scikit-monaco Documentation

Release 0.3-dev

Pascal Bugnion

Feb 22, 2018

Contents

Scikit-monaco is a toolkit for Monte Carlo integration. It is written in Cython for efficiency and includes parallelism to take advantage of multi-core processors.

CHAPTER 1

Tutorials

1.1 Installation

1.1.1 Dependencies

scikit-monaco requires the following:

- Python (tested on 2.7 and 3.3)
- NumPy (tested on 1.8)

scikit-monaco may work with other versions of python and numpy, but these are currently not supported.

You will also need the python development headers and a working C compiler. On Debian-based operating systems such as Ubuntu, you can install the requirements with:

```
sudo apt-get install python-dev python-numpy
```

Additionally, you will need the *nose* package to run the test suite. This can be installed with:

```
sudo apt-get install python-nose
```

1.1.2 Installing using easy_install or pip

The easiest way to install a stable release is using *pip* or *easy_install*. Run either:

```
pip install -U scikit-monaco
```

or:

```
easy_install -U scikit-monaco
```

This will automatically fetch the package from Pypi and install it. If your python installation is system-wide, you will need to run these as root.

1.1.3 Installing from source

Download the source code from the Python package index, extract it, move into the root directory of the project and run the installer:

```
python setup.py install
```

1.1.4 Installing the development version

scikit-monaco is version controlled under git. The repository is hosted on github. Clone the repository using:

```
git clone https://github.com/scikit-monaco/scikit-monaco.git
```

Move to the root of the project's directory and run:

```
python setup.py install
```

To build the documentation, run `python setup.py develop` to build scikit-monaco in the source directory, then `cd doc; make html` to build the html version of the documentation.

1.1.5 Testing

Running `python runtests.py` in the project's root directory will run the test suite.

1.1.6 Benchmarks

There are some benchmarks available. These can be run using:

```
python -c "import skmonaco; skmonaco.bench()"
```

1.2 Getting started

Let's suppose that we want to calculate this integral: $\int_0^1 \int_0^1 x \cdot y \, dxdy$. This could be computed using *mcquad*.

```
>>> from skmonaco import mcquad
>>> mcquad(lambda x_y: x_y[0]*x_y[1], # integrand
...      xl=[0.,0.],xu=[1.,1.], # lower and upper limits of integration
...      npoints=100000 # number of points
...      )
(0.24959359250821114, 0.0006965923631156234)
```

mcquad returns two numbers. The first (0.2496...) is the value of the integral. The second is the estimate in the error (corresponding, roughly, to one standard deviation). Note that the correct answer in this case is $1/4$.

The *mcquad* call distributes points uniformly in a hypercube and sums the value of the integrand over those points. The first argument to *mcquad* is a callable specifying the integrand. This can often be done conveniently using a lambda function. The integrand must take a single argument: a numpy array of length d, where d is the number of dimensions in the integral. For instance, the following would be valid integrands:

```
>>> from math import sin,cos
>>> integrand = lambda x: x**2
>>> integrand = lambda xs: sum(xs)
>>> def integrand(x_y):
...     x,y = x_y
...     return sin(x)*cos(y)
```

If the integrand takes additional parameters, they can be passed to the function through the *args* argument. Suppose that we want to evaluate the product of two Gaussians with exponential factors `alpha` and `beta`:

```
>>> import numpy as np
>>> from skmonaco import mcquad
>>> f = lambda x_y,alpha,beta: np.exp(-alpha*x_y[0]**2)*np.exp(-beta*x_y[1]**2)
>>> alpha = 1.0
>>> beta = 2.0
>>> mcquad(f,xl=[0.,0.],xu=[1.,1.],npoints=100000,args=(alpha,beta))
(0.44650031245386379, 0.00079929285076240579)
```

mcquad runs on a single core as default. The parallel behaviour can be controlled by the *nprocs* parameter. The following can be run in an ipython session to take advantage of their timing routines.

```
>>> from math import sin,cos
>>> from skmonaco import mcquad
>>> f = lambda x,y: sin(x)*cos(y)
>>> %timeit mcquad(f,xl=[0.,0.],xu=[1.,1.],npoints=1e6,nprocs=1)
1 loops, best of 3: 2.26 s per loop
>>> %timeit mcquad(f,xl=[0.,0.],xu=[1.,1.],npoints=1e6,nprocs=2)
1 loops, best of 3: 1.36 s per loop
```

1.3 Complex integration volumes

Monte Carlo integration is very useful when calculating integrals over complicated integration volumes. Consider the integration volume shown in red in this figure:

The integration volume is the intersection of a ring bounded by circles of radius 2 and 3, and a large square.

