vegas Documentation

Release 2.2

G.P. Lepage

June 28, 2014

Contents:

CHAPTER

ONE

TUTORIAL

1.1 Introduction

Class `vegas.Integrator` gives Monte Carlo estimates of arbitrary multidimensional integrals using the *vegas* algorithm (G. P. Lepage, J. Comput. Phys. 27 (1978) 192). The algorithm has two components. First an automatic transformation is applied to to the integration variables in an attempt to flatten the integrand. Then a Monte Carlo estimate of the integral is made using the transformed variables. Flattening the integrand makes the integral easier and improves the estimate. The transformation applied to the integration variables is optimized over several iterations of the algorithm: information about the integrand that is collected during one iteration is used to improve the transformation used in the next iteration.

Monte Carlo integration makes few assumptions about the integrand — it needn't be analytic nor even continuous. This makes Monte Carlo integation unusually robust. It also makes it well suited for adaptive integration. Adaptive strategies are essential for multidimensional integration, especially in high dimensions, because multidimensional space is large, with lots of corners, making it easy to lose important features in the integrand.

Monte Carlo integration also provides efficient and reliable methods for estimating the accuracy of its results. In particular, each Monte Carlo estimate of an integral is a random number from a distribution whose mean is the correct value of the integral. This distribution is Gaussian or normal provided the number of integrand samples is sufficiently large. In practive we generate multiple estimates of the integral in order to verify that the distribution is indeed Gaussian. Error analysis is straightforward if the integral estimates are Gaussian.

The `vegas` algorithm has been in use for decades and implementations are available in many programming languages, including Fortran (the original version), C and C++. The algorithm used here is significantly improved over the original implementation, and that used in most other implementations. It uses two adaptive strategies: importance sampling, as in the original implementation, and adaptive stratified sampling, which is new.

This module is written in Cython, so it is almost as fast as compiled Fortran or C, particularly when the integrand is also coded in Cython (or some other compiled language), as discussed below.

The following sections describe how to use `vegas`. Almost every example shown is a complete code, which can be copied into a file and run with python. It is worthwhile playing with the parameters to see how things change.

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 Basic Integrals

Here we illustrate the use of `vegas` by estimating the integral

$$C \int_{-1}^{1} dx_0 \int_{0}^{1} dx_1 \int_{0}^{1} dx_2 \int_{0}^{1} dx_3 \; \mathrm{e}^{-100 \sum_d (x_d - 0.5)^2},$$

where constant C is chosen so that the exact integral is 1. The following code shows how this can be done:

```
import vegas
import math

def f(x):
    dx2 = 0
    for d in range(4):
        dx2 += (x[d] - 0.5) ** 2
    return math.exp(-dx2 * 100.) * 1013.2118364296088

integ = vegas.Integrator([[-1, 1], [0, 1], [0, 1], [0, 1]])

result = integ(f, nitn=10, neval=1000)
print(result.summary())
print('result = %s    Q = %.2f' % (result, result.Q))
```

First we define the integrand `f(x)` where `x[d]` specifies a point in the 4-dimensional space. We then create an integrator, `integ`, which is an integration operator that can be applied to any 4-dimensional function. It is where we specify the integration volume. Finally we apply `integ` to our integrand `f(x)`, telling the integrator to estimate the integral using `nitn=10` iterations of the `vegas` algorithm, each of which uses no more than `neval=1000` evaluations of the integrand. Each iteration produces an independent estimate of the integral. The final estimate is the weighted average of the results from all 10 iterations, and is returned by `integ(f ...)`. The call `result.summary()` returns a summary of results from each iteration.

This code produces the following output:

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   2.4(1.9)        2.4(1.9)            0.00     1.00
  2   1.19(32)        1.23(32)            0.42     0.52
  3   0.910(90)       0.934(87)           0.68     0.51
  4   1.041(70)       0.999(55)           0.76     0.52
  5   1.090(43)       1.055(34)           1.00     0.41
  6   0.984(34)       1.020(24)           1.24     0.29
  7   1.036(27)       1.027(18)           1.07     0.38
  8   0.987(22)       1.011(14)           1.20     0.30
  9   0.995(18)       1.005(11)           1.11     0.35
 10   0.993(17)       1.0015(91)          1.02     0.42

result = 1.0015(91)    Q = 0.42
```

There are several things to note here:

> **Adaptation:** Integration estimates are shown for each of the 10 iterations, giving both the estimate from just that iteration, and the weighted average of results from all iterations up to that point. The estimates from the first two iterations are not accurate at all, with errors equal to 30–190% of the final result. `vegas` initially has no information about the integrand and so does a relatively poor job of estimating the integral. It uses information from the samples in one iteration, however, to remap the integration variables for subsequent iterations, concentrating samples where the function is largest and reducing errors. As a result, the per-iteration error is reduced to 4.3% by the fifth iteration, and below 2% by the end — an

improvement by almost two orders of magnitude from the start. Eventually the per-iteration error stops decreasing because `vegas` has found the optimal remapping, at which point it has fully adapted to the integrand.

Weighted Average: The final result, 1.0015 ± 0.0091, is obtained from a weighted average of the separate results from each iteration: estimates are weighted by the inverse variance, thereby giving much less weight to the early iterations, where the errors are largest. The individual estimates are statistical: each is a random number drawn from a distribution whose mean equals the correct value of the integral, and the errors quoted are estimates of the standard deviations of those distributions. The distributions are Gaussian provided the number of integrand evaluations per iteration (`neval`) is sufficiently large, in which case the standard deviation is a reliable estimate of the error. The weighted average $\overline{I}$ minimizes

$$\chi^2 \equiv \sum_i \frac{(I_i - \overline{I})^2}{\sigma_i^2}$$

where $I_i \pm \sigma_i$ are the estimates from individual iterations. If the I_i are Gaussian, χ^2 should be of order the number of degrees of freedom (plus or minus the square root of that number); here the number of degrees of freedom is the number of iterations minus 1.

The distributions are likely non-Gaussian, and error estimates unreliable, if χ^2 is much larger than the number of iterations. This criterion is quantified by the Q or *p-value* of the χ^2, which is the probability that a larger χ^2 could result from random (Gaussian) fluctuations. A very small Q (less than 0.05-0.1) indicates that the χ^2 is too large to be accounted for by statistical fluctuations — that is, the estimates of the integral from different iterations do not agree with each other to within errors. This means that `neval` is not sufficiently large to guarantee Gaussian behavior, and must be increased if the error estimates are to be trusted.

`integ(f...)` returns a weighted-average object, of type `vegas.RAvg`, that has the following attributes:

`result.mean` — weighted average of all estimates of the integral;

`result.sdev` — standard deviation of the weighted average;

`result.chi2` — χ^2 of the weighted average;

`result.dof` — number of degrees of freedom;

`result.Q` — Q or *p-value* of the weighted average's χ^2;

`result.itn_results` — list of the integral estimates from each iteration.

In this example the final Q is 0.42, indicating that the χ^2 for this average is not particularly unlikely and thus the error estimate is most likely reliable.

Precision: The precision of `vegas` estimates is determined by `nitn`, the number of iterations of the `vegas` algorithm, and by `neval`, the maximum number of integrand evaluation made per iteration. The computing cost is typically proportional to the product of `nitn` and `neval`. The number of integrand evaluations per iteration varies from iteration to iteration, here between 486 and 959. Typically `vegas` needs more integration points in early iterations, before it has fully adapted to the integrand.

We can increase precision by increasing either `nitn` or `neval`, but it is generally far better to increase `neval`. For example, adding the following lines to the code above

```
result = integ(f, nitn=100, neval=1000)
print('larger nitn  => %s    Q = %.2f' % (result, result.Q))

result = integ(f, nitn=10, neval=1e4)
print('larger neval => %s    Q = %.2f' % (result, result.Q))
```

generates the following results:

```
larger nitn  => 0.9968(15)    Q = 0.43
larger neval => 0.99978(67)    Q = 0.42
```

The total number of integrand evaluations, `nitn * neval`, is about the same in both cases, but increasing `neval` is more than twice as accurate as increasing `nitn`. Typically you want to use no more than 10 or 20 iterations beyond the point where `vegas` has fully adapted. You want some number of iterations so that you can verify Gaussian behavior by checking the χ^2 and Q, but not too many.

It is also generally useful to compare two or more results from values of `neval` that differ by a significant factor (4–10, say). These should agree within errors. If they do not, it could be due to non-Gaussian artifacts caused by a small `neval`. `vegas` estimates have two sources of error. One is the statistical error, which is what is quoted by `vegas`. The other is a systematic error due to residual non-Gaussian effects. The systematic error vanishes like `1/neval` and so becomes negligible compared with the statistical error as `neval` increases. The systematic error can bias the Monte Carlo estimate, however, if `neval` is insufficiently large. This usually results in a large χ^2 (and small Q), but a more reliable check is to compare results that use signficantly different values of `neval`. The systematic errors due to non-Gaussian behavior are likely negligible if the different estimates agree to within the statistical errors.

The possibility of systematic biases is another reason for increasing `neval` rather than `nitn` to obtain more precision. Making `neval` larger and larger is guaranteed to improve the Monte Carlo estimate, as the statistical error decreases (at least as fast as `sqrt(1/neval)` and often faster) and the systematic error decreases even more quickly (like `1/neval`). Making `nitn` larger and larger, on the other hand, is guaranteed eventually to give the wrong answer. This is because at some point the statistical error (which falls as `sqrt(1/nitn)`) will no longer mask the systematic error (which is unaffected by `nitn`). The systematic error for the integral above (with `neval=1000`) is about -0.0008(1), which is negligible compared to the statistical error unless `nitn` is of order 1500 or larger — so systematic errors aren't a problem with `nitn=10`.

Early Iterations: Integral estimates from early iterations, before `vegas` has adapted, can be quite crude. With very peaky integrands, these are often far from the correct answer with highly unreliable error estimates. For example, the integral above becomes more difficult if we double the length of each side of the integration volume by redefining `integ` as:

```
integ = vegas.Integrator([[-2, 2], [0, 2], [0, 2], [0., 2]])
```

The code above then gives:

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   0.013(13)       0.013(13)           0.00     1.00
  2   0.0165(80)      0.0154(67)          0.07     0.79
  3   2.07(96)        0.0155(67)          2.31     0.10
  4   0.86(26)        0.0160(67)          5.06     0.00
  5   1.01(11)        0.0199(67)         25.01     0.00
  6   0.963(64)       0.0302(67)         63.06     0.00
  7   1.032(41)       0.0561(66)        149.77     0.00
  8   0.924(31)       0.0924(64)        232.41     0.00
  9   1.037(28)       0.1410(63)        341.52     0.00
 10   0.976(22)       0.2026(60)        448.77     0.00

result = 0.2026(60)    Q = 0.00
```

`vegas` misses the peak completely in the first two iterations, giving estimates that are completely wrong (by 76 and 123 standard deviations!). Some of its samples hit the peak's shoulders, so `vegas` is eventually able to find the peak (by iterations 5–6), but the integrand estimates are wildly non-Gaussian before that point. This results in a non-sensical final result, as indicated by the `Q = 0.00`.

