other. Default is `1e-10`.

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

gvar.**dump**(*g*, *outputfile*)
pickle a collection `g` of *gvar.GVar*s in file `outputfile`.

The *gvar.GVar*s are recovered using *gvar.load()*.

Parameters

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

gvar.**dumps**(*g*)
pickle a collection `g` of *gvar.GVar*s and return as a string.

The *gvar.GVar*s are recovered using *gvar.loads()*.

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

gvar.**load**(*inputfile*)
Load and return pickled *gvar.GVar*s from file `inputfile`.

This function recovers *gvar.GVar*s pickled with *gvar.dump()*.

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

Returns The reconstructed *gvar.GVar*, or array or dictionary of *gvar.GVar*s.

gvar.**loads**(*inputstring*)
Load and return pickled *gvar.GVar*s from string `inputstring`.

This function recovers *gvar.GVar*s pickled with *gvar.dumps()*.

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

Returns The reconstructed *gvar.GVar*, or array or dictionary of *gvar.GVar*s.

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

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

gvar.**raniter**(*g*, *n=None*, *svdcut=None*)
Return iterator for random samples from distribution `g`

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

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

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

Parameters

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

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

gvar.**bootstrap_iter**(*g*, *n=None*, *svdcut=None*)
Return iterator for bootstrap copies of `g`.

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

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

Parameters

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

Returns An iterator that returns bootstrap copies of g.

gvar.**ranseed**(*a*)

Seed random number generators with tuple seed.

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

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

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

Returns The seed.

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

gvar.**fmt_errorbudget**(*outputs*, *inputs*, *ndecimal=2*, *percent=True*, *verify=False*, *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 distinguished). 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 or None*) – Width of each column. This is set automatically, to accommodate label widths, if colwidth=None (default).
- **verify** (*boolean*) – If True, a warning is issued if: 1) different inputs are correlated (and therefore double count errors); or 2) the sum (in quadrature) of partial errors is not

equal to the total error to within 0.1% of the error (and the error budget is incomplete or overcomplete). No checking is done if `verify==False` (default).

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 function applies an SVD cut to the correlation matrix of a set of *gvar.GVar*s:

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

Standard usage is, for example,

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

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

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

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

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

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

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

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

inv_cov = numpy.zeros((n, n), float)
i, wgts = i_wgts[0]                         # 1x1 sub-matrices
if len(i) > 0:
    inv_cov[i, i] = numpy.array(wgts) ** 2
for i, wgts in i_wgts[1:]:                  # nxn sub-matrices (n>1)
```

```
        for w in wgts:
            inv_cov[i, i[:, None]] += numpy.outer(w, w)
```

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

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

where `result` is the desired expectation value.

The input parameters are :

Parameters

- **g** – An array of *gvar.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 `eig` of the correlation matrix with `max(eig, svdcut * max_eig)` where `max_eig` is the largest eigenvalue; if negative, discard eigenmodes with eigenvalues smaller than `|svdcut| * max_eig`. Default is 1e-15.
- **wgts** – Setting `wgts=1` causes *gvar.svd()* to compute and return a spectral decomposition of the covariance matrix of the modified *gvar.GVar*s, `gmod`. Setting `wgts=-1` results in a decomposition of the inverse of the covariance matrix. The default value is `False`, in which case only `gmod` is returned.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Parameters

- **g** (*array or dictionary*) – *gvar.GVar*s to be rebuilt.
- **gvar** (`gvar.GVarFactory` or `None`) – Replacement for *gvar.gvar()* to use in rebuilding. Default is *gvar.gvar()*.
- **corr** (*number*) – Size of correlations to introduce where none exist initially.

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

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

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

Returns New *gvar.gvar()*.

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

Returns Previous *gvar.gvar()*.

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

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

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

2.3 `gvar.GVar` Objects

The fundamental class for representing Gaussian variables is:

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

mean
Mean value.

sdev
Standard deviation.

var
Variance.

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

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

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

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

Returns Partial variance due to all of `args`.

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

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

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

Returns Partial standard deviation due to `args`.

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

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

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

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

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

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

__str__()
Return string representation of `self`.

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

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

Leading zeros in the standard deviation are omitted: for example, `25.67 +- 0.02` becomes `25.67(2)`. Parameter `ndecimal` specifies how many digits follow the decimal point in the mean. Parameter `sep` is a string that is inserted between the `mean` and the `(sdev)`. If `ndecimal` is `None`

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

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

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

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

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

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

2.4 Other Classes

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**
Ordered dictionary whose data are packed into a 1-d buffer (numpy.array).

A *gvar.BufferDict* object is an ordered dictionary 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: *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` (since `BufferDict` is an ordered dictionary).

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

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

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

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

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

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

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

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

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

The main attributes are:

size
: Size of buffer array.

flat
: Buffer array iterator.

dtype
: Data type of buffer array elements.

buf
: The (1d) buffer array. Allows direct access to the buffer: for example, `self.buf[i] = new_val` sets the value of the `i-th` element in the buffer to value `new_val`. Setting `self.buf = nbuf` replaces

the old buffer by new buffer `nbuf`. This only works if `nbuf` is a one-dimensional `numpy` array having the same length as the old buffer, since `nbuf` itself is used as the new buffer (not a copy).

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

The main methods are:

flatten()
Copy of buffer array.

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

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

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

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

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

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

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

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

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

SVD analysis is handled by the following class:

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

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

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

```
    [ 0.22975292  0.97324899]

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

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

Input parameters are:

Parameters

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

The results are accessed using:

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

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

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

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

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

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

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

Computes vectors `w[i]` such that

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

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

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

Returns Array `w` of vectors.