The volume of integration Ω is given by the following three inequalities:

- $\sqrt{x^2+y^2} \geq 2$: inner circle
- $\sqrt{x^2+y^2} \leq 3$: outer circle
- $x \geq 1$: left border of box
- $y \geq -2$: top border of box

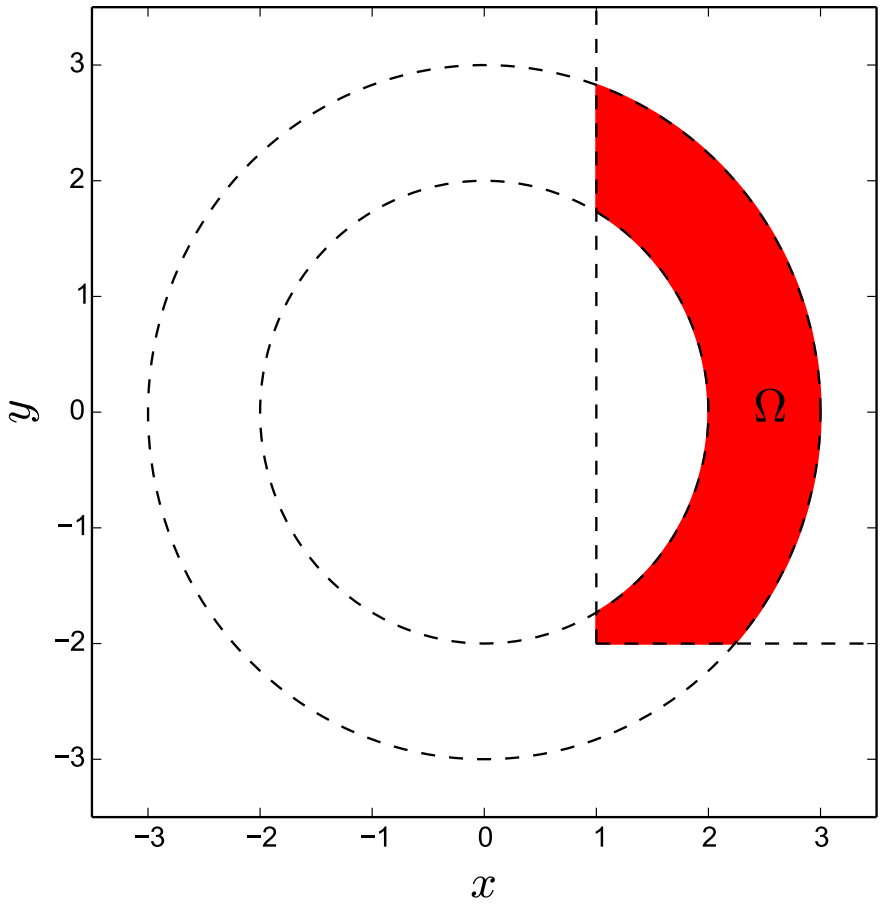
Let Ω denote this volume. Suppose that we want to find the integral of $f(x,y)$ over this region. We can re-cast the integral over Ω as an integral over a square that completely encompasses Ω. In this case, the smallest rectangle to contain Ω starts has bottom left corner $(1,-2)$ and top right corner $(3,3)$. This rectangle is shown in blue in the figure below.

Then:

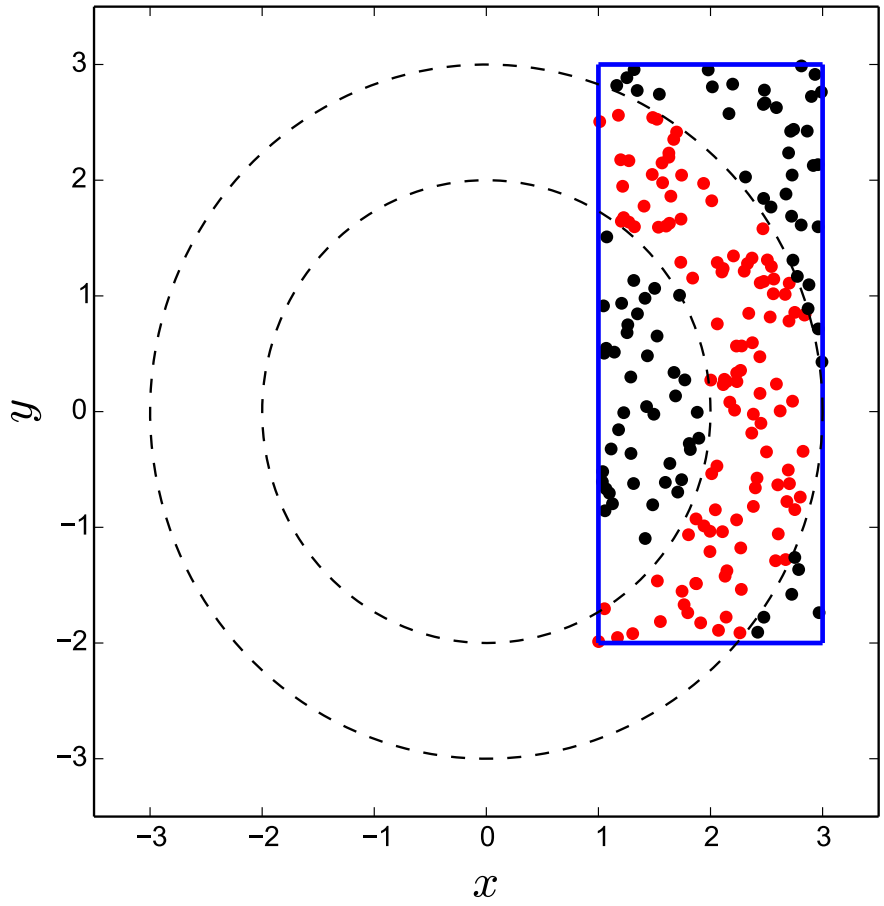
$\iint_\Omega f(x,y)\,dxdy = \int_{-2}^{3}\int_{1}^{3} g(x,y)\,dxdy$