It is common practice in using `vegas` to discard estimates from the first several iterations, before the

algorithm has adapted, in order to avoid ruining the final result in this way. This is done by replacing the single call to `integ(f...)` in the original code with two calls:

```
# step 1 -- adapt to f; discard results
integ(f, nitn=7, neval=1000)

# step 2 -- integ has adapted to f; keep results
result = integ(f, nitn=10, neval=1000)
print(result.summary())
print('result = %s    Q = %.2f' % (result, result.Q))
```

The integrator is trained in the first step, as it adapts to the integrand, and so is more or less fully adapted from the start in the second step, which yields:

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   1.042(26)       1.042(26)           0.00     1.00
  2   1.010(18)       1.020(15)           0.99     0.32
  3   0.999(14)       1.009(10)           1.05     0.35
  4   0.965(16)       0.9963(86)          2.47     0.06
  5   0.994(15)       0.9958(74)          1.86     0.11
  6   1.001(15)       0.9968(66)          1.51     0.18
  7   0.999(15)       0.9971(61)          1.26     0.27
  8   0.994(13)       0.9965(55)          1.09     0.37
  9   1.017(22)       0.9977(53)          1.05     0.39
 10   0.981(16)       0.9961(50)          1.04     0.40

result = 0.9961(50)    Q = 0.40
```

The final result is now reliable.

Other Integrands: Once `integ` has been trained on `f(x)`, it can be usefully applied to other functions with similar structure. For example, adding the following at the end of the original code,

```
def g(x):
    return x[0] * f(x)

result = integ(g, nitn=10, neval=1000)
print(result.summary())
print('result = %s    Q = %.2f' % (result, result.Q))
```

gives the following new output:

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   0.5015(83)      0.5015(83)          0.00     1.00
  2   0.5099(68)      0.5065(53)          0.61     0.43
  3   0.4992(73)      0.5040(43)          0.63     0.53
  4   0.5121(61)      0.5066(35)          0.81     0.49
  5   0.5046(73)      0.5062(32)          0.62     0.65
  6   0.4918(63)      0.5033(28)          1.34     0.25
  7   0.5053(99)      0.5035(27)          1.12     0.35
  8   0.4997(69)      0.5030(25)          1.00     0.43
  9   0.5016(81)      0.5029(24)          0.88     0.54
 10   0.4951(75)      0.5021(23)          0.88     0.54

result = 0.5021(23)    Q = 0.54
```

Again the grid is almost optimal for `g(x)` from the start, because `g(x)` peaks in the same region as `f(x)`. The exact value for this integral is very close to 0.5.

Note that `vegas.Integrator`s can be saved in files and reloaded later using Python's `pickle` module: for example, `pickle.dump(integ, openfile)` saves integrator `integ` in file `openfile`, and `integ = pickle.load(openfile)` reloads it. The is useful for costly integrations that might need to be reanalyzed later since the integrator remembers the variable transformations made to minimize errors, and so need not be readapted to the integrand when used later.

Non-Rectangular Volumes: `vegas` can integrate over volumes of non-rectangular shape. For example, we can replace integrand `f(x)` above by the same Gaussian, but restricted to a 4-sphere of radius 0.2, centered on the Gaussian:

```
import vegas
import math

def f_sph(x):
    dx2 = 0
    for d in range(4):
        dx2 += (x[d] - 0.5) ** 2
    if dx2 < 0.2 ** 2:
        return math.exp(-dx2 * 100.) * 1115.3539360527281318
    else:
        return 0.0

integ = vegas.Integrator([[-1, 1], [0, 1], [0, 1], [0, 1]])

integ(f_sph, nitn=10, neval=1000)           # adapt the grid
result = integ(f_sph, nitn=10, neval=1000)  # estimate the integral
print(result.summary())
print('result = %s    Q = %.2f' % (result, result.Q))
```

The normalization is adjusted to again make the exact integral equal 1. Integrating as before gives:

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   1.005(41)       1.005(41)           0.00     1.00
  2   1.055(37)       1.033(27)           0.82     0.37
  3   1.048(63)       1.035(25)           0.43     0.65
  4   1.051(63)       1.037(23)           0.31     0.82
  5   0.994(23)       1.015(16)           0.68     0.61
  6   1.008(33)       1.014(15)           0.55     0.74
  7   1.030(34)       1.016(13)           0.49     0.82
  8   0.971(18)       1.000(11)           0.99     0.43
  9   1.005(34)       1.001(10)           0.87     0.54
 10   1.039(29)       1.0049(97)          0.94     0.48

result = 1.0049(97)    Q = 0.48
```

It is a good idea to make the actual integration volume as large a fraction as possible of the total volume used by `vegas` — by choosing integration variables properly — so `vegas` doesn't spend lots of effort on regions where the integrand is exactly 0. Also, it can be challenging for `vegas` to find the region of non-zero integrand in high dimensions: integrating `f_sph(x)` in 20 dimensions instead of 4, for example, would require `neval=1e16` integrand evaluations per iteration to have any chance of finding the region of non-zero integrand, because the volume of the 20-dimensional sphere is a tiny fraction of the total integration volume. The final error in the example above would have been cut in half had we used the integration volume `4 * [[0.3, 0.7]]` instead of `[[-1, 1], [0, 1], [0, 1], [0, 1]]`.

Note, finally, that integration to infinity is also possible: map the relevant variable into a different variable of finite range. For example, an integral over $x \equiv \tan(\theta)$ from 0 to infinity is easily reexpressed as an integral over θ from 0 to $\pi/2$.

Damping: This result in the previous section can be improved somewhat by slowing down vegas's adaptation:

```
...
integ(f_sph, nitn=10, neval=1000, alpha=0.1)
result = integ(f_sph, nitn=10, neval=1000, alpha=0.1)
...
```

Parameter alpha controls the speed with which vegas adapts, with smaller alphas giving slower adaptation. Here we reduce alpha to 0.1, from its default value of 0.5, and get the following output:

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   1.004(21)       1.004(21)           0.00     1.00
  2   0.988(24)       0.997(16)           0.26     0.61
  3   1.023(24)       1.005(13)           0.56     0.57
  4   0.996(19)       1.002(11)           0.43     0.73
  5   1.009(25)       1.0032(99)          0.34     0.85
  6   0.981(22)       0.9995(90)          0.44     0.82
  7   1.010(22)       1.0010(84)          0.40     0.88
  8   0.979(20)       0.9978(77)          0.48     0.85
  9   1.068(25)       1.0037(74)          1.29     0.24
 10   0.973(24)       1.0010(71)          1.32     0.22

result = 1.0031(72)    Q = 0.14
```

Notice how the errors fluctuate less from iteration to iteration with the smaller alpha in this case. Persistent, large fluctuations in the size of the per-iteration errors is often a signal that alpha should be reduced. With larger alphas, vegas can over-react to random fluctuations it encounters as it samples the integrand.

In general, we want alpha to be large enough so that vegas adapts quickly to the integrand, but not so large that it has difficulty holding on to the optimal tuning once it has found it. The best value depends upon the integrand.

adapt=False: Adaptation can be turned off completely by setting parameter adapt=False. There are three reasons one might do this. The first is if vegas is exhibiting the kind of instability discussed in the previous section — one might use the following code, instead of that presented there:

```
...
integ(f_sph, nitn=10, neval=1000, alpha=0.1)
result = integ(f_sph, nitn=10, neval=1000, adapt=False)
...
```

The second reason is that vegas runs slightly faster when it is no longer adapting to the integrand. The difference is not signficant for complicated integrands, but is noticable in simpler cases.

The third reason for turning off adaptation is that vegas uses unweighted averages, rather than weighted averages, to combine results from different iterations when adapt=False. Unweighted averages are not biased. They have no systematic error of the sort discussed above, and so give correct results even for very large numbers of iterations, nitn.

The lack of systematic biases is *not* a strong reason for turning off adaptation, however, since the biases are usually negligible (see above). Also, again, errors tend to fall faster if the number of evaluations per iteration neval is increased rather than the number of iterations. Finally in practice it is difficult to know precisely when vegas is finished adapting. One often finds (modest) continued improvement after the training step, leading to more accurate final results.

Training the integrator and then setting adapt=False for the final results works best if the number of evaluations per iteration (neval) is the same in both steps. This is because the second of vegas's

adaptation strategies (adaptive stratified sampling) is usually reinitialized when `neval` changes, and so is not used at all when `neval` is changed at the same time `adapt=False` is set.

1.3 Multiple Integrands Simultaneously

`vegas` can be used to integrate multiple integrands simultaneously, using the same integration points for each of the integrands. This is useful in situations where the integrands have similar structure, with peaks in the same locations. There can be signficant advantages in sampling different integrands at precisely the same points in x space, because then Monte Carlo estimates for the different integrals are correlated. If the integrands are very similar to each other, the correlations can be very strong. This leads to greatly reduced errors in ratios or differences of the resulting integrals as the fluctuations cancel.