2.5 Requirements

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

CHAPTER

THREE

GVAR.DATASET - RANDOM DATA SETS

3.1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
import yaml

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

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

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

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

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

creates the same dataset as above.

3.2 Functions

The functions defined in the module are:

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

Average random data to estimate mean.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Classes

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

class `gvar.dataset.`**Dataset**
Dictionary for collecting random data.

This dictionary class simplifies the collection of random data. The random data are stored in a dictionary, with each piece of random data being a number or a `numpy` 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 the data set is huge. To bin the data contained in file `datafile` in bins of binsize 2 we use:

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

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

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

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

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

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

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

The main attributes and methods are:

samplesize
Smallest number of samples for any key.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

slice (*sl*)

Create new dataset with `self[k] -> self[k][sl]`.

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

If parameter `sl` is a tuple of slice objects, these are applied to successive indices of `self[k]`. An exception is called if the number of slice objects exceeds the number of dimensions for any `self[k]`.

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()
Create dictionary `d` where `d[k]=numpy.array(self[k])` for all `k`.

CHAPTER

FOUR

NUMERICAL ANALYSIS MODULES IN GVAR

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

The sub-modules included here are:

- *gvar.cspline* — cubic splines for 1-d data.
- *gvar.linalg* — basic linear algebra.
- ***gvar.ode* — integration of systems of ordinary differential equations;** *one-dimensional integrals*.
- *gvar.powerseries* — power series representation of functions.
- *gvar.root* — root-finding for one-dimensional functions.

4.1 Cubic Splines

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

class `gvar.cspline.`**`CSpline`**(*xknots*, *yknots*, *deriv=(None*, *None)*, *extrap_order=3*, *warn=True*)

Cubic spline approximation to a function.

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

```
from gvar.cspline import CSpline

f = CSpline(xknot, yknot)
```

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

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

```
f = CSpline(xknot, yknot, deriv=[dydx_i, dydx_f])
```

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

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

`f.D(x)` — first derivative at `x`;

`f.D2(x)` — second derivative at `x`;

`f.integ(x)` — integral from `xknot[0]` to `x`.

Splines can be used outside the range covered by the defining `xknot` values. As this is often a bad idea, the `CSpline` methods issue a warning when called with out-of-range points. The warning can be suppressed by setting parameter `warn=False`. The spline value for an out-of-range point is calculated using a polynomial whose value and derivatives match those of the spline at the knot closest to the out-of-range point. The extrapolation polynomial is cubic by default, but lower orders can be specified by setting parameter `extrap_order` to a (non-negative) integer less than 3; this is often a good idea.

Examples

Typical usage is:

```
>>> import math
>>> import gvar as gv
>>> xknot = [0., 0.78539816, 1.57079633, 2.35619449, 3.14159265]
>>> yknot = [0., 0.70710678, 1.0, 0.70710678, 0.]
>>> f = gv.cspline.CSpline(xknot, yknot)
>>> print(f(0.7), f.D(0.7), f.D2(0.7), f.integ(0.7))
0.644243383101 0.765592448296 -0.663236750777 0.234963942648
```

Here the `yknot` values were obtained by taking `sin(xknot)`. Tabulating results from the spline together with the exact results shows that this 5-knot spline gives a pretty good approximation of the function `sin(x)`, as well as its derivatives and integral:

```
x    f(x)    f.D(x)   f.D2(x)  f.integ(x)  | sin(x)   cos(x)   1-cos(x)
-------------------------------------------------------------------------
0.3  0.2951  0.9551   -0.2842  0.04458     | 0.2955   0.9553   0.04466
0.5  0.4791  0.8793   -0.4737  0.1222      | 0.4794   0.8776   0.1224
0.7  0.6442  0.7656   -0.6632  0.235       | 0.6442   0.7648   0.2352
0.9  0.783   0.6176   -0.7891  0.3782      | 0.7833   0.6216   0.3784
1.1  0.8902  0.452    -0.8676  0.5461      | 0.8912   0.4536   0.5464
1.3  0.9627  0.2706   -0.9461  0.7319      | 0.9636   0.2675   0.7325
1.5  0.9974  0.07352  -1.025   0.9286      | 0.9975   0.07074  0.9293
```

Using the spline outside the range covered by the knots is less good:

```
>>> print(f(2 * math.pi))
gvar/cspline.py:164: UserWarning: x outside of spline range: [ 6.28318531]
1.7618635470106501
```

The correct answer is 0.0, of course. This is why the spline function issues a warning. Working just outside the knot region is often fine, although it is usually a good idea to limit the order of the polynomial used in such regions: for example, setting

```
>>> f = gv.cspline.CSpline(xknot, yknot, extrap_order=2)
```

implies that quadratic polynomials are used outside the spline range. Finally one can specify the values of the first derivatives of the function at one or the other endpoints of the spline region, if they are known. Continuing from above, for example, one would take

```
>>> f = gv.cspline.CSpline(xknot, yknot, deriv=[1., -1.])
```

since the derivatives of `sin(x)` at `x=0` and `x=3.14159265` are 1 and -1, respectively.