where

$$g(x,y) = \begin{cases} f(x,y) & (x,y) \in \Omega \\ 0 & \text{otherwise} \end{cases}$$


3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
y
x
Ω

This is illustrated in the figure below, which shows 200 points randomly distributed within the rectangle bounded by the blue lines. Points in red fall within Ω and thus contribute to the integral, while points in black fall outside Ω.



To give a concrete example, let's take $f(x, y) = y^2$.

```
import numpy as np

def f(x_y):
    x,y = x_y
    return y**2

def g(x_y):
    """
    The integrand.
    """
    x,y = x_y
    r = np.sqrt(x**2 + y**2)
    if r >= 2. and r <= 3. and x >= 1. and y >= -2.:
        # (x,y) in correct volume
        return f(x_y)
    else:
        return 0.

mcquad(g,npoints=100000,xl=[1.,-2.],xu=[3.,3.],nprocs=4)
# (9.9161745618624231, 0.049412524880183335)
```

1.4 Array-like integrands

We are not limited to integrands that return floats. We can have integrands that return array objects. This can be useful for calculating several integrals at the same time. For instance, let's say that we want to calculate both x^2 and y^2 in the volume Ω described in the previous section. We can do both these integrals simultaneously.

```
import numpy as np

def f(x_y):
    """ f(x_y) now returns an array with both integrands. """
    x,y = x_y
    return np.array((x**2,y**2))

def g(x_y):
    x,y = x_y
    r = np.sqrt(x**2 + y**2)
    if r >= 2. and r <= 3. and x >= 1. and y >= -2.:
        # (x,y) in correct volume
        return f(x_y)
    else:
        return np.zeros((2,))

result, error = mcquad(g,npoints=100000,xl=[1.,-2.],xu=[3.,3.],nprocs=4)
print result
# [ 23.27740875   9.89103493]
print error
# [ 0.08437993  0.04938343]
```

We see that if the integrand returns an array, *mcquad* will return two arrays of the same shape, the first corresponding to the values of the integrand and the second to the error in those values.

1.5 Importance sampling

The techniques described in the previous sections work well if the variance of the integrand is similar over the whole integration volume. As a counter-example, consider the integral $\int_{-\infty}^{\infty} \cos(x)^2 e^{-x^2/2}\,dx$. Only the area around $x = 0$ contributes to the integral in a significant way, while the tails, which stretch out to infinity, contribute almost nothing.

We would therefore like a method that samples more thoroughly regions where the integral varies rapidly. This is the idea behind **importance sampling**.

To use importance sampling, we need to factor the integrand $f(x)$ into a probability distribution $\rho(x)$ and another function $h(x)$. For the integrand described above, we can take $\rho(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$ (normal distribution) and $h(x) = \sqrt{2\pi}\cos^2(x)$. We can then draw samples from $g(x)$ and, for each sample, calculate $h(x)$. This is done using the function *mcimport*:

```
>>> from skmonaco import mcimport
>>> from numpy.random import normal
>>> from math import cos,sqrt,pi
>>> result, error = mcimport(lambda x: sqrt(2.*pi)*cos(x)**2, # h(x)
...     npoints = 100000,
...     distribution = normal # rho(x)
...     )
>>> result # Correct answer: 1.42293
1.423388348518721
```

(continues on next page)

(continued from previous page)

```
>>> error
0.002748743084305
```

We see that *mcimport* takes at least three arguments, the first one being the function $h(x)$, that is, the integrand divided by the probability distribution from which we sample, the second being the number of points and the third is a function that returns random samples from the probability distribution $\rho(x)$. The `numpy.random` module provides many useful distributions that can be passed to `mcimport`. *mcimport* returns two numbers: the first corresponds to the integral estimate, and the second corresponds to the estimated error.

