vegas Documentation

Release 1.0

G.P. Lepage

December 17, 2013

CONTENTS

1 Overview and Tutorial **3**
- 1.1 Introduction . . . 3
- 1.2 Basic Integrals . . . 4
- 1.3 Faster Integrands . . . 9

2 How vegas Works **13**
- 2.1 The vegas Grid: Importance Sampling . . . 13
- 2.2 Adaptive Stratified Sampling . . . 15

3 The vegas Package **19**
- 3.1 Introduction . . . 19
- 3.2 Integrator Objects . . . 19
- 3.3 AdaptiveMap Objects . . . 21
- 3.4 Other Objects . . . 24

4 Indices and tables **27**

Python Module Index **29**

Index **31**

Contents:

CHAPTER

ONE

OVERVIEW AND 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 beyond square-integrability — 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. For example, 90% of the 1-dimensional integral

$$\int_0^1 dx\, \mathrm{e}^{-100(x-0.5)^2}$$

comes from about 23% of the integration volume. In 20 dimensions, 90% of the analogous integral,

$$\int_0^1 dx_1 \cdots \int_0^1 dx_{20}\ \mathrm{e}^{-100\sum_\mu (x_\mu-0.5)^2},$$

comes from only 10^{-8} of the total integration volume. Non-adaptive algorithms would have a very hard time noticing that there was a peak at all. *vegas* has no trouble with this integral.

Monte Carlo integration also provides efficient and reliable methods for 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 one generates 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 may 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. This module is written in Dython, so it is almost as fast as optimized Fortran or C, particularly when the integrand is also coded in Cython (or some other compiled language), as discussed below.

1.2 Basic Integrals

Here we illustrate `vegas` by estimating the integral

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

where constant C is chosen so that the exact value 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` 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 worth noting 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. The integrand has a large narrow peak in the center of the integration volume. Most of `vegas`'s integrand samples miss the peak in early iterations, but it uses information from the samples in one iteration to remap the integration variables for subsequent iterations, concentrating samples where the function is largest. As

a result, the per iteration error is reduced to 3.4% by the fifth iteration, and below 2% by the end — an improvement by almost two orders of magnitude from the start.

Weighted Average: The final result, 1.0015 ± 0.0091, is obtained from a weighted average of the separate results from each iteration. Provided the number of integrand evaluations per iteration (`neval`) is large enough, the results from individual iterations are random numbers whose distribution is Gaussian, with a mean equal to the integral's value. 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, here the number of iterations minus 1. The error estimates are not reliable 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.RunningWAvg`, 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` — nuumber 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.

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 of `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 one wants 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 that use 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, 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 is guaranteed to improve the Monte Carlo estimate, with the systematic error vanishing quickly. 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 will no longer mask the systematic error (which is affected by `neval` but not `nitn`). The systematic error for the integral above (with `neval=1000`) is about -0.00073(7), 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 can be 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   0.018(11)       0.0159(82)          0.13     0.72
  3   1.74(80)        0.0161(82)          2.36     0.09
  4   0.83(20)        0.0174(82)          6.97     0.00
  5   0.934(87)       0.0255(82)         32.60     0.00
  6   0.905(53)       0.0463(81)         80.46     0.00
  7   1.010(42)       0.0805(80)        150.57     0.00
  8   0.964(30)       0.1385(77)        244.64     0.00
  9   1.023(29)       0.1985(74)        326.07     0.00
 10   0.987(22)       0.2777(70)        415.67     0.00

result = 0.2777(70)    Q = 0.00
```

`vegas` misses the peak completely in the first two iterations, giving estimates that are completely wrong (by 76 and 89 standard deviations!). Some of its samples hit the peak's shoulders, so `vegas` is eventually able to find it (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 results from the second step are well adapted from the start, and the final result is good:

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   1.015(27)       1.015(27)           0.00     1.00
  2   1.024(24)       1.020(18)           0.06     0.80
  3   0.991(15)       1.003(12)           0.81     0.44
  4   0.989(17)       0.9989(97)          0.70     0.55
  5   1.002(16)       0.9998(83)          0.53     0.71
  6   1.019(18)       1.0030(76)          0.60     0.70
  7   1.016(16)       1.0053(69)          0.59     0.74
  8   0.988(16)       1.0028(63)          0.63     0.73
  9   0.978(15)       0.9990(58)          0.84     0.57
 10   1.004(14)       0.9997(54)          0.75     0.66

result = 0.9997(54)    Q = 0.66
```