Consider a simple example. We want to compute the normalization and first two moments of a sharply peaked probability distribution:

$$I_0 \equiv \int_0^1 d^4x\ \mathrm{e}^{-200\sum_d (x_d-0.5)^2}$$
$$I_1 \equiv \int_0^1 d^4x\ x_0\ \mathrm{e}^{-200\sum_d (x_d-0.5)^2}$$
$$I_2 \equiv \int_0^1 d^4x\ x_0^2\ \mathrm{e}^{-200\sum_d (x_d-0.5)^2}$$

From these integrals we determine the mean and width of the distribution projected onto one of the axes:

$$\langle x\rangle \equiv I_1/I_0$$
$$\sigma_x^2 \equiv \langle x^2\rangle - \langle x\rangle^2 = I_2/I_0 - (I_1/I_0)^2$$

This can be done using the following code:

```
import vegas
import math
import gvar as gv

def f(x):
    dx2 = 0.0
    for d in range(4):
        dx2 += (x[d] - 0.5) ** 2
    f = math.exp(-200 * dx2)
    return [f, f * x[0], f * x[0] ** 2]

integ = vegas.Integrator(4 * [[0, 1]])

# adapt grid
training = integ(f, nitn=10, neval=1000)

# final analysis
result = integ(f, nitn=10, neval=5000)
print('I[0] =', result[0], '  I[1] =', result[1], '  I[2] =', result[2])
print('Q = %.2f\n' % result.Q)
print('<x> =', result[1] / result[0])
print(
    'sigma_x**2 = <x**2> - <x>**2 =',
    result[2] / result[0] - (result[1] / result[0]) ** 2
```

```
    )
print('\ncorrelation matrix:\n', gv.evalcorr(result))
```

The code is very similar to that used in the previous section. The main difference is that the integrand function and `vegas` return arrays of results — in both cases, one result for each of the three integrals. `vegas` always adapts to the first integrand in the array. The `Q` value is for all three of the integrals, taken together.

The code produces the following output:

```
I[0] = 0.00024686(26)    I[1] = 0.00012347(13)    I[2] = 0.000062372(72)
Q = 0.86

<x> = 0.50016(11)
sigma_x**2 = <x**2> - <x>**2 = 0.002506(15)

correlation matrix:
 [[ 1.          0.98054444  0.92734262]
 [ 0.98054444  1.          0.98207749]
 [ 0.92734262  0.98207749  1.        ]]
```

The estimates for the individual integrals are separately accurate to about $\pm 0.1\%$, but the estimate for $\langle x\rangle = I_1/I_0$ is accurate to $\pm 0.02\%$. This is almost an order of magnitude (7x) more accurate than we would obtain absent correlations. The correlation matrix shows that there is 98% correlation between the statistical fluctuations in estimates for I_0 and I_1, and so the bulk of these fluctuations cancel in the ratio. The estimate for the variance σ_x^2 is 45x more accurate than we would have obtained had the integrals been evaluated separately. Both estimates are correct to within the quoted errors.

The individual results are objects of type `gvar.GVar`, which represent Gaussian random variables. Such objects have means (`result[i].mean`) and standard deviations (`result[i].sdev`), but also can be statistically correlated with other `gvar.GVars`. Such correlations are handled automatically by `gvar` when `gvar.GVars` are combined with each other or with numbers in arithmetical expressions. `vegas` provides a simplified implementation of `GVars` for use if the `gvar` module is not installed, but that version does *not* handle correlations at all (and, therefore, won't allow a statement like `result[1] / result[0]` above). To make full use of this `vegas` feature install the `gvar` module. It can be installed as part of the `lsqfit` distribution (e.g., `pip install lsqfit`) or by itself (e.g., `pip install gvar`); source code can be found at https://github.com/gplepage/lsqfit.git.

1.4 Faster Integrands

The computational cost of a realistic multidimensional integral comes mostly from the cost of evaluating the integrand at the Monte Carlo sample points. Integrands written in pure Python are probably fast enough for problems where `neval=1e3` or `neval=1e4` gives enough precision. Some problems, however, require hundreds of thousands or millions of function evaluations, or more.

We can significantly reduce the cost of evaluating the integrand by using `vegas`'s batch mode. For example, replacing

```
import vegas
import math

dim = 4
norm = 1013.2118364296088 ** (dim / 4.)

def f_scalar(x):
    dx2 = 0.0
    for d in range(dim):
        dx2 += (x[d] - 0.5) ** 2
    return math.exp(-100. * dx2) * norm
```

```
integ = vegas.Integrator(dim * [[0, 1]])

integ(f_scalar, nitn=10, neval=2e5)
result = integ(f_scalar, nitn=10, neval=2e5)
print('result = %s   Q = %.2f' % (result, result.Q))
```

by

```
import vegas
import numpy as np

class f_batch(vegas.BatchIntegrand):
    def __init__(self, dim):
        self.dim = dim
        self.norm = 1013.2118364296088 ** (dim / 4.)

    def __call__(self, x):
        # evaluate integrand at multiple points simultaneously
        dx2 = 0.0
        for d in range(self.dim):
            dx2 += (x[:, d] - 0.5) ** 2
        return np.exp(-100. * dx2) * self.norm

f = f_batch(dim=4)
integ = vegas.Integrator(f.dim * [[0, 1]], nhcube_batch=1000)

integ(f, nitn=10, neval=2e5)
result = integ(f, nitn=10, neval=2e5)
print('result = %s   Q = %.2f' % (result, result.Q))
```

reduces the cost of the integral by almost an order of magnitude. Internally `vegas` processes integration points in batches, where parameter `nhcube_batch` determines the number of points per batch (typically 1000s). In batch mode, `vegas` presents all of the integration points from a batch together, in a single array, to the integrand function, rather than offering them one at a time. Here, for example, an instance `f` of class `f_batch` behaves like a function `f(x)` of an array of integration points — `x[i, d]` where `i=0...` labels the integration point and `d=0...` the direction — and returns an array of integrand values corresponding to these points.

We derive class `f_batch` from `vegas.BatchIntegrand` to signal to `vegas` that it should present integration points in batches to the integrand function.

An alternative to deriving from `vegas.BatchIntegrand` is to apply the `vegas.batchintegrand()` decorator to a batch function for the integrand: e.g.,

```
import vegas
import numpy as np

dim = 4
norm = 1013.2118364296088 ** (dim / 4.)

@vegas.batchintegrand
def f(x):
    # evaluate integrand at multiple points simultaneously
    dx2 = 0.0
    for d in range(dim):
        dx2 += (x[:, d] - 0.5) ** 2
    return np.exp(-100. * dx2) * norm

integ = vegas.Integrator(dim * [[0, 1]], nhcube_batch=1000)
```

```
integ(f, nitn=10, neval=200000)
result = integ(f, nitn=10, neval=200000)
print('result = %s   Q = %.2f' % (result, result.Q))
```

This batch integrand is fast because it is expressed in terms `numpy` operators that act on entire arrays. That optimization is unnecessary (and the result is faster) if we write the integrand in Cython, which is a compiled hybrid of Python and C. The Cython version of this code is:

```
# file: cython_integrand.pyx

cimport vegas                   # for BatchIntegrand
from libc.math cimport exp      # use exp() from C library

import vegas
import numpy

cdef class f_cython(vegas.BatchIntegrand):
    cdef double norm
    cdef readonly int dim

    def __init__(self, dim):
        self.dim = dim
        self.norm = 1013.2118364296088 ** (dim / 4.)

    def __call__(self, double[:, ::1] x):
        cdef int i, d
        cdef double dx2
        cdef double[::1] f = numpy.empty(x.shape[0], float)
        for i in range(f.shape[0]):
            dx2 = 0.0
            for d in range(self.dim):
                dx2 += (x[i, d] - 0.5) ** 2
            f[i] = exp(-100. * dx2) * self.norm
        return f
```

We put this in a separate file called, say, `cython_integrand.pyx`, and rewrite the main code as:

```
import pyximport; pyximport.install()

import vegas
from cython_integrand import f_cython

f = f_cython(dim=4)
integ = vegas.Integrator(f.dim * [[0, 1]], nhcube_batch=1000)

integ(f, nitn=10, neval=200000)
result = integ(f, nitn=10, neval=200000)
print('result = %s   Q = %.2f' % (result, result.Q))
```

The first line (`import pyximport; ...`) causes the Cython module `cython_integrand.pyx` to be compiled the first time it is called. The compiled code is stored and used in subsequent calls, so compilation occurs only once.

Batch mode is also a good idea for array-valued integrands. The code from the previous section could have been written as:

```
import vegas
import gvar as gv
import numpy as np
```

```
dim = 4

@vegas.batchintegrand
def f(x):
    ans = np.empty((x.shape[0], 3), float)
    dx2 = 0.0
    for d in range(dim):
        dx2 += (x[:, d] - 0.5) ** 2
    ans[:, 0] = np.exp(-200 * dx2)
    ans[:, 1] = x[:, 0] * ans[:, 0]
    ans[:, 2] = x[:, 0] ** 2 * ans[:, 0]
    return ans

integ = vegas.Integrator(4 * [[0, 1]])

# adapt grid
training = integ(f, nitn=10, neval=1000)

# final analysis
result = integ(f, nitn=10, neval=5000)
print('I[0] =', result[0], '  I[1] =', result[1], '  I[2] =', result[2])
print('Q = %.2f\n' % result.Q)
print('<x> =', result[1] / result[0])
print(
    'sigma_x**2 = <x**2> - <x>**2 =',
    result[2] / result[0] - (result[1] / result[0]) ** 2
    )
print('\ncorrelation matrix:\n', gv.evalcorr(result))
```

Note that the batch index (here `:`) always comes first.

Cython code can also link easily to compiled C or Fortran code, so integrands written in these languages can be used as well (and would be faster than pure Python).

1.5 Multiple Processors

`vegas` code normally runs on a single CPU. It is possible to distribute the evaluation of the integrand over multiple processors by using the batch mode described in the previous section. This becomes worthwhile when the integrand becomes more expensive to evaluate.