Parameters

- **xknot** (*1-d sequence of number*) – The knots of the spline, where the function values are specified. The knots are sorted (from small to large) if necessary.
- **yknot** (*1-d sequence of number*) – Function values at the locations specified by `xknot[i]`.
- **deriv** (*2-component sequence*) – Derivatives at initial and final boundaries of the region specified by `xknot[i]`. Default value is `None` for each boundary.
- **extrap_order** (*int*) – Order of polynomial used for extrapolations outside of the spline range. The polynomial is constructed from the spline's value and derivatives at the (nearest) knot of the spline. The allowed range is `0 <= extrap_order <= 3`. The default value is 3 although it is common practice to use smaller values.
- **warn** (*bool*) – If `True`, warnings are generated when the spline function is called for `x` values that fall outside of the original range of `xknots` used to define the spline. Default value is `True`; out-of-range warnings are suppressed if set to `False`.

4.2 Linear Algebra

The module *gvar.linalg* implements several methods for doing basic linear algebra with matrices whose elements can be either numbers or *gvar.GVar*s:

`linalg.`**`det`**`(`*a*`)`
Determinant of matrix `a`.

Parameters **a** – Two-dimensional, square matrix/array of numbers and/or *gvar.GVar*s.

Returns Deterimant of the matrix.

Raises `ValueError` – If matrix is not square and two-dimensional.

`linalg.`**`slogdet`**`(`*a*`)`
Sign and logarithm of determinant of matrix `a`.

Parameters **a** – Two-dimensional, square matrix/array of numbers and/or *gvar.GVar*s.

Returns Tuple `(s, logdet)` where the determinant of matrix `a` is `s * exp(logdet)`.

Raises `ValueError` – If matrix is not square and two-dimensional.

`linalg.`**`inv`**`(`*a*`)`
Inverse of matrix `a`.

Parameters **a** – Two-dimensional, square matrix/array of numbers and/or *gvar.GVar*s.

Returns The inverse of matrix `a`.

Raises `ValueError` – If matrix is not square and two-dimensional.

`linalg.`**`solve`**`(`*a*, *b*`)`
Find `x` such that `a.dot(x) = b` for matrix `a`.

Parameters

- **a** – Two-dimensional, square matrix/array of numbers and/or *gvar.GVar*s.
- **b** – One-dimensional vector/array of numbers and/or *gvar.GVar*s, or an array of such vectors. Requires `b.shape[0] == a.shape[1]`.

Returns The solution `x` of `a.dot(x) = b`, which is equivalent to `inv(a).dot(b)`.

Raises

- `ValueError` – If `a` is not square and two-dimensional.
- `ValueError` – If shape of `b` does not match that of `a` (that is `b.shape[0] != a.shape[1]`).

`linalg.`**`eigvalsh`**(*a*, *eigvec=False*)

Eigenvalues of Hermitian matrix `a`.

Parameters

- **a** – Two-dimensional, square matrix/array of numbers and/or *gvar.GVar*s.
- **eigvec** (*bool*) – If `True`, method returns a tuple of arrays `(val, vec)` where the `val[i]` are the eigenvalues. Arrays `vec[:, i]` are the corresponding eigenvectors of `a` when one ignores uncertainties (that is, they are eigenvectors of `gvar.mean(a)`). Only `val` is returned if `eigvec=False` (default).

Returns Array of eigenvalues of matrix `a` if parameter `eigvec==False` (default). where the `val[i]` are the eigenvalues; otherwise it returns a tuple of arrays `(val, vec)` where the `val[i]` are the eigenvalues. Arrays `vec[:, i]` are the corresponding eigenvectors of `a` when one ignores uncertainties (that is, they are eigenvectors of `gvar.mean(a)`).

Raises `ValueError` – If matrix is not square and two-dimensional.

4.3 Ordinary Differential Equations

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

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

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

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

```
from gvar.ode import Integrator

def f(x, y):
    return y

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

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

If the `interval` is a list with more than two entries, then `odeint(y0, interval=[x0, x1, x2 ...])` in the example above returns an array of solutions for points `x1, x2 ...`. So the example above could have been written equivalently as

```
...

odeint = Integrator(deriv=f,  tol=1e-8)
```

```
y0 = 1.
y1, y2 ... = odeint(y0, interval=[0, 1., 2. ...])
```

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

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

Here method `Integrator.solution()` returns a function `y(x)` where: a) `y(x0) = y0`; and b) `y(x)` uses the integator to integrate the differential equation to point `x` starting from the last point at which `y` was evaluated (or from `x0` for the first call to `y(x)`). The function can also be called with an array of `x` values, in which case an array containing the corresponding `y` values is returned.

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

A custom criterion for step-size changes can be implemented by specifying a function for parameter delta. This is a function `delta(yerr, y, delta_y)` — of the estimated error `yerr` after a given step, the proposed value for `y`, and the proposed change `delta_y` in `y` — that returns a number to compare with tolerance `tol`. The step size is decreased and the step repeated if `delta(yerr, y, delta_y) > tol`; otherwise the step is accepted and the step size increased. The default definition of `delta` is roughly equivalent to:

```
import numpy as np
import gvar as gv

def delta(yerr, y, delta_y):
    return np.max(
        np.fabs(yerr) / (np.fabs(y) + np.fabs(delta_y) + gv.ode.TINY)
        )
```

A custom definition can be used to allow an `Integrator` to work with data types other than floats or `numpy` arrays of floats. All that is required of the data type is that it support ordinary arithmetic. Therefore, for example, defining `delta(yerr, y, delta_y)` with `np.abs()` instead of `np.fabs()` allows `y` to be complex valued. (Actually the default `delta` allows this as well.)

An analyzer `analyzer(x,y)` can be specified using parameter `analyzer`. This function is called after every full step of the integration, with the current values of `x` and `y`. Objects of type *gvar.ode.Solution* are examples of (simple) analyzers.

Parameters

- **deriv** – Function of `x` and `y` that returns `dy/dx`. The return value should have the same shape as `y` if arrays are used.
- **tol** (*float*) – Relative accuracy in `y` relative to `|y| + h|dy/dx|` for each step in the integration. Any integration step that achieves less precision is repeated with a smaller step size. The step size is increased if precision is higher than needed. Default is 1e-5.
- **h** (*float or None*) – Absolute value of initial step size. The default value equals the entire width of the integration interval.

- **hmin** (*float or None*) – Smallest step size allowed. A warning is raised if a smaller step size is requested, and the step size is not decreased. This prevents infinite loops at singular points, but the solution may not be reliable when a warning has been issued. The default value is zero (which does *not* prevent infinite loops).
- **delta** – Function `delta(yerr, y, delta_y)` that returns a number to be compared with `tol` at each integration step: if it is larger than `tol`, the step is repeated with a smaller step size; if it is smaller the step is accepted and a larger step size used for the subsequent step. Here `yerr` is an estimate of the error in `y` on the last step; `y` is the proposed value; and `delta_y` is the change in `y` over the last step.
- **analyzer** – Function of `x` and `y` that is called after each step of the integration. This can be used to analyze intermediate results.

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

Integrate `dy/dx = deriv(x,y)` where `y` is a dictionary.

An *DictIntegrator* object `odeint` integrates `dy/dx = f(x,y)` to obtain `y(x1)` from `y0 = y(x0)`. `y` and `f(x,y)` are dictionary types having the same keys, and containing scalars and/or `numpy` arrays as values. Typical usage is:

```
from gvar.ode import DictIntegrator

def f(x, y):
    ...

odeint = DictIntegrator(deriv=f,  tol=1e-8)
y1 = odeint(y0, interval=(x0, x1))
y2 = odeint(y1, interval=(x1, x2))
...
```

The first call to `odeint` integrates from `x=x0` to `x=x1`, returning `y1=y(x1)`. The second call continues the integration to `x=x2`, returning `y2=y(x2)`. Multiple integration points can be specified in `interval`, in which case a list of the corresponding `y` values is returned: for example,

```
odeint = DictIntegrator(deriv=f,  tol=1e-8)
y1, y2 ... = odeint(y0, interval=[x0, x1, x2 ...])
```

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

An analyzer `analyzer(x,y)` can be specified using parameter `analyzer`. This function is called after every full step of the integration with the current values of `x` and `y`. Objects of type *gvar.ode.Solution* are examples of (simple) analyzers.

Parameters

- **deriv** – Function of `x` and `y` that returns `dy/dx`. The return value should be a dictionary with the same keys as `y`, and values that have the same shape as the corresponding values in `y`.
- **tol** (*float*) – Relative accuracy in `y` relative to `|y| + h|dy/dx|` for each step in the integration. Any integration step that achieves less precision is repeated with a smaller step size. The step size is increased if precision is higher than needed.
- **h** (*float*) – Absolute value of initial step size. The default value equals the entire width of the integration interval.

- **hmin** (*float*) – Smallest step size allowed. An exception is raised if a smaller step size is needed. This is mostly useful for preventing infinite loops caused by programming errors. The default value is zero (which does *not* prevent infinite loops).
- **analyzer** – Function of `x` and `y` that is called after each step of the integration. This can be used to analyze intermediate results.

A simple analyzer class is:

class `gvar.ode.`**Solution**
ODE analyzer for storing intermediate values.

Usage: eg, given

```
odeint = Integrator(...)
soln = Solution()
y0 = ...
y = odeint(y0, interval=(x0, x), analyzer=soln)
```

then the `soln.x[i]` are the points at which the integrator evaluated the solution, and `soln.y[i]` is the solution of the differential equation at that point.

4.4 One-Dimensional Integration

The module *gvar.ode* also provides a method for evaluating one-dimensional integrals (using its adaptive Runge-Kutta algorithm):

`ode.`**integral** (*fcn*, *interval*, *fcnshape=None*, *tol=1e-08*)
Compute integral of `fcn(x)` on interval.

Given a function `fcn(x)` the call