1.5.1 Multi-dimensional integrals

To look at a slightly more complicated example, let's integrate $f(x, y) = e^{-(y+2)}$ in the truncated ring described *above*. The integrand is largest around $y = -2$, and decays very quickly:

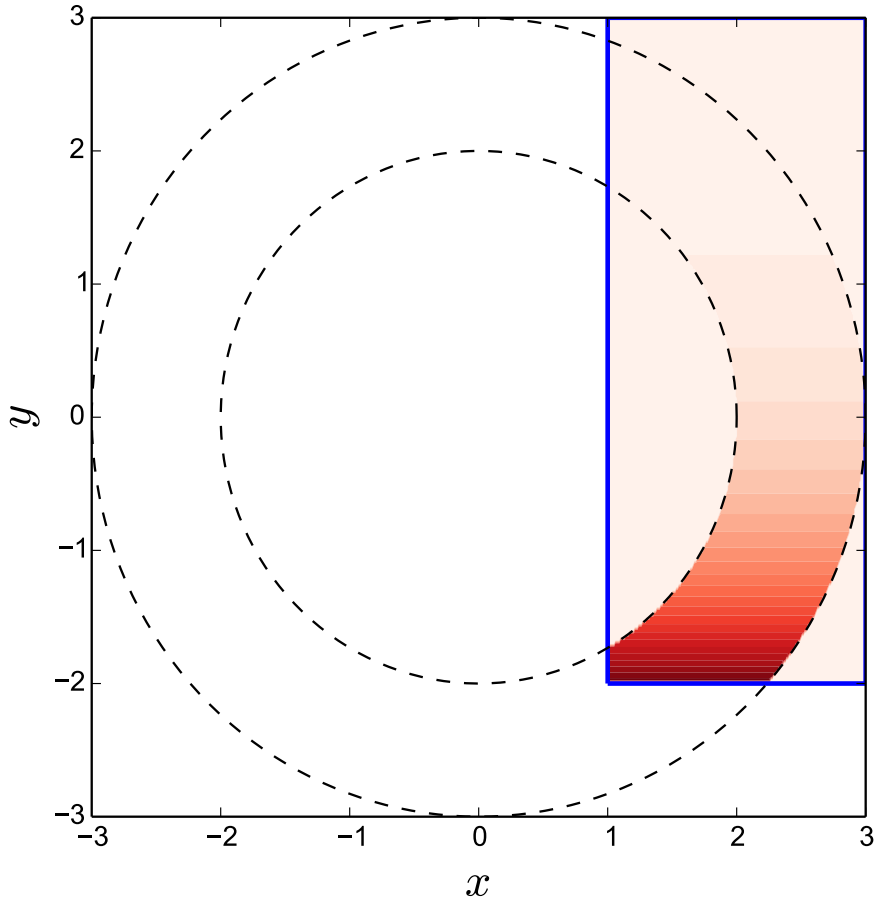


Fig. 1: $e^{-(y+2)}$ in the truncated ring.

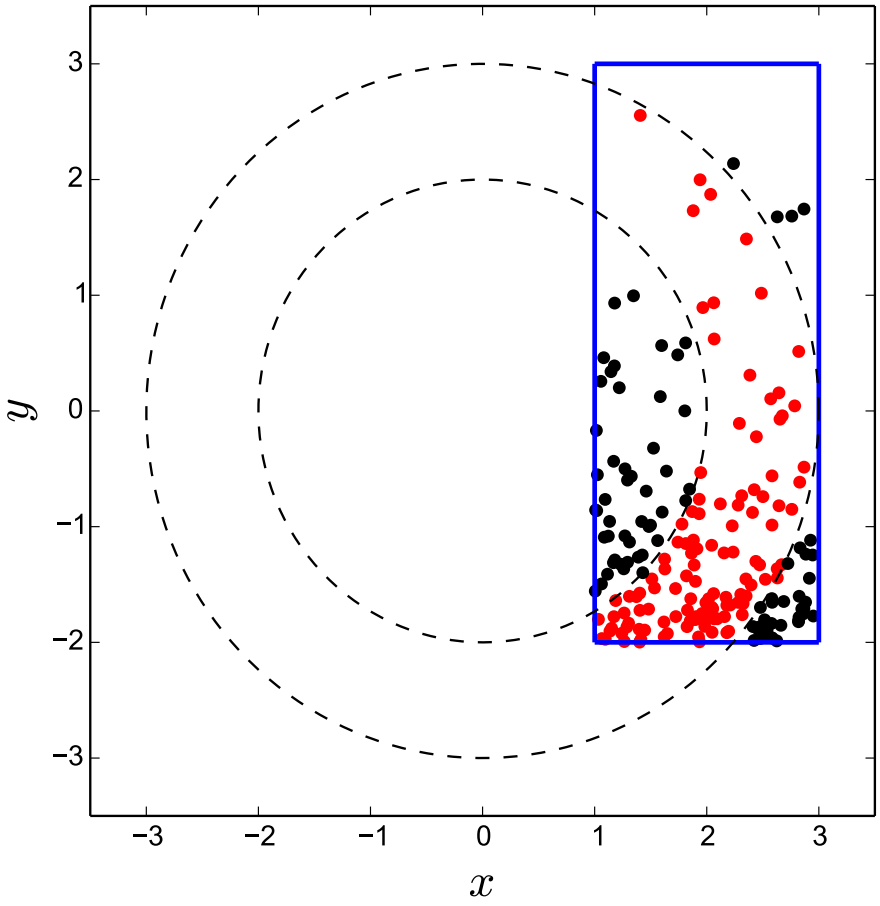
It makes sense to try and concentrate sample points around $y = -2$. We can do this by sampling from the exponential distribution (centered about $y = -2$) along y and uniformly along x (in the region $1 \leq x < 3$), such that $\rho(x, y) = \frac{1}{2}e^{-(y+2)}$, where the pre-factor of $1/2$ arises from the normalisation condition on the uniform distribution along x. The *distribution* function that must be passed to *mcimport* must take a *size* argument and return an array of shape `(size,d)` where *d* is the number of dimensions of the integral. This could be achieved with the following function:

```
from numpy.random import exponential, uniform

def distribution(size):
    """
    Return `size` (x,y) points distributed according to a uniform distribution
    along x and an exponential distribution along y.
    """
    xs = uniform(size=size,low=1.0,high=3.0)
    ys = exponential(size=size,scale=1.0) - 2.
    return np.array((xs,ys)).T

distribution(100).shape
# (100,2)
```