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)
```

gives the following new output:

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   0.5089(72)      0.5089(72)          0.00     1.00
  2   0.5001(70)      0.5044(50)          0.76     0.38
  3   0.4955(66)      0.5011(40)          0.95     0.39
  4   0.4960(68)      0.4998(35)          0.77     0.51
  5   0.5128(79)      0.5019(32)          1.14     0.34
  6   0.5038(69)      0.5022(29)          0.92     0.46
  7   0.5025(71)      0.5023(27)          0.77     0.59
  8   0.4885(72)      0.5006(25)          1.12     0.35
  9   0.4933(65)      0.4997(23)          1.11     0.35
 10   0.500(15)       0.4997(23)          0.99     0.44

result = 0.4997(23)    Q = 0.44
```

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 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 by the same Gaussian, but restricted to a 4-sphere of radius 0.2, centered on the Gaussian:

```
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))
    integ
```

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

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   1.057(81)       1.057(81)           0.00     1.00
  2   0.984(34)       0.995(31)           0.69     0.41
  3   1.001(39)       0.997(24)           0.35     0.70
  4   1.003(32)       0.999(19)           0.24     0.87
  5   0.974(25)       0.990(15)           0.34     0.85
  6   0.973(34)       0.987(14)           0.31     0.91
  7   1.65(46)        0.987(14)           0.60     0.73
  8   1.049(60)       0.991(14)           0.65     0.71
  9   1.049(83)       0.992(13)           0.63     0.75
 10   1.055(51)       0.996(13)           0.72     0.69

result = 0.996(13)    Q = 0.69
```

This result 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 `alpha`s giving slower adaptation. Here we reduce it to 0.1, from its default value of 0.5, and get the following output:

```
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   1.026(23)       1.026(23)           0.00     1.00
  2   0.968(22)       0.995(16)           3.38     0.07
  3   1.039(23)       1.009(13)           2.89     0.06
  4   0.991(22)       1.004(11)           2.09     0.10
  5   1.022(26)       1.007(10)           1.67     0.15
  6   0.964(22)       0.9995(94)          1.96     0.08
  7   0.992(19)       0.9980(84)          1.65     0.13
  8   1.007(22)       0.9991(79)          1.44     0.19
  9   1.002(22)       0.9995(74)          1.26     0.26
 10   0.969(18)       0.9952(68)          1.38     0.19

result = 0.9952(68)    Q = 0.19
```

Notice how the errors fluctuate less from iteration to iteration with the smaller `alpha`. `vegas` finds

and holds onto the edge of the actual integration volume (at radius 0.2) more effectively when it is less precipitous about adapting. This leads to better results in this case.

It is a good idea to make the actual integration volume as large a fraction as possible of the total volume used by `vegas`, so `vegas` doesn't spend lots of effort on regions where the integrand is exactly 0. Also, it can be challenging 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.

Note, finally, that integration to infinity is also possible by mapping the relevant variable into a different variable of finite range — for example, by changing an integral over $x \equiv \tan(\theta)$ from 0 to infinity into one over θ from 0 to $\pi/2$.

1.3 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. Realistic problems, however, can require hundreds of thousands or millions of function evaluations, or more.

The cost of evaluating the integrand can be reduced significantly by vectorizing it, if that is possible. For example, replacing