```
result = integral(fcn, interval=(x0, x1))
```

calculates the integral of `fcn(x)` from `x0` to `x1`. For example:

```
>>> def fcn(x):
...     return math.sin(x) ** 2 / math.pi
>>> result = integral(fcn, (0, math.pi))
>>> print(result)
0.500000002834
```

Function `fcn(x)` can return a scalar or an array (any shape): for example,

```
>>> def fcn(x):
...     return np.array([1., x, x**3])

>>> result = integral(fcn, (0,1))
>>> print(result)
[1.  0.5  0.25]
```

The function can also return dictionaries whose values are scalars or arrays: for example,

```
>>> def fcn(x):
...     return dict(x=x, x3=x**3)
>>> result = integral(fcn, (0,1))
>>> print(result)
{'x': 0.5,'x3': 0.25}
```

Parameters

- **fcn** – Function of scalar variable `x` that returns the integrand. The return value should be either a scalar or an array, or a dictionary whose values are scalars and/or arrays.
- **interval** – Contains the interval `(x0,x1)` over which the integral is computed.
- **fcnshape** – Contains the shape of the array returned by `f(x)` or `()` if the function returns a scalar. Setting `fshape=None` (the default) results in an extra function evaluation to determine the shape.
- **tol** – Relative accuracy of result.

4.5 Power Series

This module provides tools for manipulating power series approximations of functions. A function's power series is specified by the coefficients in its Taylor expansion with respect to an independent variable, say `x`:

```
f(x) = f(0) + f'(0)*x + (f''(0)/2)*x**2 + (f'''(0)/6)*x**3 + ...
     = f0 + f1*x + f2*x**2 + f3*x**3 + ...
```

In practice a power series is different from a polynomial because power series, while infinite order in principle, are truncated at some finite order in numerical applications. The `order` of a power series is the highest power of `x` that is retained in the approximation; coefficients for still higher-order terms are assumed to be unknown (as opposed to zero).

Taylor's theorem can be used to generate power series for functions of power series:

```
g(f(x)) = g(f0) + g'(f0)*(f(x)-f0) + (g''(f0)/2)*(f(x)-f0)**2 + ...
        = g0 + g1*x + g2*x**2 + ...
```

This allows us to define a full calculus for power series, where arithmetic expressions and (sufficiently differentiable) functions of power series return new power series.

4.5.1 Power series arithmetic

Class *PowerSeries* provides a numerical implementation of the power series calculus. `PowerSeries([f0,f1,f2,f3...])` is a numerical representation of a power series with coefficients `f0, f1, f2, f3...` (as in `f(x)` above). Thus, for example, we can define a 4th-order power series approximation `f` to `exp(x)=1+x+x**2/2+...` using

```
>>> from gvar.powerseries import *
>>> f = PowerSeries([1., 1., 1/2., 1/6., 1/24.])
>>> print f              # print the coefficients
[ 1.          1.          0.5         0.16666667  0.04166667]
```

Arithmetic expressions involving instances of class *PowerSeries* are themselves *PowerSeries* as in, for example,

```
>>> print 1/f            # power series for exp(-x)
[ 1.         -1.          0.5        -0.16666667  0.04166667]
>>> print log(f)         # power series for x
[ 0.  1.  0. -0.  0.]
>>> print f/f            # power series for 1
[ 1.  0.  0.  0.  0.]
```

The standard arithmetic operators (`+,-,*,/,=,**`) are supported, as are the usual elementary functions (`exp, log, sin, cos, tan ...`). Different *PowerSeries* can be combined arithmetically to create new *PowerSeries*; the order of the result is that of the operand with the lowest order.

PowerSeries can be differentiated and integrated:

```
>>> print f.deriv()      # derivative of exp(x)
[ 1.          1.          0.5         0.16666667]
>>> print f.integ()      # integral of exp(x) (from x=0)
[ 0.          1.          0.5         0.16666667  0.04166667  0.00833333]
```

Each *PowerSeries* represents a function. The *PowerSeries* for a function of a function is easily obtained. For example, assume `f` represents function `f(x)=exp(x)`, as above, and `g` represents `g(x)=log(1+x)`:

```
>>> g = PowerSeries([0, 1, -1/2., 1/3., -1/4.])
```

Then `f(g)` gives the *PowerSeries* for `exp(log(1+x)) = 1 + x`:

```
>>> print f(g)
[  1.0000e+00   1.0000e+00   0.0000e+00  -2.7755e-17 -7.6327e-17]
```

Individual coefficients from the powerseries can be accessed using array-element notation: for example,

```
>>> print f[0], f[1], f[2], f[3]
1.0 1.0 0.5 0.166666666667
>>> f[0] = f[0] - 1.
>>> print f              # f is now the power series for exp(x)-1
[ 0.          1.          0.5         0.16666667  0.04166667]
```

4.5.2 Numerical evaluation of power series

The power series can also be evaluated for a particular numerical value of x: continuing the example,