`vegas` comes with a decorator, `vegas.MPIintegrand`, that adapts batch integrands for multiprocessor use through MPI. It assumes that Python module `mpi4py` is installed (and MPI, of course). To illustrate its use consider an integrand consisting of 1000 narrow Gaussians distributed evenly along the diagonal of a 4-dimensional unit hypercube. To maximize speed, we implement the integrand in Cython (it could have been done almost as easily in Fortran or C, or in Python), putting the result in a file `ridge.pyx`:

```
# file: ridge.pyx

from libc.math cimport exp      # use exp() from C library
import numpy as np

def f(double[:, ::1] x):
    cdef double dx2, x0
    cdef int d, i, j
    cdef int dim=4
    cdef int N=1000
```

```
    cdef double[::1] ans = np.zeros(x.shape[0], float)
    for i in range(x.shape[0]):
        for j in range(N):
            x0 = j / (N - 1.)
            dx2 = 0.0
            for d in range(dim):
                dx2 += (x[i, d] - x0) ** 2
            ans[i] += exp(-100. * dx2)
        ans[i] *= (100. / np.pi) ** 2 / N
    return ans
```

This is a standard batch integrand. The main integration code, in file `mpi-integral.py`, is then:

```
# file: mpi-integral.py

import pyximport; pyximport.install()  # compiles ridge.pyx

import numpy as np
import vegas
import ridge

def main():
    integ = vegas.Integrator(4 * [[0, 1]])
    # convert ridge.f into an MPI integrand
    fparallel = vegas.MPIintegrand(ridge.f)
    # adapt
    integ(fparallel, nitn=10, neval=1e5)
    # final results
    result = integ(fparallel, nitn=10, neval=1e5)
    if fparrallel.rank == 0:
        # result should be approximately 0.851
        print('result = %s    Q = %.2f' % (result, result.Q))

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

The integrand and main program are identical to what one would use for a batch integral except that `vegas.MPIintegrand(ridge.f)` is used in place of `vegas.batchintegrand(ridge.f)`, and we check the MPI rank of the process to avoid printing out multiple copies of the result, after the integration. To run this code on 4 CPUs, we might execute:

```
mpirun -np 4 python mpi-integral.py
```

This code runs 2.5–3 times faster on 4 CPUs than a single CPU. One might have hoped that 4 CPUs would be 4x faster, but they aren't quite that fast because of the time needed to transfer integration information between the processes on the different CPUs. Multiple CPUs are efficient only for costly integrands.

There are many other ways to implement multiprocessing for `vegas`. All methods work with `vegas` in batch mode, and distribute different integration points to different CPUs. For example, a class similar in function to `vegas.MPIintegrand`, but where Python's `multiprocessing` module replaces MPI, is:

```
import multiprocessing
import numpy as np
import vegas

class parallelintegrand(vegas.BatchIntegrand):
    """ Convert (batch) integrand into multiprocessor integrand.

    Integrand should return a numpy array.
```

```
    """
    def __init__(self, fcn, nproc=4):
        " Save integrand; create pool of nproc processes. "
        self.fcn = fcn
        self.nproc = nproc
        self.pool = multiprocessing.Pool(processes=nproc)
    def __del__(self):
        " Standard cleanup. "
        self.pool.close()
        self.pool.join()
    def __call__(self, x):
        " Divide x into self.nproc chunks, feeding one to each process. "
        nx = x.shape[0] // self.nproc + 1
        # launch evaluation of self.fcn for each chunk, in parallel
        po = self.pool.map_async(
            self.fcn,
            [x[i*nx : (i+1)*nx] for i in range(self.nproc)],
            1,
            )
        # harvest the results
        results = po.get()
        # convert list of results into a single numpy array
        return np.concatenate(results)
```

Then `fparallel = parallelintegrand(f, 4)`, for example, will create a new integrand `fparallel(x)` that uses 4 CPUs. This particular implementation of parallelism is not as efficient as the `vegas.MPIintegrand`.

1.6 Sums with vegas

The code in the previous sections is inefficient in the way it handles the sum over 1000 Gaussians. It is not necessary to include every term in the sum for every integration point. Rather we can sample the sum, using `vegas` to do the sampling. The trick is to replace the sum with an equivalent integral:

$$\sum_{i=0}^{N-1} f(i) = N \int_0^1 dx\; f(\text{floor}(xN))$$

where $\text{floor}(x)$ is the largest integer smaller than x. The resulting integral can then be handed to `vegas`. Using this trick, the integral in the previous section can be re-cast as a 5-dimensional integral (again in Cython),

```
# file: ridge.pyx

from libc.math cimport exp, floor
import numpy as np

def fsum(double[:, ::1] x):
    cdef double dx2, x0, j
    cdef int d, i
    cdef int dim=4
    cdef int N=1000
    cdef double[::1] ans = np.zeros(x.shape[0], float)
    for i in range(x.shape[0]):
        j = floor(x[i, -1] * N)
        x0 = j / (N - 1.)
        dx2 = 0.0
        for d in range(dim):
            dx2 += (x[i, d] - x0) ** 2
```

```
        ans[i] += exp(-100. * dx2)
        # drop 1/N because multiplying by N
        ans[i] *= (100. / np.pi) ** 2
    return np.asarray(ans)
```

and the main program becomes:

```
import pyximport; pyximport.install()

import numpy as np
import vegas
import ridge

def main():
    integ = vegas.Integrator(5 * [[0, 1]])
    f = vegas.batchintegrand(ridge.fsum)
    # adapt
    integ(f, nitn=10, neval=5e5)
    # final results
    result = integ(f, nitn=10, neval=5e5)
    # result should be approximately 0.851
    print('result = %s    Q = %.2f' % (result, result.Q))

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

This gives about the same precision but is 3x faster (on a laptop in 2014) than the code in the previous section.

The same trick can be generalized to sums over multiple indices, including sums to infinity. `vegas` will provide Monte Carlo estimates of the sums, emphasizing the more important terms.

1.7 `vegas` as a Random Number Generator

A `vegas` integrator generates random points in its integration volume from a distribution that is optimized for integrals of whatever function it was trained on. The integrator provides low-level access to the random-point generator through the iterators `vegas.Integrator.random()` and `vegas.Integrator.random_batch()`.

To illustrate, the following code snippet estimates the integral of function `f(x)` using integrator `integ`:

```
integral = 0.0
for x, wgt in integ.random():
    integral += wgt * f(x)
```

Here `x[d]` is a random point in the integration volume and `wgt` is the weight `vegas` assigns to that point in an integration. The iterator generates integration points and weights corresponding to a single iteration of the `vegas` algorithm. In practice, we would train `integ` on a function whose shape is similar to that of `f(x)` before using it to estimate the integral of `f(x)`.

It is usually more efficient to generate and use integration points in batches. The `vegas.Integrator.random_batch()` iterator does just this:

```
integral = 0.0
for x, wgt in integ.random_batch():
    integral += wgt.dot(batch_f(x))
```

Here `x[i, d]` is an array of integration points, `wgt[i]` contains the corresponding weights, and `batch_f(x)` returns an array containing the corresponding integrand values.

The random points generated by vegas are stratified into hypercubes: vegas uses transformed integration variables to improve its Monte Carlo estimates. It further improves those estimates by subdividing the integration volume in the transformed variables into a large number of hypercubes, and doing a Monte Carlo integral in each hypercube separately. The final result is the sum of the results from all the hypercubes. To mimic a full vegas integral estimate using the iterators above, we need to know which points belong to which hypercubes. The following code shows how this is done:

```
integral = 0.0
variance = 0.0
for x, wgt, hcube in integ.random_batch(yield_hcube=True):
    wgt_fx = wgt * batch_f(x)
    # iterate over hypercubes: compute variance for each,
    #                          and accumulate for final result
    for i in range(hcube[0], hcube[-1] + 1):
        idx = (hcube == i)          # select array items for h-cube i
        nwf = np.sum(idx)           # number of points in h-cube i
        wf = wgt_fx[idx]
        sum_wf = np.sum(wf)         # sum of wgt * f(x) for h-cube i
        sum_wf2 = np.sum(wf ** 2)   # sum of (wgt * f(x)) ** 2
        integral += sum_wf
        variance += (sum_wf2 * nwf - sum_wf ** 2) / (nwf - 1.)
# answer = integral;   standard deviation = variance ** 0.5
result = gvar.gvar(integral, variance ** 0.5)
```

Here `hcube[i]` identifies the hypercube containing `x[i, d]`.

1.8 Implementation Notes

This implementation relies upon Cython for its speed and numpy for array processing. It also uses matplotlib for graphics, but graphics is optional.

vegas also uses the `gvar` module from the `lsqfit` distribution if that package is installed (`pip install lsqfit` or `pip install gvar` for just `gvar`). Integration results are returned as objects of type `gvar.GVar`, which is a class representing Gaussian random variables (i.e., something with a mean and standard deviation). These objects can be combined with numbers and with each other in arbitrary arithmetic expressions to get new `gvar.GVar`s with the correct standard deviations, and properly correlated with other `gvar.GVar`s — that is the tricky part.

If `gvar` is not installed, vegas uses a limited substitute that supports arithmetic between `gvar.GVar`s and numbers, but not between `gvar.GVar`s and other `gvar.GVar`s. It also supports `log`, `sqrt` and `exp` of `gvar.GVar`s, but not trig functions — for these install the `gvar` module. Most importantly vegas will not provide correlation information for integrals of array-valued integrands unless the `gvar` module is available.

CHAPTER

TWO

HOW VEGAS WORKS

vegas uses two adaptive strategies: importance sampling, and adaptive stratified sampling. Here we discuss the ideas behind each, in turn.

2.1 Importance Sampling

The most important adaptive strategy vegas uses is its remapping of the integration variables in each direction, before it makes Monte Carlo estimates of the integral. This is equivalent to a standard Monte Carlo optimization called "importance sampling."

vegas chooses transformations for each integration variable that minimize the statistical errors in Monte Carlo estimates whose integrand samples are uniformly distributed in the new variables. The idea in one-dimension, for example, is to replace the original integral over x,

$$I = \int_a^b dx\ f(x),$$

by an equivalent integral over a new variable y,

$$I = \int_0^1 dy\ J(y)\ f(x(y)),$$

where $J(y)$ is the Jacobian of the transformation. A simple Monte Carlo estimate of the transformed integral is given by

$$I \approx S^{(1)} \equiv \frac{1}{M} \sum_y\ J(y)\ f(x(y))$$

where the sum is over M random points uniformly distributed between 0 and 1.

The estimate $S^{(1)}$ is a itself a random number from a distribution whose mean is the exact integral and whose variance is:

$$\begin{aligned}\sigma_I^2 &= \frac{1}{M}\left(\int_0^1 dy\ J^2(y)\ f^2(x(y)) - I^2\right)\\ &= \frac{1}{M}\left(\int_a^b dx\ J(y(x))\ f^2(x) - I^2\right)\end{aligned}$$

The standard deviation σ_I is an estimate of the possible error in the Monte Carlo estimate. A straightforward variational calculation, constrained by

$$\int_a^b \frac{dx}{J(y(x))} = \int_0^1 dy = 1,$$

shows that σ_I is minimized if

$$J(y(x)) = \frac{\int_a^b dx\ |f(x)|}{|f(x)|}.$$

Such transformations greatly reduce the standard deviation when the integrand has high peaks. Since

$$1/J = \frac{dy}{dx} \propto |f(x)|,$$

the regions in x space where $|f(x)|$ is large are stretched out in y space. Consequently, a uniform Monte Carlo in y space places more samples in the peak regions than it would if were we integrating in x space — its samples are concentrated in the most important regions, which is why this is called "importance sampling." The product $J(y)\ f(x(y))$ has no peaks when the transformation is optimal.