This is what 200 points distributed according to `distribution` look like. Points that fall within Ω are colored in red, and those that fall outside are colored in black. Evidently, points are concentrated about $y = -2$, such that this region gets sampled more often.



The function to sample is $f(x, y) = 2$ (the factor of 2 cancels the `1/(high-low)` prefactor in the uniform distribution) if $f(x, y) \in \Omega$ and 0 otherwise.

```
def f(x_y):
    """
    The integrand.
```

(continues on next page)

(continued from previous page)

```
    """
    x,y = x_y
    r = np.sqrt(x**2 + y**2)
    if r >= 2. and r <= 3. and x >= 1. and y >= -2.:
        # (x,y) in correct volume
        return 2.
    else:
        return 0.

mcimport(f,1e5,distribution,nprocs=4)
# (1.18178, 0.0031094848254976256)
```

1.5.2 Choosing a probability distribution

Generally, it is far from obvious how to split the integrand $f(x)$ into a probability distribution ρ and an integration kernel $h(x)$. It can be shown *[NR]* that, optimally $\rho \propto |f|$. Unfortunately, this is likely to be very difficult to sample from.

The following recipe is relatively effective:

1. If the integrand decays to 0 at the edge of the integration region, find a distribution that has the same rate of decay, or decays slightly slower. Thus, if, for instance, the integrand if $p(x)e^{-x^2}$, where $p(x)$ is a polynomial, sample from a normal distribution with $\sigma = 1/\sqrt{2}$, or slightly less if $p(x)$ contains high powers that delay the onset of the exponential decay.
2. Locate the mode of your distribution at the same place as the mode of the integrand.

The following two cases should be avoided:

1. The probability distribution should not be low in places where the integrand is large.
2. Somewhat less importantly, the probability distribution should not be high in regions where the variance of the integrand is low.

CHAPTER 2

Reference Guide

2.1 Uniform sampling Monte Carlo integration

mcquad samples uniformly from a hypercube. This method can also be used to integrate over more complicated volumes using the procedure described in *Complex integration volumes*. It will lead to large errors if the integration region is large and the integrand changes rapidly over a small fraction of the total integration region.

skmonaco.**mcquad**(*f*, *npoints*, *xl*, *xu*, *args=()*, *rng=None*, *nprocs=1*, *seed=None*, *batch_size=None*)

Compute a definite integral.

Integrate *f* in a hypercube using a uniform Monte Carlo method.

Parameters **f** : function

The integrand. It must take an iterable of length *d*, where *d* is the dimensionality of the integral, as argument, and return either a float or a numpy array.

npoints : int

Number of points to use for the integration.

xl, xu : iterable

Iterable of length *d*, where *d* is the dimensionality of the integrand, denoting the bottom left corner and upper right corner of the integration region.

Returns **value** : float or numpy array.

The estimate for the integral. If the integrand returns an array, this will be an array of the same shape.

error : float or numpy array

An estimate for the error (the integral has, approximately, a 0.68 probability of being within *error* of the correct answer).