```
import vegas

dim = 4
norm = 1013.2118364296088

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

by

```
import vegas
import numpy as np

dim = 4
norm = 1013.2118364296088

class f_vector(vegas.VecIntegrand):
    def __init__(self, dim):
        self.dim = dim

    def __call__(self, xx, ff, nx):
        # convert integration points xx[i, d] to numpy array
        x = np.asarray(xx)[:nx, :]

        # convert array for answer into a numpy array
```

```
        f = np.asarray(ff)

        # evaluate integrand for all values of i simultaneously
        dx2 = 0.0
        for d in range(self.dim):
            dx2 += (x[:, d] - 0.5) ** 2
        f[:nx] = np.exp(-100. * dx2) * norm

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

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

reduces the cost of the integral by about an order of magnitude. An instance of class `f_vector` behaves like a function of three variables:

`xx[i, d]` — integration points for each `i=0...nx-1`;

`ff[i]` — buffer to hold the integrand values for each integration point;

`nx` — number of integration points.

We derive class `f_vector` from `vegas.VecIntegrand` to signal to `vegas` that it should present integration points in batches to the integrand function. Parameter `nhcube_vec` tells `vegas` how many hypercubes to put in a batch; the bigger this parameter is, the larger the vectors.

Unfortunately many realistic problems are difficult to vectorize. The fastest option in such cases (and actually every case) is to write the integrand in Cython, which is a compiled hybrid of Python and C. The Cython version of this code, which we put in a separate file we call `cython_integrand.pyx`, is simpler than the vector version:

```
cimport vegas
from libc.math cimport exp

import vegas

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

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

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

The main code is then

```
import pyximport; pyximport.install()

import vegas
```

```
from cython_integrand import f_cython

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

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

where 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.

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

CHAPTER

TWO

HOW VEGAS WORKS

2.1 The vegas Grid: 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. In one dimension, for example, vegas converts

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

into

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

and then does a Monte Carlo estimate in y space. The transformation function $x(y)$ is chosen so that Jacobian $J(y) \propto 1/|f(x)|$. This choice minimizes the statistical errors of the y-space Monte Carlo estimate of integral: it flattens the integrand, removing peaks by spreading them out in y space.

vegas implements this transformation 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.

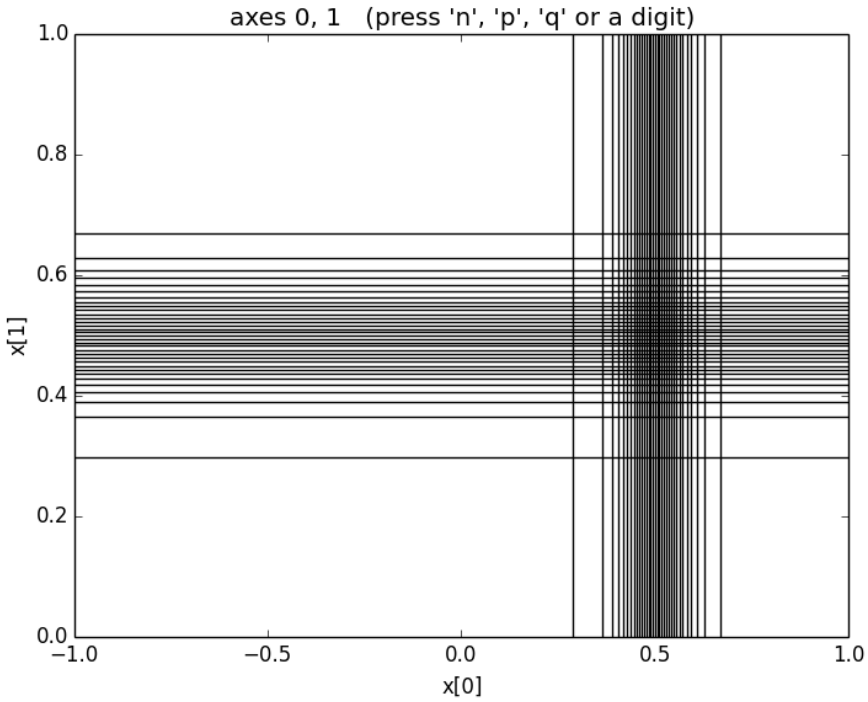
The Jacobian for this transformation function is piecewise constant:

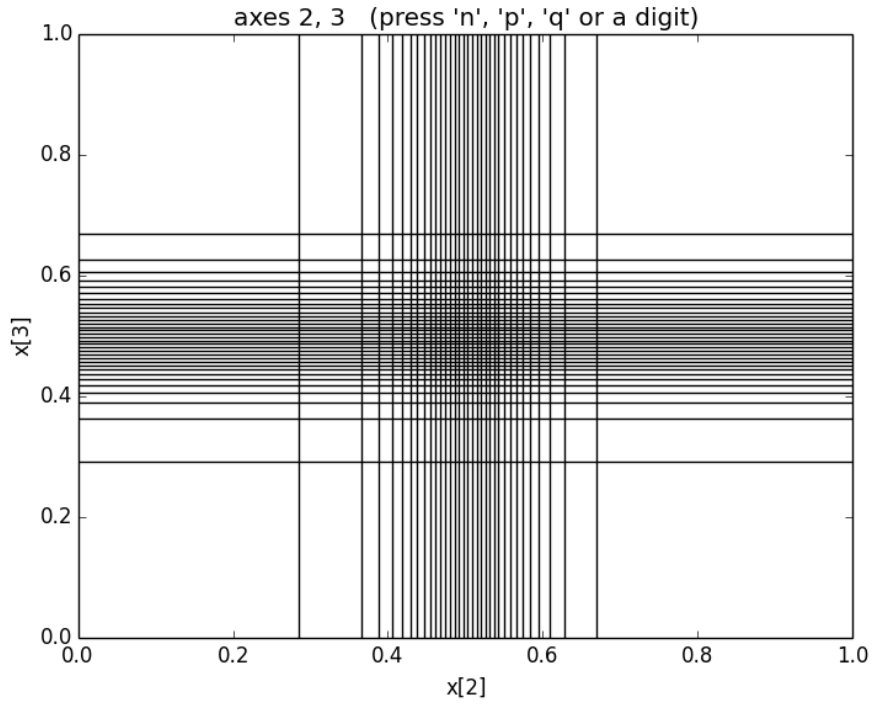
$$J(y) = N\Delta x_i$$

for $i/N < y < (i+1)/N$. Consequently the optimal grid is one where the grid increments Δx_i are small in regions where $|f(x)|$ is large. These small increments map into proportionally larger increments in y space. This causes the Monte Carlo integration to focus its sampling of the integrand in regions where the integrand is large — a standard Monte Carlo technique called "importance sampling."

`vegas` typically starts with no knowledge of the integrand and so starts with a uniform x grid. As it samples the integrand it gathers information about where the peaks are, and refines its choice of Δx_is for use in subsequent iterations. The grid 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 plots were obtained by including the line