```
>>> x = 0.01
>>> print f(x)           # should be exp(0.01)-1 approximately
0.0100501670833
```

```
>>> print exp(x)-1       # verify that it is
0.0100501670842
```

The independent variable `x` could be of any arithmetic type (it need not be a `float`).

4.5.3 Taylor expansions of Python functions

PowerSeries can be used to compute Taylor series for more-or-less arbitrary pure-Python functions provided the functions are locally analytic (or at least sufficiently differentiable). To compute the `N`-th order expansion of a Python function `g(x)`, first create a `N`-th order *PowerSeries* variable that represents the expansion parameter: say, `x = PowerSeries([0.,1.],order=N)`. The Taylor series for function `g` is then given by `g_taylor = g(x)` which is a *PowerSeries* instance. For example, consider:

```
>>> from gvar.powerseries import *
>>> def g(x):            # an example of a Python function
...     return 0.5/sqrt(1+x) + 0.5/sqrt(1-x)
...
>>> x = PowerSeries([0.,1.],order=5)     # Taylor series for x
>>> print x
[ 0.  1.  0.  0.  0.  0.]
```

```
>>> g_taylor = g(x)      # Taylor series for g(x) about x=0
>>> print g_taylor
[ 1.          0.          0.375       0.          0.2734375   0.        ]
>>> exp_taylor = exp(x) # Taylor series for exp(x) about x=0
>>> print exp_taylor
[ 1.          1.          0.5         0.16666667  0.04166667  0.00833333]
```

class gvar.powerseries.**PowerSeries**(*c=None*, *order=None*)
Power series representation of a function.

The power series created by `PowerSeries(c)` corresponds to:

```
c[0] + c[1]*x + c[2]*x**2 + ... .
```

The order of the power series is normally determined by the length of the input list `c`. This can be overridden by specifying the order of the power series using the `order` parameter. The list of `c[i]`s is then padded with zeros if `c` is too short, or truncated if it is too long. Omitting `c` altogether results in a power series all of whose coefficients are zero. Individual series coefficients are accessed using array/list notation: for example, the 3rd-order coefficient of `PowerSeries p` is `p[3]`. The order of `p` is `p.order`. *PowerSeries* should work for coefficients of any data type that supports ordinary arithmetic.

Arithmetic expressions of *PowerSeries* variables yield new *PowerSeries* results that represent the power series expansion of the expression. Expressions can include the standard mathematical functions (`log, exp, sqrt, sin, cos, tan...`). *PowerSeries* can also be differentiated (`p.deriv()`) and integrated (`p.integ()`).

Parameters

- **c** (*list or array*) – Power series coefficients (optional if parameter *order* specified).
- **order** (*integer*) – Highest power in power series (optional if parameter *c* specified).

__iter__()
Iterate over coefficients of power series C{self}.

__setitem__(*i*, *val*)
Set C{i}th coefficient of power series equal to C{val}.

deriv(*n=1*)
Compute *n*-th derivative of `self`.

Parameters **n** (*positive integer*) – Number of derivatives.

Returns *n*-th derivative of `self`.

integ(*n=1*, *x0=None*)
Compute *n*-th indefinite integral of `self`.

If *x0* is specified, then the definite integral, integrating from point *x0*, is returned.

Parameters

- **n** (*integer*) – Number of integrations.
- **x0** – Starting point for definite integral (optional).

Returns *n*-th integral of `self`.

order
Highest power in power series.

4.6 Root Finding

The module *gvar.root* contains methods for finding the roots of of one-dimensional functions: that is, finding `x` such that `fcn(x)=0` for a given function `fcn`. Typical usage is:

```
>>> import math
>>> import gvar as gv
>>> interval = gv.root.search(math.sin, 1.)      # bracket root
>>> print(interval)
(3.1384283767210035, 3.4522712143931042)
>>> root = gv.root.refine(math.sin, interval)    # refine root
>>> print(root)
3.14159265359
```

This code finds the first root of `sin(x)=0` larger than 1. The first setp is a search to find an interval containing a root. Here *gvar.root.search()* examines `sin(x)` for a sequence of points `1.  * 1.1 ** n` for `n=0,1,2...`, stopping when the function changes sign. The last two points in the sequence then bracket a root since `sin(x)` is continuous; they are returned as a tuple to `interval`. The final root is found by refining the interval, using `gvar.root.refine`. By default, the root is refined iteratively to machine precision, but this requires only a small number (4) of iterations:

```
>>> print(root.nit)                             # number of iterations
4
```

The most challenging situations are ones where the function is extremely flat in the vicinity of the root — that is, two or more of its leading derivatives vanish there. For example:

```
>>> import gvar as gv
>>> def f(x):
...     return (x + 1) ** 3 * (x - 0.5) ** 11
>>> root = gv.root.refine(f, (0, 2))
>>> print(root)
0.5
>>> print(root.nit)                             # number of iterations
142
```

This routine also works with variables of type *gvar.GVar*: for example,