Other Parameters **nprocs** : int >= 1, optional

Number of processes to use concurrently for the integration. Use nprocs=1 to force a serial evaluation of the integral. This defaults to 1.

seed : int, iterable or None

Seed for the random number generator. Running the integration with the same seed guarantees that identical results will be obtained (even if nprocs is different). If the argument is absent, this lets the random number generator handle the seeding. If the default rng is used, this means the seed will be read from */dev/random*.

rng : module or class, optional

Random number generator. Must expose the attributes *seed* and *ranf*. The `numpy.random` module by default.

batch_size : int, optional

The integration is batched, meaning that *batch_size* points are generated, the integration is run with these points, and the results are stored. Each batch is run by a single process. It may be useful to reduce *batch_size* if the dimensionality of the integration is very large.

args : list

List of arguments to pass to f when integrating.

Examples

Integrate x*y over the unit square. The true value is 1./4.

```
>>> mcquad(lambda x: x[0]*x[1], npoints=20000, xl=[0.,0.],xu=[1.,1.])
(0.24966..., 0.0015488...)
```

Calculate pi/4 by summing over all points in the unit circle that are within 1 unit of the origin.

```
>>> mcquad(lambda x: 1 if sum(x**2) < 1 else 0.,
...     npoints=20000, xl=[0.,0.], xu=[1.,1.])
(0.78550..., 0.0029024...)
>>> np.pi/4.
0.7853981633974483
```

The integrand can return an array. This can be used to calculate several integrals at once.

```
>>> result, error = mcquad(
...      lambda x: np.exp(-x**2)*np.array((1.,x**2,x**4,x**6)),
...     npoints=20000, xl=[0.], xu=[1.])
>>> result
array([ 0.7464783 ,  0.18945015,  0.10075603,  0.06731908])
>>> error
array([ 0.0014275 ,  0.00092622,  0.00080145,  0.00069424])
```

2.2 Importance sampling

In importance sampling, the integrand is factored into the product of a probability density $\rho(x)$ and another function $h(x)$:

$$f(x) = \rho(x)h(x)$$

The integration proceeds by sampling from $\rho(x)$ and calculating $h(x)$ at each point. In *scikit-monaco*, this is achieved with the *mcimport* function.

skmonaco.**mcimport**(*f*, *npoints*, *distribution*, *args=()*, *dist_kwargs={}*, *rng=None*, *nprocs=1*, *seed=None*, *batch_size=None*, *weight=1.0*)

Compute a definite integral, sampling from a non-uniform distribution.

This routine integrates `f(x)*distribution(x)` by drawing samples from *distribution*. Choosing *distribution* such that the variance of *f* is small will lead to much faster convergence than just using uniform sampling.

Parameters **f** : function

A Python function or method to integrate. It must take an iterable of length *d*, where *d* is the dimensionality of the integral, as argument, and return either a float or a numpy array.

npoints : int >= 2

Number of points to use in the integration.

distribution : function

A Python function or method which returns random points. `distribution(size) -> numpy array of shape (size,d)` where *d* is the dimensionality of the integral. If `d==1`, distribution can also return an array of shape `(size,)`. The module *numpy.random* contains a large number of distributions that can be used here.

Returns **value** : float or numpy array

The estimate for the integral. If the integrand returns an array, this will be an array of the same shape.

error : float or numpy array

The standard deviation of the result. If the integrand returns an array, this will be an array of the same shape.

Other Parameters **args** : tuple, optional

Extra arguments to be passed to *f*.

dist_kwargs : dictionary, optional

Keyword arguments to be passed to *distribution*.

nprocs : int >= 1, optional

Number of processes to use for the integration. 1 by default.

seed : int, iterable, optional

Seed for the random number generator. Running the integration with the same seed guarantees that identical results will be obtained (even if nprocs is different). If the argument is absent, this lets the random number generator handle the seeding. If the default rng is used, this means the seed will be read from */dev/random*.

rng : module or class, optional

Random number generator. Must expose the attributes *seed* by default. The `numpy.random` module by default.

batch_size : int, optional

The integration is batched, meaning that *batch_size* points are generated, the integration is run with these points, and the results are stored. Each batch is run by a single process.

It may be useful to reduce *batch_size* if the dimensionality of the integration is very large.

weight : float, optional

Multiply the result and error by this number. 1.0 by default. This can be used when the measure of the integral is not 1.0. For instance, if one is sampling from a uniform distribution, the integration volume could be passed to *weight*.

Examples

Suppose that we want to integrate `exp(-x**2/2)` from x = -1 to 1. We can sample from the normal distribution, such that the function `f` is `f = sqrt(2*pi) if -1. < x < 1. else 0`.

```
>>> import numpy as np
>>> from numpy.random import normal
>>> f = lambda x: np.sqrt(2*np.pi) * (-1. < x < 1.)
>>> npoints = 1e5
>>> mcimport(f,npoints,normal)
(1.7119..., 0.00116...)
>>> from scipy.special import erf
>>> np.sqrt(2.*np.pi) * erf(1/np.sqrt(2.)) # exact value
1.7112...
```