```
integ.map.plot_grid(30)
```

in the integration code after the integration is finished. It causes `matplotlib` (if it is installed) to create images showing 30 nodes (out of the 99 actually used) of the grid in each direction. Obviously `vegas` is focusing its resources on the region around `x = [0.5, 0.5, 0.5, 0.5]`.

2.2 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:

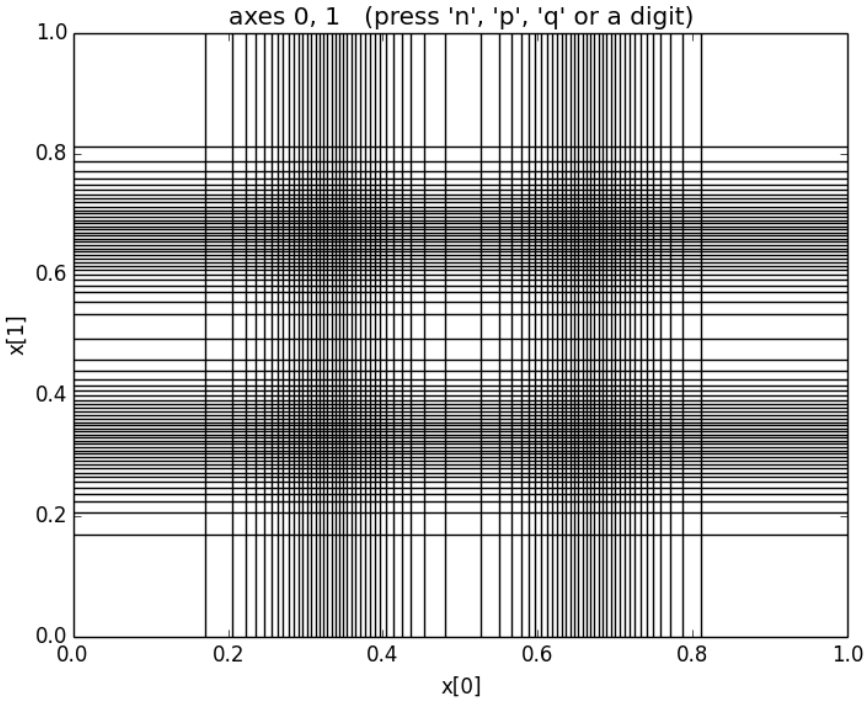
```
def f2(x):
    dx2 = 0
    for i in range(4):
        dx2 += (x[i] - 1/3.) ** 2
    ans = math.exp(-dx2 * 100.) * 1013.2167575422921535
    dx2 = 0
    for i in range(4):
        dx2 += (x[i] - 2/3.) ** 2
    ans += math.exp(-dx2 * 100.) * 1013.2167575422921535
    return ans / 2.