```
>>> import gvar as gv
>>> def f(x, w=gv.gvar(1, 0.1)):
...     return gv.sin(w * x)
>>> root = gv.root.refine(f, (1, 4))
>>> print(root)
3.14(31)
```

returns a root with a 10% uncertainty, reflecting the uncertainty in parameter `w`.

Descriptions of the two methods follow.

`root`.**search**(*fcn*, *x0*, *incr=0*, *fac=1.1*, *maxit=100*, *analyzer=None*)
Search for and bracket root of one-dimensional function `fcn(x)`.

This method searches for an interval in `x` that brackets a root of `fcn(x)=0`. It examines points

```
x[j + 1] = fac * x[j] + incr
```

where `x[0]=x0` and `j=0...maxit-1`, looking for a pair of successive points where `fcn(x[j])` changes sign. These points bracket a root (assuming the function is continuous), providing a coarse estimate of the root. That estimate can be refined using `root.refine()`.

Example

The following code seeks to bracket the first zero of `sin(x)` with `x>0.1`:

```
>>> import math
>>> import gvar as gv
>>> interval = gv.root.search(math.sin, 0.1)
>>> print(interval)
(3.0912680532870755, 3.4003948586157833)
```

The resulting interval correctly brackets the root at `pi`.

Parameters

- **fcn** – One dimenionsal function whose root is sought.
- **x0** (*float*) – Starting point for search.
- **incr** (*float, optional*) – Increment used for linear searches. Default value is 0.
- **fac** (*float, optional*) – Rescaling factor for exponential searches. Default value is 1.1.
- **maxit** (*int, optional*) – Maximum number of steps allowed for search. An exception is raised if a root is not found in time. Default value is 100.
- **analyzer** – Optional function `f(x, fcn(x))` that is called for each point `x` that is examined. This can be used, for example, to monitor the search while debugging. Default is `None`.

Returns

Tuple `(a, b)` where `fcn(a) * fcn(b) <= 0`, which implies that a root occurs between `a` and `b` (provided the function is continuous). The tuple has extra attributes that provide additional information about the search:

- **nit** — Number of iterations used to find interval `(a,b)`.
- **fcnval** — Tuple containing the function values at `(a,b)`.

Raises `RuntimeError` – If unable to find a root in `maxit` steps.

`root.`**`refine`**(*fcn*, *interval*, *rtol=None*, *maxit=1000*, *analyzer=None*)

Find root `x` of one-dimensional function `fcn` on an interval.

This method finds a root `x` of `fcn(x)=0` inside an `interval=(a,b)` that brackets the root, with `fcn(a) * fcn(b) <= 0`.

This method is a pure Python adaptation of an algorithm from Richard Brent's book "Algorithms for Minimization without Derivatives" (1973). Being pure Python it works with *gvar.GVar*-valued functions and variables.

Example

The following code finds a root of `sin(x)` in the interval `1 <= x <= 4`, using 7 iterative refinements of the initial interval:

```
>>> import math
>>> import gvar as gv
>>> root = gv.root.refine(math.sin, (1, 4))
>>> print(root)
3.14159265359
```

```
>>> print(root.nit)
7
```

Parameters

- **fcn** – One-dimensional function whose zero/root is sought.
- **interval** – Tuple `(a,b)` specifying an interval containing the root, with `fcn(a) * fcn(b) <= 0`. The search for a root is confined to this interval.
- **rtol** (*float, optional*) – Relative tolerance for the root. The default value is `None`, which sets `rtol` equal to machine precision (`sys.float_info.epsilon`). A larger value usually leads to less precision but is faster.
- **maxit** (*int, optional*) – Maximum number of iterations used to find a root with the given tolerance. A warning is issued if the algorithm does not converge in time. (Default value is 1000.)
- **analyzer** – Optional function `f(x, fcn(x))` that is called for each point `x` examined by the algorithm. This can be used, for example, to monitor convergence while debugging. Default is `None`.

Returns

The root, which is either a `float` or a *gvar.GVar* but with extra attributes that provide additional information about the root:

- **nit** — Number of iterations used to find the root.
- **interval** — Smallest interval `(b,c)` found containing the root, where `b` is the root returned by the method.
- **fcnval** — Value of `fcn(x)` at the root.

Raises

- `ValueError` – If `fcn(a) * fcn(b) > 0` for initial interval `(a,b)`.
- `UserWarning` – If the algorithm fails to converge after `maxit` iterations.

CHAPTER

FIVE

CASE STUDY: PENDULUM CLOCK

This case study illustrates how to mix *gvar.GVar*s with numerical routines for integrating differential equations (*gvar.ode*) and for finding roots of functions (*gvar.root*). It also gives a simple example of a simulation that uses *gvar.GVar*s.

5.1 The Problem

The precision of a particular pendulum clock is limited by two dominant factors: 1) the length of the pendulum (0.25m) can be adjusted with a precision of at best $\pm$0.5mm; and 2) irregularities in the drive mechanism cause the maximum angle of swing (π/6) to fluctuate by $\pm$0.025 radians. The challenge is to determine how these uncertainties affect time-keeping over a day.

The angle `theta(t)` of the pendulum satisfies a differential equation

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

where `g` is the acceleration due to gravity and the `l` is the length of the pendulum.

5.2 Pendulum Dynamics; Finding the Period

We start by designing code to integrate the differential equation:

```
import numpy as np
import gvar as gv

def make_pendulum(theta0, l):
    """ Create pendulum solution y(t) = [theta(t), d/dt theta(t)].