Now suppose that we want to integrate exp(z^2) in the unit sphere (x^2 + y^2 + z^2 < 1). Since the integrand is uniform along x and y and normal along z, we choose to sample uniformly from x and y and normally along z. We can hasten the integration by using symmetry and only considering the octant (x,y,z > 0).

```
>>> import numpy as np
>>> from numpy.random import normal, uniform
>>> f = lambda (x,y,z): np.sqrt(2.*np.pi)*(z>0.)*(x**2+y**2+z**2<1.)
>>> def distribution(size):
...     xs = uniform(size=size)
...     ys = uniform(size=size)
...     zs = normal(size=size)
...     return np.array((xs,ys,zs)).T
>>> npoints = 1e5
>>> result, error = mcimport(f,npoints,distribution)
>>> result*8,error*8
(3.8096..., 0.0248...)
```

The integrand can also return an array. Suppose that we want to calculate the integrals of both `exp(z**2)` and `z**2*exp(z**2)` in the unit sphere. We choose the same distribution as in the previous example, but the function that we sum is now:

```
>>> f = lambda (x,y,z): (np.sqrt(2.*np.pi)*(z>0.)*(x**2+y**2+z**2<1.)*
...     np.array((1.,z**2)))
>>> result, error = mcimport(f,npoints,distribution)
>>> result*8
array([ 3.81408558,  0.67236413])
>>> error*8
array([ 0.02488709,  0.00700179])
```

2.3 MISER Monte Carlo

mcmiser samples from a hypercube using the MISER algorithm, and can also be used to integrate over more complicated volumes using the procedure described in *Complex integration volumes*. The algorithm is adaptive, inasmuch as it will use more points in regions where the variance of the integrand is large. It is almost certainly likely to be superior to *mcquad* for "complicated" integrands (integrands which are smooth over a large fraction of the integration region but with large variance in a small region), with dimensionality below about 6.

skmonaco.**mcmiser**(*f*, *npoints*, *xl*, *xu*, *args=()*, *rng=None*, *nprocs=1*, *seed=None*, *min_bisect=100*, *pre_frac=0.1*, *exponent=0.6666666666666666*)

Compute a definite integral.

Integrate *f* in a hypercube using the MISER algorithm.

Parameters **f** : function

The integrand. Must take an iterable of length *d*, where *d* is the dimennsionality of the integral, as argument, and return a float.

npoints : int

Number of points to use for the integration.

xl, xu : iterable

Iterable of length *d*, where *d* is the dimensionality of the integrand, denoting the bottom left corner and upper right corner of the integration region.

Returns **value** : float

The estimate for the integral.

error : float

An estimate for the error (the integral has, approximately, a 0.68 probability of being within *error* of the correct answer).

Other Parameters **nprocs** : int >= 1, optional

Number of processes to use concurrently for the integration. Use nprocs=1 to force a serial evaluation of the integral. This defaults to 1. Increasing *nprocs* will increase the stochastic error for a fixed number of samples (the algorithm just runs several MISER runs in parallel).

seed : int, iterable or None

Seed for the random number generator. Running the integration with the same seed and the same *nprocs* guarantees that identical results will be obtained. If the argument is absent, this lets the random number generator handle the seeding. If the default rng is used, this means the seed will be read from */dev/random*.

rng : module or class, optional

Random number generator. Must expose the attributes *seed* and *ranf*. The `numpy.random` module by default.

args : list

List of arguments to pass to *f* when integrating.

min_bisect : int

Minimum number of points inn which to run a bisection. If the integrator has a budget of points < *min_bisect* for a region, it will fall back onto uniform sampling.

pre_frac : float

Fraction of points to use for pre-sampling. The MISER algorithm will use this fraction of its budget for a given area to decide how to bisect and how to apportion point budgets.

exponent : float

When allocating points to the sub-region, the algorithm will give a fraction of points proportional to `range**exponent`, where `range` is the range of the integrand in the sub-region (as estimated by using a fraction *pre_frac* of points). Numerical Recipes *[NR0]* recommends a fraction of 2/3.

Notes

Unlike mcquad, the integrand cannot return an array. It must return a float.

The implementation is that proposed in Numerical Recipes *[NR0]*: when apportioning points, we use `|max-min|` as an estimate of the variance in each sub-area, as opposed to calculating the variance explicitly.

References

[NR0]

Examples

Integrate x*y over the unit square. The correct value is 1/4.

```
>>> mcmiser(lambda x: x[0]*x[1], npoints=20000, xl=[0.,0.], xu=[1.,1.])
(0.249747..., 0.000170...)
```

Note that this is about 10 times more accurate than the equivalent call to *mcquad*, for the same number of points.

2.4 Utility functions

skmonaco.**integrate_from_points**(*f*, *points*, *args=()*, *nprocs=1*, *batch_size=None*, *weight=1.0*)

Compute a definite integral over a set of points.

This routine evaluates *f* for each of the points passed, returning the average value and variance.