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.plot_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. It is a consequence of the fact that 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 lots of realistic applications.

An axis-oriented strategy will always have difficulty adapting to structures that lie along diagonals of the integration hypercube. To address such problems, this 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 hypercubes using a uniform y-space grid with M stratifications on each axis. It estimates the integral by doing a separate Monte Carlo integration in each of the M^d 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^d smaller than neval/2. This restricts M 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 integral estimate from that hypercube — it concentrates integration evaluations where the statistical errors are largest. In the two-Gaussian example above, for example, it 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 new strategy significantly reduces the statistical errors for integrals with large diagonal structures, like the two-Gaussian integral, provided `neval` is large enough to permit a large number M (more than 2 or 3) of stratifications on each axis. For the two-Gaussain integral, the new adaptive strategy (i.e., adaptive stratified sampling) reduces statistical errors by more than a factor of 3 over what older versions of `vegas` give. This is a relatively easy integral; the difference can be more than an order of magnitude for more difficult (and realistic) integrals.

CHAPTER

THREE

THE VEGAS PACKAGE

3.1 Introduction

This package provides tools for estimating multidimensional integrals numerically using an enhanced version of the adaptive Monte Carlo vegas algorithm (G. P. Lepage, J. Comput. Phys. 27(1978) 192). A vegas code generally involves two objects, one representing the integrand and the other representing an integration operator for a particular multidimensional volume. A typical code sequence for a D-dimensional integral has the structure:

```
# create the integrand
def f(x):
    ... compute the integrand at point x[d] d=0,1...D-1
    ...

# create an integrator for volume with
# xl0 <= x[0] <= xu0, xl1 <= x[1] <= xu1 ...
integration_region = [[xl0, xu0], [xl1, xu1], ...]
integrator = vegas.Integrator(integration_region)

# do the integral and print out the result
result = integrator(f, nitn=10, neval=10000)
print(result)
```

The algorithm iteratively adapts to the integrand over `nitn` iterations, each of which uses at most `neval` integrand samples to generate a Monte Carlo estimate of the integral. The final result is the weighted average of the results fom all iterations.

The integrator remembers the adaptations it made to `f(x)` and uses them as its starting point if it is reapplied to `f(x)` or applied to some other function `g(x)`. An integrator's state can be archived for future applications using Python's `pickle` module.

The vegas tutorial and overview contains extended explanations and examples.

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 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 `RunningWAvg` (which is derived from `gvar.GVar`).

Parameters

- **map** (array or `vegas.AdaptiveMap` or `vegas.Integrator`) – The integration region as specified by an array `xlimit[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 :class:Integrator, or the `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.

- **neval** (*positive int*) – The maximum number of integrand evaluations in each iteration of the `vegas` algorithm. Default value is 1000.

- **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. The default value is 0.5.

- **beta** (*float*) – Damping parameter controlling the redistribution of integrand evaluations across hypercubes in the stratified sampling of the integrand (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.

- **nhcube_vec** (*positive int*) – The number of hypercubes (in y space) whose integration points are combined into a single vector to be passed to the integrand when using `vegas` in vectorized mode. The default value is 100.

- **maxinc_axis** (*positive int*) – The maximum number of increments per axis allowed for the x-space grid. The default value is 1000.

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

  `mode=adapt_to_errors` 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 in the y grid is large (> 50?). It is typically useful only in one or two dimensions.

- **fcntype** – Specifies the default type of integrand.

  `fcntype='scalar'` imples that the integrand is a function `f(x)` of a single integration point.

  `fcntype='vector'` implies that the integrand function takes three arguments: a list multiple integration points `x[i, d]`, where `i=0...nx-1` labels the integration point and `d` labels the direction; a buffer `f[i]` into which the corresponding integrand values are written; and the number `nx` of integration points provided.