    Initial conditions are y(0) = [theta0, 0]. Parameter l is the
    length of the pendulum.
    """
    g_l = 9.8 / l
    def deriv(t, y):
        """ Calculate d/dt [theta(t), d/dt theta(t)]. """
        theta, dtheta_dt = y
        return np.array([dtheta_dt, - g_l * gv.sin(theta)])
    y0 = np.array([theta0, 0.0])
    return gv.ode.Integrator(deriv=deriv).solution(0.0, y0)
```

Given a solution `y(t)` of the differential equation from this method, we find the period of oscillation using *gvar.root*: the period is the time at which the pendulum returns to its starting point and its velocity (`y(t)[1]`) vanishes:

```
def find_period(y, Tapprox):
    """ Find oscillation period of pendulum solution y(t).

    Parameter Tapprox is the approximate period. The code finds the time
    between 0.7 * Tapprox and 1.3 * Tapprox where y(t)[1] = d/dt theta(t)
    vanishes. This is the period, provided Tapprox is correctly chosen.
    """
    def dtheta_dt(t):
        """ vanishes when dtheta/dt = 0 """
        return y(t)[1]
    return  gv.root.refine(dtheta_dt, (0.7 * Tapprox, 1.3 * Tapprox))
```

5.3 Analysis

The last piece of the code does the analysis:

```
def main():
    l = gv.gvar(0.25, 0.0005)               # length of pendulum
    theta_max = gv.gvar(np.pi / 6, 0.025)   # max angle of swing
    y = make_pendulum(theta_max, l)         # y(t) = [theta(t), d/dt  theta(t)]

    # period in sec
    T = find_period(y, Tapprox=1.0)
    print('period T = {} sec'.format(T))

    # uncertainty in minutes per day
    fmt = 'uncertainty = {:.2f} min/day\n'
    print(fmt.format((T.sdev / T.mean) * 60. * 24.))

    # error budget for T
    inputs = dict(l=l, theta_max=theta_max)
    outputs = dict(T=T)
    print(gv.fmt_errorbudget(outputs=outputs, inputs=inputs))

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

Here both the length of the pendulum and the maximum angle of swing have uncertainties and are represented by *gvar.GVar* objects. These uncertainties work their way through both the integration and root finding to give a final result for the period that is also a *gvar.GVar*. Running the code results in the following output:

```
period T = 1.0210(20) sec
uncertainty = 2.79 min/day

Partial % Errors:
                   T
--------------------
        l:      0.10
theta_max:      0.17
--------------------
    total:      0.19
```

The period is `T = 1.0210(20) sec`, which has an uncertainty of about $\pm 0.2\%$. This corresponds to an uncertainty of ± 2.8 min/day for the clock.

The uncertainty in the period is caused by the uncertainties in the length `l` and the angle of maximum swing `theta_max`. The error budget at the end of the output shows how much error comes from each source:

0.17% comes from the angle, and 0.10% comes from the length. (The two errors added in quadrature give the total.) We could have estimated the error due to the length from the standard formula 2π sqrt(l/g) for the period, which is approximately true here. Estimating the uncertainty due to the angle is trickier, since it comes from nonlinearities in the differential equation.

The error budget tells us how to improve the clock. For example, we can reduce the error due to the angle by redesigning the clock so that the maximum angle of swing is $\pi/36 \pm 0.025$ rather than $\pi/6 \pm 0.025$. The period becomes independent of the maximum angle as that angle vanishes, and so becomes less sensitive to uncertainties in it. Taking the smaller angle reduces that part of the period's error from 0.17% to 0.03%, thereby cutting the total error almost in half, to $\pm$0.10% or about $\pm$1.5 min/day. Further improvement requires tighter control over the length of the pendulum.

5.4 Simulation

We can check the error propagation analysis above using a simulation. Adding the following code at the end of `main()` above

```
# check errors in T using a simulation
Tlist = []
for i in range(100):
    y = make_pendulum(theta_max(), l())
    T = find_period(y, Tapprox=1.0)
    Tlist.append(T)
print('period T = {:.4f} +- {:.4f}'.format(np.mean(Tlist), np.std(Tlist)))
```

gives the following additional output:

```
period T = 1.0209 +- 0.0020
```

The new code generates 100 different values for the period `T`, corresponding to randomly chosen values for `theta_max` and `l` drawn from the Gaussian distributions corresponding to their *gvar.GVar*s. (In general, each call `x()` for *gvar.GVar* `x` is a new random number drawn from `x`'s Gaussian distribution.) The mean and standard deviation of the list of periods give us our final result. Results fluctuate with only 100 samples; taking 10,000 samples shows that the result is 1.0210(20), as we obtained in the previous section above (using a tiny fraction of the computer time).

Note that the *gvar.GVar*s in this simulation are uncorrelated and so their random values can be generated independently. *gvar.raniter()* should be used to generate random values from correlated *gvar.GVar*s.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

g

Symbols

A

B

C

D

E

F

G

I

L