The distribution of the Monte Carlo estimates $S^{(1)}$ becomes Gaussian in the limit of large M. Non-Gaussian corrections vanish like $1/M$. For example, it is easy to show that

$$\langle (S^{(1)} - I)^4 \rangle = 3\sigma_I^4 \left(1 - \frac{1}{M}\right) + \frac{1}{M^3} \int_0^1 dy\ (J(y)\ f(x(y)) - I)^4$$

This moment would equal $3\sigma_I^4$, which falls like $1/M^2$, if the distribution was Gaussian. The corrections to the Gaussian result fall as $1/M^3$ and so become negligible at large M. These results assume that $(J(y)\ f(x(y)))^n$ is integrable for all n, which need not be the case if $f(x)$ has (integrable) singularities.

2.2 The vegas Grid

vegas implements the transformation of an integration variable x into a new variable y using a grid in x space:

$$\begin{aligned}
x_0 &= a \\
x_1 &= x_0 + \Delta x_0 \\
x_2 &= x_1 + \Delta x_1 \\
&\cdots \\
x_N &= x_{N-1} + \Delta x_{N-1} = b
\end{aligned}$$

The grid specifies the transformation function at the points $y = i/N$ for $i = 0, 1 \ldots N$:

$$x(y{=}i/N) = x_i$$

Linear interpolation is used between those points. The Jacobian for this transformation function is piecewise constant:

$$J(y) = J_i = N\Delta x_i$$

for $i/N < y < (i+1)/N$.

The variance for a Monte Carlo estimate using this transformation becomes

$$\sigma_I^2 = \frac{1}{M}\left(\sum_i J_i \int_{x_i}^{x_{i+1}} dx\ f^2(x) - I^2\right)$$

Treating the J_i as independent variables, with the constraint

$$\sum_i \frac{\Delta x_i}{J_i} = \sum_i \Delta y_i = 1,$$

it is trivial to show that the standard deviation is minimized when

$$\frac{J_i^2}{\Delta x_i} \int_{x_i}^{x_{i+1}} dx\ f^2(x) = N^2 \Delta x_i \int_{x_i}^{x_{i+1}} dx\ f^2(x)\ =\ \text{constant}$$

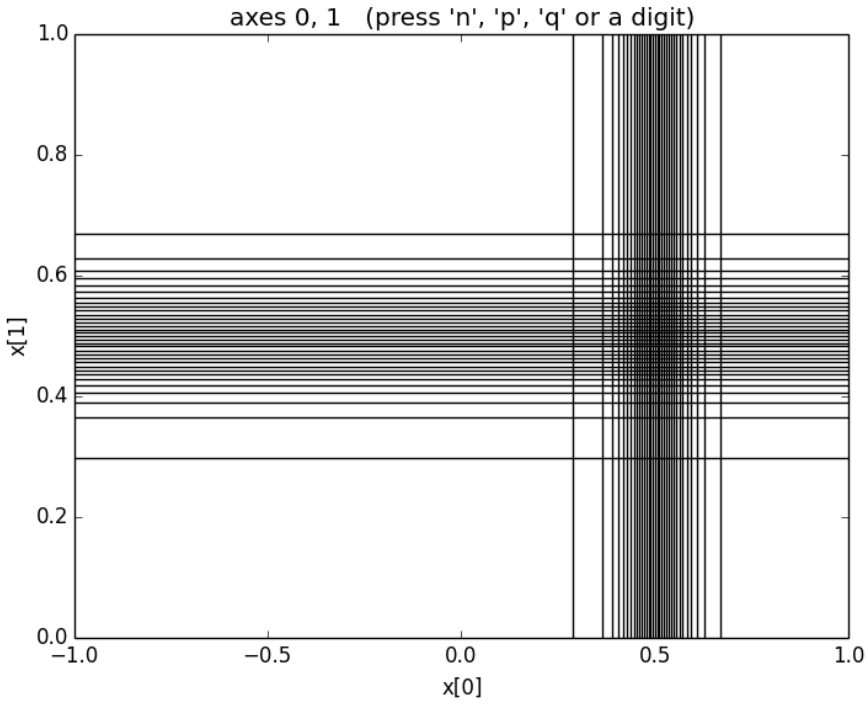
for all i.

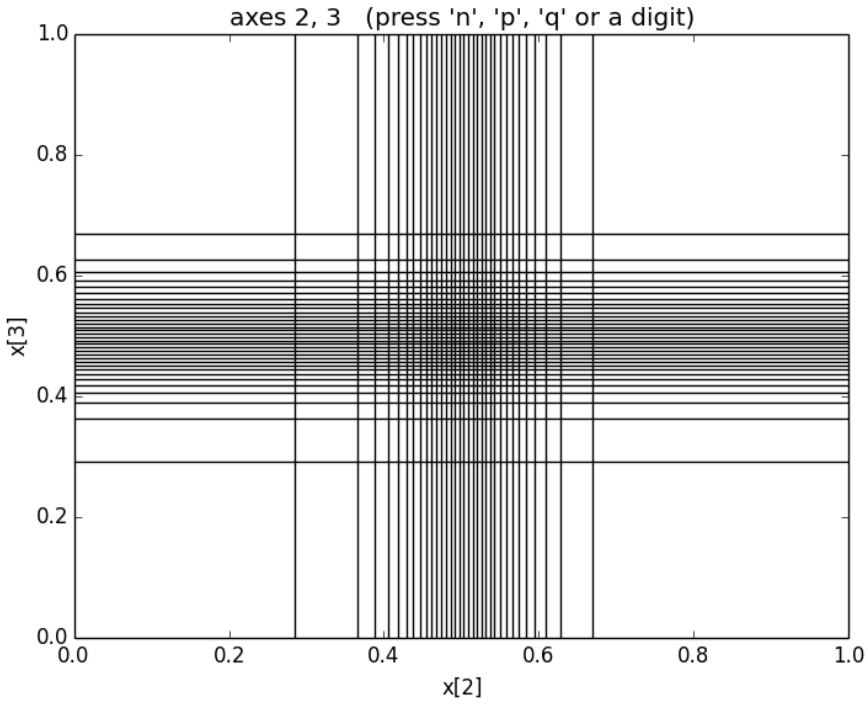
vegas adjusts the grid until this last condition is satisfied. As a result grid increments Δx_i are small in regions where $|f(x)|$ is large. vegas typically has no knowledge of the integrand initially, and so starts with a uniform x grid. As it samples the integrand it also estimates the integrals

$$\int_{x_i}^{x_{i+1}} dx\ f^2(x),$$

and use this information to refine its choice of Δx_is, bringing them closer to their optimal values, for use in subsequent iterations. The grid usually converges, after several iterations, to the optimal grid.

This analysis generalizes easily to multi-dimensional integrals. vegas applies a similar transformation in each direction, and the grid increments along an axis are made smaller in regions where the projection of the integral onto that axis is larger. For example, the optimal grid for the four-dimensional Gaussian integral in the previous section looks like:





These grids transform into uniformly-spaced grids in y space. Consequently a uniform, y-space Monte Carlo places the same number of integrand evaluations, on average, in every rectangle of these pictures. (The average number is typically much less one in higher dimensions.) Integrand evaluations are concentrated in regions where the x-space rectangles are small (and therefore numerous) — here in the vicinity of `x = [0.5, 0.5, 0.5, 0.5]`, where the peak is.

These plots were obtained by including the line

```
integ.map.show_grid(30)
```

in the integration code after the integration is finished. It causes `matplotlib` (if it is installed) to create images showing the locations of 30 nodes of the grid in each direction. (The grid uses 99 nodes in all on each axis, but that is too many to display at low resolution.)

2.3 Adaptive Stratified Sampling

A limitation of `vegas`'s remapping strategy becomes obvious if we look at the grid for the following integral, which has two Gaussians arranged along the diagonal of the hypercube:

```
import vegas
import math

def f2(x):
    dx2 = 0
    for d in range(4):
        dx2 += (x[d] - 1/3.) ** 2
    ans = math.exp(-dx2 * 100.) * 1013.2167575422921535
    dx2 = 0
    for d in range(4):
        dx2 += (x[d] - 2/3.) ** 2
```

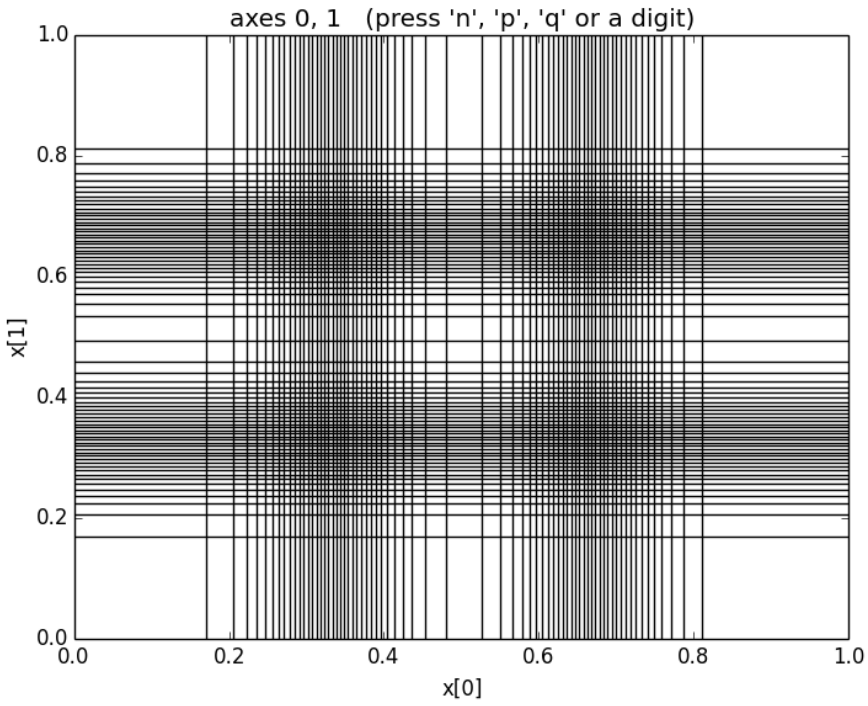
```
    ans += math.exp(-dx2 * 100.) * 1013.2167575422921535
    return ans / 2.

integ = vegas.Integrator(4 * [[0, 1]])

integ(f2, nitn=10, neval=4e4)
result = integ(f2, nitn=30, neval=4e4)
print('result = %s    Q = %.2f' % (result, result.Q))

integ.map.show_grid(70)
```

This code gives the following grid, now showing 70 nodes in each direction:



The grid shows that `vegas` is concentrating on the regions around `x=[0.33, 0.33, 0.33, 0.33]` and `x=[0.67, 0.67, 0.67, 0.67]`, where the peaks are. Unfortunately it is also concentrating on regions around points like `x=[0.67, 0.33, 0.33, 0.33]` where the integrand is very close to zero. There are 14 such phantom peaks that `vegas`'s new integration variables emphasize, in addition to the 2 regions where the integrand actually is large. This grid gives much better results than using a uniform grid, but it obviously wastes integration resources. The waste occurs because `vegas` remaps the integration variables in each direction separately. Projected on the `x[0]` axis, for example, this integrand appears to have two peaks and so `vegas` will focus on both regions of `x[0]`, independently of what it does along the `x[1]` axis.