  The default is `fcntype=scalar`, but this is overridden if the integrand has a `fcntype` attribute. It is also overridden for classes derived from `VecIntegrand`, which are treated

as `fcntype='vector'` integrands.

- **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.

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

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.

3.3 AdaptiveMap Objects

`vegas`'s remapping of the integration variables is handled by `AdaptiveMap`.

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 space 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-1`). The mapping for 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 increments with small `delta_x[i]` are stretched out in `y` space, while larger increments are shrunk.

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

```
map = 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 so that `F(x(y), dx(y)/dy)`, for some training function `F`, is (approximately) constant across `y` space. The training function is assumed to grow monotonically with the jacobian `dx(y)/dy` at fixed `x`. The adaptation is done iteratively, beginning with a uniform map:

```
map = AdaptiveMap([[xl[0], xu[0]], [xl[1], xu[1]]...], ninc=N)
```

which creates an `x` grid with `N` equal-sized increments between `x[d]=xl[d]` and `x[d]=xu[d]`. The training function is then evaluated for the `x` values corresponding to a list of `y` values `y[i, d]` spread over `(0,1)` for each direction `d`:

```
...
for i in range(ny):
    for d in range(dim):
        y[i, d] = ....
x = numpy.empty(y.shape, float)     # container for corresponding x's
jac = numpy.empty(y.shape, float)   # container for corresponding dx/dy's
map.map(y, x, jac)                  # fill x and jac
f = F(x, jac)                       # compute training function
```

The number of `y` points is arbitrary, but typically large. Training data is often generated in batches that are accumulated by the map through multiple calls to:

```
map.add_training_data(y, f)
```

Finally the map is adapted to the data:

```
m.adapt(alpha=1.5)
```

The process of computing training data and then adapting the map typically has to be repeated several times before the map converges, at which point the `x`-grid's nodes, `map.grid[d, i]`, stop changing.

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.

The new grid is designed to make the training function constant in `y[d]` when the `y`s in the other directions are integrated out. 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()`.

Parameters

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

plot_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** – Displays entire range of each maxinc_axis if `False`; otherwise shrink range to include just the nodes being displayed. The default is `False`.

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

class `vegas.`**RunningWAvg**
Running weighted average of Monte Carlo estimates.

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

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.**VecIntegrand**
Base class for classes providing vectorized integrands.

A class derived from `vegas.VecInterand` should provide a `__call__(x, f, nx)` member where:

`x[i, d]` is a contiguous array where `i=0...nx-1` labels different integrtion points and `d=0...` labels different directions in the integration space.

`f[i]` is a buffer that is filled with the integrand values for points `i=0...nx-1`.

`nx` is the number of integration points.

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

CHAPTER

FOUR

INDICES AND TABLES

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

PYTHON MODULE INDEX

V

vegas, 19

INDEX

Symbols

__call__() (vegas.AdaptiveMap method), 23

A

adapt() (vegas.AdaptiveMap method), 23
AdaptiveMap (class in vegas), 21
add() (vegas.RunningWAvg method), 25
add_training_data() (vegas.AdaptiveMap method), 23

C

chi2 (vegas.RunningWAvg attribute), 24

D

dim (vegas.AdaptiveMap attribute), 23
dof (vegas.RunningWAvg attribute), 25

G

grid (vegas.AdaptiveMap attribute), 23

I

inc (vegas.AdaptiveMap attribute), 23
Integrator (class in vegas), 19
itn_results (vegas.RunningWAvg attribute), 25

J

jac() (vegas.AdaptiveMap method), 23

M

make_uniform() (vegas.AdaptiveMap method), 23
map() (vegas.AdaptiveMap method), 24
mean (vegas.RunningWAvg attribute), 24

N

ninc (vegas.AdaptiveMap attribute), 23

P

plot_grid() (vegas.AdaptiveMap method), 24

Q

Q (vegas.RunningWAvg attribute), 25

R

RunningWAvg (class in vegas), 24

S

sdev (vegas.RunningWAvg attribute), 24
set() (vegas.Integrator method), 21
settings() (vegas.AdaptiveMap method), 24
settings() (vegas.Integrator method), 21
summary() (vegas.RunningWAvg method), 25

V

VecIntegrand (class in vegas), 25
vegas (module), 19