Parameters **f** : function

A Python function or method to integrate. It must take an iterable of length *d*, where *d* is the dimensionality of the integral, as argument, and return either a float or a numpy array.

points : numpy array

A numpy array of shape `(npoints,dim)`, where *npoints* is the number of points and *dim* is the dimentionality of the integral.

Returns **value** : float or numpy array.

The estimate for the integral. If the integrand returns an array, this will be an array of the same shape.

error : float or numpy array

An estimate for the error (the integral has, approximately, a 0.68 probability of being within *error* of the correct answer).

Other Parameters **nprocs** : int >= 1, optional

Number of processes to use concurrently for the integration. Use nprocs=1 to force a serial evaluation of the integral. This defaults to the value returned by multiprocessing.cpu_count().

batch_size : int, optional

The integration is batched, meaning that *batch_size* points are generated, the integration is run with these points, and the results are stored. Each batch is run by a single process. It may be useful to reduce *batch_size* if the dimensionality of the integration is very large.

Examples

Integrate x*y over the unit square.

```
>>> from numpy.random import ranf
>>> npoints = int(1e5)
>>> points = ranf(2*npoints).reshape((npoints,2)) # Generate some points
>>> points.shape
(100000,2)
>>> integrate_from_points(lambda x_y:x_y[0]*x_y[1], points)
(0.24885..., 0.00069...)
```

CHAPTER 3

Contributing

Scikit-monaco is in its infancy. There is a lot of scope for developpers to make a strong impact by contributing code, documentation and expertise. All contributions are welcome and, if you are unsure how to contribute or are unfamiliar with scipy and numpy, we will happily mentor you. Just send an email to <*pascal@bugnion.org*>.

3.1 How to contribute

The documentation for Numpy gives a detailed description of how to contribute. Most of this information applies to development for `scikit-monaco`.

3.1.1 Developping with git

You will need the Git version control system and an account on Github to contribute to scikit-monaco.

1. Fork the project repository by clicking *Fork* in the top right of the page. This will create a copy of the fork under your account on Github.
2. Clone the repository to your computer:

   ```
   $ git clone https://github.com/YourUserID/scikit-monaco.git
   ```

3. Install scikit-monaco by running:

   ```
   $ python setup.py install
   ```

   in the package's root directory. You should now be able to test the code by running `$ nosetests` in the package's root directory.

You can now make changes and contribute them back to the source code:

1. Create a branch to hold your changes:

```
$ git checkout -b my-feature
```

and start making changes.

2. Work on your local copy. When you are satisfied with the changes, commit them to your local repository:

```
$ git add modified files
$ git commit
```

You will be asked to write a commit message. Explain the reasoning behind the changes that you made.

3. Propagate the changes back to your github account:

```
$ git push -u origin my-feature
```

4. To integrate the changes into the main code repository, click *Pull Request* on the *scikit-quantum* repository page on your accont. This will notify the committers who will review your code.

3.1.2 Updating your repository

To keep your private repository updated, you should add the main repository as a remote:

```
$ git remote add upstream git://github.com/scikit-monaco/scikit-monaco.git
```

To update your private repository, you can then fetch new commits from upstream:

```
$ git fetch upstream
$ git rebase upstream/master
```

3.2 Guidelines

3.2.1 Workflow

We loosely follow the git workflow used in numpy development. Features should be developped in separate branches and merged into the master branch when complete. Avoid putting new commits directly in your `master` branch.

3.2.2 Code

Please follow the PEP8 conventions for formatting and indenting code and for variable names.

3.2.3 Documentation

Scikit-quantum uses sphinx with numpydoc to process the documentation. We follow the numpy convention on writing docstrings.

Use `make html` in the `doc` folder to build the documentation.

3.2.4 Testing

We use nose to test *scikit-quantum*. Running `nosetests` in the root directory of the package will run the tests.

CHAPTER 4

Release Notes

4.1 Scikit-monaco v0.2.1 release notes

- Fixed bug when using MISER with numpy 1.9.0.

4.2 Scikit-monaco v0.2 release notes

- Add MISER algorith for recursive stratified sampling.
- All integration routines now respond to KeyboardInterrupt.
- Additional benchmarks for importance sampling.

4.3 Scikit-monaco v0.1.5 release notes

Version 0.1.5 provides an important bugfix. Seeds were not being properly generated, such that repeated calls to `mcquad` or `mcimport` that came within the same value of `int(time.time())` had the same seed.

Seeding is now left to the (more capable) hands of the RNG, which just gets passed `seed(None)` if the user hasn't specified a seed.

Bibliography

[NR] Press, W. H, Teutolsky, S. A., Vetterling, W. T., Flannery, B. P., Numerical Recipes, The art of scientific computing, 3rd edition, Cambridge University Press (2007).

[NR0] W. H. Press, S. A. Teutolsky, W. T. Vetterling, B. P. Flannery, "Numerical recipes: the art of scientific computing", 3rd edition. Cambridge University Press (2007)

Index

I

M