`vegas` uses axis-oriented remappings because other alternatives are much more complicated and expensive; and `vegas`'s principal adaptive strategy has proven very effective in many realistic applications.

An axis-oriented strategy will always have difficulty adapting to structures that lie along diagonals of the integration volume. To address such problems, the new version of `vegas` introduces a second adaptive strategy, based upon another standard Monte Carlo technique called "stratified sampling." `vegas` divides the d-dimensional y-space volume into $M_{\rm st}^d$ hypercubes using a uniform y-space grid with $M_{\rm st}$ stratifications on each axis. It estimates the integral by doing a separate Monte Carlo integration in each of the hypercubes, and adding the results together to provide an estimate for the integral over the entire integration region. Typically this y-space grid is much coarser than the x-space grid used to remap the integration variables. This is because `vegas` needs at least two integrand evaluations in each

y-space hypercube, and so must keep the number of hypercubes $M_{\rm st}^d$ smaller than `neval/2`. This can restrict $M_{\rm st}$ severely when d is large.

Older versions of `vegas` also divide y-space into hypercubes and do Monte Carlo estimates in the separate hypercubes. These versions, however, use the same number of integrand evaluations in each hypercube. In the new version, `vegas` adjusts the number of evaluations used in a hypercube in proportion to the standard deviation of the integrand estimates (in y space) from that hypercube. It uses information about the hypercube's standard deviation in one iteration to set the number of evaluations for that hypercube in the next iteration. In this way it concentrates integrand evaluations where the potential statistical errors are largest.

In the two-Gaussian example above, for example, the new `vegas` shifts integration evaluations away from the phantom peaks, into the regions occupied by the real peaks since this is where all the error comes from. This improves `vegas`'s ability to estimate the contributions from the real peaks and reduces statistical errors, provided `neval` is large enough to permit a large number (more than 2 or 3) $M_{\rm st}$ of stratifications on each axis. With `neval=4e4`, statistical errors for the two-Gaussian integral are reduced by more than a factor of 3 relative to what older versions of `vegas` give. This is a relatively easy integral; the difference can be much larger for more difficult (and realistic) integrals.

CHAPTER

THREE

VEGAS MODULE

3.1 Introduction

The key Python objects supported by the `vegas` module are:

- `vegas.Integrator` — an object describing a multidimensional integration operator. Such objects contain information about the integration volume, and also about optimal remappings of the integration variables based upon the last integral evaluated using the object.
- `vegas.AdaptiveMap` — an object describing the remappings used by `vegas`.
- `vegas.RAvg` — an object describing the result of a `vegas` integration. `vegas` returns the weighted average of the integral estimates from each `vegas` iteration as an object of class `vegas.RAvg`. These are Gaussian random variables — that is, they have a mean and a standard deviation — but also contain information about the iterations `vegas` used in generating the result.
- `vegas.RAvgArray` — an array version of `vegas.RAvg` used when the integrand is array-valued.

These are described in detail below.

3.2 Integrator Objects

The central component of the `vegas` package is the integrator class:

class `vegas.`**`Integrator`**

Adaptive multidimensional Monte Carlo integration.

`vegas.Integrator` objects make Monte Carlo estimates of multidimensional functions `f(x)` where `x[d]` is a point in the integration volume:

```
integ = vegas.Integrator(integration_region)

result = integ(f, nitn=10, neval=10000)
```

The integator makes `nitn` estimates of the integral, each using at most `neval` samples of the integrand, as it adapts to the specific features of the integrand. Successive estimates (iterations) typically improve in accuracy until the integrator has fully adapted. The integrator returns the weighted average of all `nitn` estimates, together with an estimate of the statistical (Monte Carlo) uncertainty in that estimate of the integral. The result is an object of type `RAvg` (which is derived from `gvar.GVar`).

Integrands can be array-valued, in which case `f(x)` returns an array of values corresponding to different integrands. Also `vegas` can generate integration points in batches for integrands built from classes derived from `vegas.BatchIntegrand`, or integrand functions decorated by `vegas.batchintegrand()`. Batch integrands are typically much faster, especially if they are coded in Cython.

vegas.Integrators have a large number of parameters but the only ones that most people will care about are: the number nitn of iterations of the vegas algorithm; the maximum number neval of integrand evaluations per iteration; and the damping parameter alpha, which is used to slow down the adaptive algorithms when they would otherwise be unstable (e.g., with very peaky integrands). Setting parameter analyzer=vegas.reporter() is sometimes useful, as well, since it causes vegas to print (on sys.stdout) intermediate results from each iteration, as they are produced. This helps when each iteration takes a long time to complete (e.g., an hour) because it allows you to monitor progress as it is being made (or not).

Parameters

- **map** (array or vegas.AdaptiveMap or vegas.Integrator) – The integration region as specified by an array map[d, i] where d is the direction and i=0,1 specify the lower and upper limits of integration in direction d.

  map could also be the integration map from another vegas.Integrator, or that vegas.Integrator itself. In this case the grid is copied from the existing integrator.

- **nitn** (*positive int*) – The maximum number of iterations used to adapt to the integrand and estimate its value. The default value is 10; typical values range from 10 to 20.

- **neval** (*positive int*) – The maximum number of integrand evaluations in each iteration of the vegas algorithm. Increasing neval increases the precision: statistical errors should fall at least as fast as sqrt(1./neval) and often fall much faster. The default value is 1000; real problems often require 10–1000 times more evaluations than this.

- **alpha** (*float*) – Damping parameter controlling the remapping of the integration variables as vegas adapts to the integrand. Smaller values slow adaptation, which may be desirable for difficult integrands. Small or zero alphas are also sometimes useful after the grid has adapted, to minimize fluctuations away from the optimal grid. The default value is 0.5.

- **beta** (*float*) – Damping parameter controlling the redistribution of integrand evaluations across hypercubes in the stratified sampling of the integral (over transformed variables). Smaller values limit the amount of redistribution. The theoretically optimal value is 1; setting beta=0 prevents any redistribution of evaluations. The default value is 0.75.

- **adapt** (*bool*) – Setting adapt=False prevents further adaptation by vegas. Typically this would be done after training the vegas.Integrator on an integrand, in order to stabilize further estimates of the integral. vegas uses unweighted averages to combine results from different iterations when adapt=False. The default setting is adapt=True.

- **nhcube_batch** (*positive int*) – The number of hypercubes (in y space) whose integration points are combined into a single batch to be passed to the integrand, together, when using vegas in batch mode. The default value is 1000. Larger values may be lead to faster evaluations, but at the cost of more memory for internal work arrays.

- **minimize_mem** (*bool*) – When True, vegas minimizes internal workspace at the cost of extra evaluations of the integrand. This can increase execution time by 50–100% but might be desirable when the number of evaluations is very large (e.g., neval=1e9). Normally vegas uses internal work space that grows in proportion to neval. If that work space exceeds the size of the RAM available to the processor, vegas runs much more slowly. Setting minimize_mem=True greatly reduces the internal storage used by vegas; in particular memory becomes independent of neval. The default setting (minimize_mem=False), however, is much superior unless memory becomes a problem. (The large memory is needed for adaptive stratified sampling, so memory is not an issue if beta=0.)

- **adapt_to_errors** (*bool*) – adapt_to_errors=False causes vegas to remap the integration variables to emphasize regions where |f(x)| is largest. This is the default mode.

`adapt_to_errors=True` causes `vegas` to remap variables to emphasize regions where the Monte Carlo error is largest. This might be superior when the number of the number of stratifications (`self.nstrat`) in the y grid is large (> 50?). It is typically useful only in one or two dimensions.

- **maxinc_axis** (*positive int*) – The maximum number of increments per axis allowed for the x-space grid. The default value is 1000; there is probably little need to use other values.
- **max_nhcube** (*positive int*) – Maximum number of y-space hypercubes used for stratified sampling. Setting `max_nhcube=1` turns stratified sampling off, which is probably never a good idea. The default setting (1e9) was chosen to correspond to the point where internal work arrays become comparable in size to the typical amount of RAM available to a processor (in a laptop in 2014). Internal memory usage is large only when `beta>0` and `minimize_mem=False`; therefore `max_nhcube` is ignored if `beta=0` or `minimize_mem=True`.
- **max_neval_hcube** (*positive int*) – Maximum number of integrand evaluations per hypercube in the stratification. The default value is 1e7. Larger values might allow for more adaptation (when `neval` is larger than `2 * max_neval_hcube`), but also can result in very large internal work arrays.
- **rtol** (*float less than 1*) – Relative error in the integral estimate at which point the integrator can stop. The default value is 0.0 which means that the integrator will complete all iterations specified by `nitn`.
- **atol** (*float*) – Absolute error in the integral estimate at which point the integrator can stop. The default value is 0.0 which means that the integrator will complete all iterations specified by `nitn`.
- **analyzer** – An object with methods

  ```
  analyzer.begin(itn, integrator)

  analyzer.end(itn_result, result)
  ```

  where: `begin(itn, integrator)` is called at the start of each `vegas` iteration with `itn` equal to the iteration number and `integrator` equal to the integrator itself; and `end(itn_result, result)` is called at the end of each iteration with `itn_result` equal to the result for that iteration and `result` equal to the cummulative result of all iterations so far. Setting `analyzer=vegas.reporter()`, for example, causes vegas to print out a running report of its results as they are produced. The default is `analyzer=None`.
- **ran_array_generator** – Function that generates `numpy` arrays of random numbers distributed uniformly between 0 and 1. `ran_array_generator(shape)` should create an array whose dimensions are specified by the integer-valued tuple `shape`. The default generator is `numpy.random.random`.

`vegas.Integrator` objects have attributes for each of these parameters. In addition they have the following methods:

__call__ (*fcn*, ***kargs*)
Integrate integrand `fcn`.

A typical integrand has the form, for example:

```
def f(x):
    return x[0] ** 2 + x[1] ** 4
```

The argument `x[d]` is an integration point, where index `d=0...` represents direction within the integration volume.

Integrands can be array-valued, representing multiple integrands: e.g.,

```
def f(x):
    return [x[0] ** 2, x[0] / x[1]]
```

The return arrays can have any shape. Array-valued integrands are useful for integrands that are closely related, and can lead to substantial reductions in the errors for ratios or differences of the results.

It is usually much faster to use `vegas` in batch mode, where integration points are presented to the integrand in batches. A simple batch integrand might be, for example:

```
@vegas.batchintegrand
def f(x):
    return x[:, 0] ** 2 + x[:, 1] ** 4
```

where decorator `@vegas.batchintegrand` tells `vegas` that the integrand processes integration points in batches. The array `x[i, d]` represents a collection of different integration points labeled by `i=0...`. (The number of points is controlled `vegas.Integrator` parameter `nhcube_batch`.) The batch index is always first.

Batch integrands can also be constructed from classes derived from `vegas.BatchIntegrand`.

Batch mode is particularly useful (and fast) when the class derived from `vegas.BatchIntegrand` is coded in Cython. Then loops over the integration points can be coded explicitly, avoiding the need to use `numpy`'s whole-array operators if they are not well suited to the integrand.

Any `vegas` parameter can also be reset: e.g., `self(fcn, nitn=20, neval=1e6)`.

set (*ka={}*, ***kargs*)
Reset default parameters in integrator.

Usage is analogous to the constructor for `vegas.Integrator`: for example,

```
old_defaults = integ.set(neval=1e6, nitn=20)
```

resets the default values for `neval` and `nitn` in `vegas.Integrator` `integ`. A dictionary, here `old_defaults`, is returned. It can be used to restore the old defaults using, for example:

```
integ.set(old_defaults)
```

settings (*ngrid=0*)
Assemble summary of integrator settings into string.

Parameters **ngrid** (*int*) – Number of grid nodes in each direction to include in summary. The default is 0.

Returns String containing the settings.

random (*yield_hcube=False*, *yield_y=False*)
Iterator over integration points and weights.

This method creates an iterator that returns integration points from `vegas`, and their corresponding weights in an integral. Each point `x[d]` is accompanied by the weight assigned to that point by `vegas` when estimating an integral. Optionally it will also return the index of the hypercube containing the integration point and/or the y-space coordinates:

```
integ.random()  yields  x, wgt

integ.random(yield_hcube=True) yields x, wgt, hcube

integ.random(yield_y=True) yields x, y, wgt

integ.random(yield_hcube=True, yield_y=True) yields x, y, wgt, hcube
```

The number of integration points returned by the iterator corresponds to a single iteration.

random_batch (*yield_hcube=False*, *yield_y=False*)
Iterator over integration points and weights.

This method creates an iterator that returns integration points from `vegas`, and their corresponding weights in an integral. The points are provided in arrays `x[i, d]` where `i=0...` labels the integration points in a batch and `d=0...` labels direction. The corresponding weights assigned by `vegas` to each point are provided in an array `wgt[i]`.

Optionally the integrator will also return the indices of the hypercubes containing the integration points and/or the y-space coordinates of those points:

```
integ.random()  yields  x, wgt

integ.random(yield_hcube=True) yields x, wgt, hcube

integ.random(yield_y=True) yields x, y, wgt

integ.random(yield_hcube=True, yield_y=True) yields x, y, wgt, hcube
```

The number of integration points returned by the iterator corresponds to a single iteration. The number in a batch is controlled by parameter `nhcube_batch`.

3.3 AdaptiveMap Objects

`vegas`'s remapping of the integration variables is handled by a `vegas.AdaptiveMap` object, which maps the original integration variables x into new variables y in a unit hypercube. Each direction has its own map specified by a grid in x space:

$$\begin{aligned} x_0 &= a \\ x_1 &= x_0 + \Delta x_0 \\ x_2 &= x_1 + \Delta x_1 \\ &\cdots \\ x_N &= x_{N-1} + \Delta x_{N-1} = b \end{aligned}$$

where a and b are the limits of integration. The grid specifies the transformation function at the points $y = i/N$ for $i = 0, 1 \ldots N$:

$$x(y{=}i/N) = x_i$$

Linear interpolation is used between those points. The Jacobian for this transformation is:

$$J(y) = J_i = N\Delta x_i$$

`vegas` adjusts the increments sizes to optimize its Monte Carlo estimates of the integral. This involves training the grid. To illustrate how this is done with `vegas.AdaptiveMap`s consider a simple two dimensional integral over a unit hypercube with integrand:

```
def f(x):
   return x[0] * x[1] ** 2
```

We want to create a grid that optimizes uniform Monte Carlo estimates of the integral in y space. We do this by sampling the integrand at a large number `ny` of random points `y[j, d]`, where `j=0...ny-1` and `d=0,1`, uniformly distributed throughout the integration volume in y space. These samples be used to train the grid using the following code:

```
import vegas
import numpy as np

def f(x):
   return x[0] * x[1] ** 2

m = vegas.AdaptiveMap([[0, 1], [0, 1]], ninc=5)

ny = 1000
y = np.random.uniform(0., 1., (ny, 2))  # 1000 random y's

x = np.empty(y.shape, float)            # work space
jac = np.empty(y.shape[0], float)
f2 = np.empty(y.shape[0], float)

print('intial grid:')
print(m.settings())

for itn in range(5):                    # 5 iterations to adapt
   m.map(y, x, jac)                     # compute x's and jac

   for j in range(ny):                  # compute training data
      f2[j] = (jac[j] * f(x[j])) ** 2

   m.add_training_data(y, f2)           # adapt
   m.adapt(alpha=1.5)

   print('iteration %d:' % itn)
   print(m.settings())
```

In each of the 5 iterations, the `vegas.AdaptiveMap` adjusts the map, making increments smaller where `f2` is larger and larger where `f2` is smaller. The map converges after only 2 or 3 iterations, as is clear from the output:

```
initial grid:
    grid[ 0] = [ 0.   0.2  0.4  0.6  0.8  1. ]
    grid[ 1] = [ 0.   0.2  0.4  0.6  0.8  1. ]

iteration 0:
    grid[ 0] = [ 0.     0.412  0.62   0.76   0.883  1.   ]
    grid[ 1] = [ 0.     0.506  0.691  0.821  0.91   1.   ]

iteration 1:
    grid[ 0] = [ 0.     0.428  0.63   0.772  0.893  1.   ]
    grid[ 1] = [ 0.     0.531  0.713  0.832  0.921  1.   ]

iteration 2:
    grid[ 0] = [ 0.     0.433  0.63   0.772  0.894  1.   ]
    grid[ 1] = [ 0.     0.533  0.714  0.831  0.922  1.   ]

iteration 3:
```

```
    grid[ 0] = [ 0.     0.435  0.631  0.772  0.894  1.   ]
    grid[ 1] = [ 0.     0.533  0.715  0.831  0.923  1.   ]

iteration 4:
    grid[ 0] = [ 0.     0.436  0.631  0.772  0.895  1.   ]
    grid[ 1] = [ 0.     0.533  0.715  0.831  0.924  1.   ]
```

The grid increments along direction 0 shrink at larger values `x[0]`, varying as `1/x[0]`. Along direction 1 the increments shrink more quickly varying like `1/x[1]**2`.

`vegas` samples the integrand in order to estimate the integral. It uses those same samples to train its `vegas.AdaptiveMap` in this fashion, for use in subsequent iterations of the algorithm.

class `vegas.`**`AdaptiveMap`**

Adaptive map `y->x(y)` for multidimensional `y` and `x`.

An `AdaptiveMap` defines a multidimensional map `y -> x(y)` from the unit hypercube, with `0 <= y[d] <= 1`, to an arbitrary hypercube in `x` space. Each direction is mapped independently with a Jacobian that is tunable (i.e., "adaptive").

The map is specified by a grid in `x`-space that, by definition, maps into a uniformly spaced grid in `y`-space. The nodes of the grid are specified by `grid[d, i]` where d is the direction (`d=0,1...dim-1`) and `i` labels the grid point (`i=0,1...N`). The mapping for a specific point `y` into `x` space is:

```
y[d] -> x[d] = grid[d, i(y[d])] + inc[d, i(y[d])] * delta(y[d])
```

where `i(y)=floor(y*N)`, `delta(y)=y*N - i(y)`, and `inc[d, i] = grid[d, i+1] - grid[d, i]`. The Jacobian for this map,

```
dx[d]/dy[d] = inc[d, i(y[d])] * N,
```

is piece-wise constant and proportional to the `x`-space grid spacing. Each increment in the `x`-space grid maps into an increment of size `1/N` in the corresponding `y` space. So regions in `x` space where `inc[d, i]` is small are stretched out in `y` space, while larger increments are compressed.

The `x` grid for an `AdaptiveMap` can be specified explicitly when the map is created: for example,

```
m = AdaptiveMap([[0, 0.1, 1], [-1, 0, 1]])
```

creates a two-dimensional map where the `x[0]` interval `(0,0.1)` and `(0.1,1)` map into the `y[0]` intervals `(0,0.5)` and `(0.5,1)` respectively, while `x[1]` intervals `(-1,0)` and `(0,1)` map into `y[1]` intervals `(0,0.5)` and `(0.5,1)`.

More typically an initially uniform map is trained with data `f[j]` corresponding to `ny` points `y[j, d]`, with `j=0...ny-1`, uniformly distributed in y space: for example,

```
m.add_training_data(y, f)
m.adapt(alpha=1.5)
```

`m.adapt(alpha=1.5)` shrinks grid increments where `f[j]` is large, and expands them where `f[j]` is small. Typically one has to iterate over several sets of `y`s and `f`s before the grid has fully adapted.

The speed with which the grid adapts is determined by parameter `alpha`. Large (positive) values imply rapid adaptation, while small values (much less than one) imply slow adaptation. As in any iterative process, it is usually a good idea to slow adaptation down in order to avoid instabilities.

Parameters

- **grid** – Initial `x` grid, where `grid[d, i]` is the `i`-th node in direction `d`.
- **ninc** (`int` or `None`) – Number of increments along each axis of the `x` grid. A new grid is generated if `ninc` differs from `grid.shape[1]`. The new grid is designed to give the

same Jacobian `dx(y)/dy` as the original grid. The default value, `ninc=None`, leaves the grid unchanged.

dim
Number of dimensions.

ninc
Number of increments along each grid axis.

grid
The nodes of the grid defining the maps are `self.grid[d, i]` where `d=0...` specifies the direction and `i=0...self.ninc` the node.

inc
The increment widths of the grid:

```
self.inc[d, i] = self.grid[d, i + 1] - self.grid[d, i]
```

adapt(*alpha=0.0*, *ninc=None*)
Adapt grid to accumulated training data.

`self.adapt(...)` projects the training data onto each axis independently and maps it into `x` space. It shrinks `x`-grid increments in regions where the projected training data is large, and grows increments where the projected data is small. The grid along any direction is unchanged if the training data is constant along that direction.

The number of increments along a direction can be changed by setting parameter `ninc`.

The grid does not change if no training data has been accumulated, unless `ninc` is specified, in which case the number of increments is adjusted while preserving the relative density of increments at different values of `x`.

Parameters

- **alpha** (*double or None*) – Determines the speed with which the grid adapts to training data. Large (postive) values imply rapid evolution; small values (much less than one) imply slow evolution. Typical values are of order one. Choosing `alpha<0` causes adaptation to the unmodified training data (usually not a good idea).
- **ninc** (*int or None*) – Number of increments along each direction in the new grid. The number is unchanged from the old grid if `ninc` is omitted (or equals `None`).

add_training_data(*y*, *f*, *ny=-1*)
Add training data `f` for `y`-space points `y`.

Accumulates training data for later use by `self.adapt()`. Grid increments will be made smaller in regions where `f` is larger than average, and larger where `f` is smaller than average. The grid is unchanged (converged?) when `f` is constant across the grid.

Parameters

- **y** (*contiguous 2-d array of floats*) – `y` values corresponding to the training data. `y` is a contiguous 2-d array, where `y[j, d]` is for points along direction `d`.
- **f** (*contiguous 2-d array of floats*) – Training function values. `f[j]` corresponds to point `y[j, d]` in `y`-space.
- **ny** (*int*) – Number of `y` points: `y[j, d]` for `d=0...dim-1` and `j=0...ny-1`. `ny` is set to `y.shape[0]` if it is omitted (or negative).

__call__(*y*)
Return `x` values corresponding to `y`.

y can be a single dim-dimensional point, or it can be an array y[i,j, ..., d] of such points (d=0..dim-1).

jac(*y*)
Return the map's Jacobian at y.

y can be a single dim-dimensional point, or it can be an array y[d,i,j,...] of such points (d=0..dim-1).

make_uniform(*ninc=None*)
Replace the grid with a uniform grid.

The new grid has ninc increments along each direction if ninc is specified. Otherwise it has the same number of increments as the old grid.

map(*y*, *x*, *jac*, *ny=-1*)
Map y to x, where jac is the Jacobian.

y[j, d] is an array of ny y-values for direction d. x[j, d] is filled with the corresponding x values, and jac[j] is filled with the corresponding Jacobian values. x and jac must be preallocated: for example,

```
x = numpy.empty(y.shape, float)
jac = numpy.empty(y.shape[0], float)
```

Parameters

- **y** (*contiguous 2-d array of floats*) – y values to be mapped. y is a contiguous 2-d array, where y[j, d] contains values for points along direction d.
- **x** (*contiguous 2-d array of floats*) – Container for x values corresponding to y.
- **jac** (*contiguous 1-d array of floats*) – Container for Jacobian values corresponding to y.
- **ny** (*int*) – Number of y points: y[j, d] for d=0...dim-1 and j=0...ny-1. ny is set to y.shape[0] if it is omitted (or negative).

show_grid(*ngrid=40*, *shrink=False*)
Display plots showing the current grid.

Parameters

- **ngrid** (*int*) – The number of grid nodes in each direction to include in the plot. The default is 40.
- **shrink** – Display entire range of each axis if False; otherwise shrink range to include just the nodes being displayed. The default is False.

Nparam axes List of pairs of directions to use in different views of the grid. Using None in place of a direction plots the grid for only one direction. Omitting axes causes a default set of pairings to be used.

settings(*ngrid=5*)
Create string with information about grid nodes.

Creates a string containing the locations of the nodes in the map grid for each direction. Parameter ngrid specifies the maximum number of nodes to print (spread evenly over the grid).

3.4 Other Objects and Functions

class vegas.**RAvg**

Running average of Monte Carlo estimates.

This class accumulates independent Monte Carlo estimates (e.g., of an integral) and combines them into a single average. It is derived from gvar.GVar (from the gvar module if it is present) and represents a Gaussian random variable.

Different estimates are weighted by their inverse variances if parameter weight=True; otherwise straight, unweighted averages are used.

mean

The mean value of the weighted average.

sdev

The standard deviation of the weighted average.

chi2

*chi**2* of weighted average.

dof

Number of degrees of freedom in weighted average.

Q

Q or *p-value* of weighted average's *chi**2*.

itn_results

A list of the results from each iteration.

add(*g*)

Add estimate g to the running average.

summary()

Assemble summary of independent results into a string.

class vegas.**RAvgArray**

Running average of array-valued Monte Carlo estimates.

This class accumulates independent arrays of Monte Carlo estimates (e.g., of an integral) and combines them into an array of averages. It is derived from numpy.ndarray. The array elements are gvar.GVars (from the gvar module if present) and represent Gaussian random variables.

Different estimates are weighted by their inverse covariance matrices if parameter weight=True; otherwise straight, unweighted averages are used.

chi2

*chi**2* of weighted average.

dof

Number of degrees of freedom in weighted average.

Q

Q or *p-value* of weighted average's *chi**2*.

itn_results

A list of the results from each iteration.

add(*g*)

Add estimate g to the running average.

summary()

Assemble summary of independent results into a string.

vegas.**batchintegrand**()
Decorator for batch integrand functions.

Applying vegas.batchintegrand() to a function fcn repackages the function in a format that vegas can understand. Appropriate functions take a numpy array of integration points x[i, d] as an argument, where i=0... labels the integration point and d=0... labels direction, and return an array f[i] of integrand values (or arrays of integrand values) for the corresponding points. The meaning of fcn(x) is unchanged by the decorator, but the type of fcn is changed.

An example is

```
import vegas
import numpy as np

@vegas.batchintegrand
def f(x):
    return np.exp(-x[:, 0] - x[:, 1])
```

for the two-dimensional integrand $\exp(-x_0 - x_1)$.

This decorator provides an alternative to deriving an integrand class from vegas.BatchIntegrand.

class vegas.**BatchIntegrand**
Base class for classes providing batch integrands.

A class derived from vegas.BatchIntegrand will normally provide a __call__(self, x) method that returns an array f where:

x[i, d] is a contiguous numpy array where i=0... labels different integrtion points and d=0... labels different directions in the integration space.

f[i] is a contiguous array containing the integrand values corresponding to the integration points x[i, :]. f[i] is either a number, for a single integrand, or an array (of any shape) for multiple integrands (i.e., an array-valued integrand).

An example is

```
import vegas
import numpy as np

class batchf(vegas.BatchIntegrand):
    def __call__(x):
        return np.exp(-x[:, 0] - x[:, 1])

f = batchf()       # the integrand
```

for the two-dimensional integrand $\exp(-x_0 - x_1)$.

Deriving from vegas.BatchIntegrand is the easiest way to construct integrands in Cython, and gives the fastest results.

class vegas.**MPIintegrand**
Convert (batch) integrand into an MPI multiprocessor integrand.

Applying decorator vegas.MPIintegrand to a function repackages the function as a batch vegas integrand that can execute in parallel on multiple processors. Appropriate functions take a numpy array of integration points x[i, d] as an argument, where i=0... labels the integration point and d=0... labels direction, and return an array f[i] of integrand values (or arrays f[i,...] of integrand values) for the corresponding points.

An example is

```
import vegas
import numpy as np

@vegas.MPIintegrand
def f(x):
    return np.exp(-x[:, 0] - x[:, 1])
```

for the two-dimensional integrand $\exp(-x_0 - x_1)$. Of course, one could write `f = vegas.MPIintegrand(f)` instead of using the decorator.

Message passing between processors uses MPI via Python module `mpi4py`, which must be installed in Python. To run an MPI integration code `mpi-integral.py` on 4 processors, for example, one might execute:

```
mpirun -np 4 python mpi-integral.py
```

Executing `python mpi-integral.py`, without the `mpirun`, causes it to run on a single processor, in more or less the same way an integral with a batch integrand runs.

An object of type `vegas.MPIintegrand` contains information about the MPI processes in the following attributes:

comm
: MPI intracommunicator — `mpi4py.MPI.Intracomm` object `mpi4py.MPI.COMM_WORLD`.

nproc
: Number of processors used.

rank
: MPI rank of current process. Each process has a unique rank, ranging from `0` to `nproc-1`. The rank is used to make different processes do different things (for example, one generally wants only one of the processes to report out final results).

seed
: The random number see used to reset `numpy.random.random` in all the processes.

The implementation used here has the entire integration code run on every processor. It is only when evaluating the integrand that the processes do different things. This is efficient provided most of the time is spent evaluating the integrand, which, in any case, is the only situation where it might make sense to use multiple processors.

Note that `vegas.MPIintegrand` assumes that `vegas.Integrator` is using the default random number generator (`numpy.random.random`). If this is not the case, it is important to seed the other random number generator so that all processes use the same random numbers.

The approach used here to make `vegas` parallel is based on a strategy used by R. Horgan and Q. Mason with the original Fortran version of `vegas`.

CHAPTER

FOUR

INDICES AND TABLES

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

V

Symbols

A

B

C

D

G

I

J

M

N

Q

R

S

V