Lab 18

Monte-Carlo Integration

Lab Objective: *Use Monte-Carlo integration to estimate areas.*

Some multivariable integrals which are critical in applications are impossible to evaulate symbolically. For example, the integral of the joint normal distribution

$$\int_{\Omega} \frac{1}{\sqrt{(2\pi)^k}} e^{-\frac{\mathbf{x}^T\mathbf{x}}{2}}$$

is ubiquitous in statistics. However, the integrand does not have a symbolic antiderivative. This means we must use numerical methods to evaluate this integral.

The standard technique for numerically evaluating multivariable integrals is *Monte-Carlo Integration*. Monte-Carlo (MC) integration is radically different from 1-dimensional techniques like Simpson's rule, relying on probability to calculate the integral. Although it converges slowly, MC integration is frequently used to evaluate multivariable integrals because the higher-dimensional analogs of methods like Simpson's rule are inefficient.

A motivating example

Suppose we want to numerically compute the area of a circle of radius 1. From analytic methods, we know the answer is π. Empirically, we can estimate this quantity by randomly choosing points in a 2×2 square. The percent of points that land in the inscribed circle, times the area of the square, should approximately equal the area of the circle (see Figure 18.1).

We do this in NumPy as follows. First generate 500 random points in the square $[0, 1] \times [0, 1]$.

```
>>> numPoints = 500
>>> points = np.random.rand(2, numPoints)
```

We rescale and shift these points to be uniformly distributed in $[-1, 1] \times [-1, 1]$.

```
>>> points = points*2-1
```

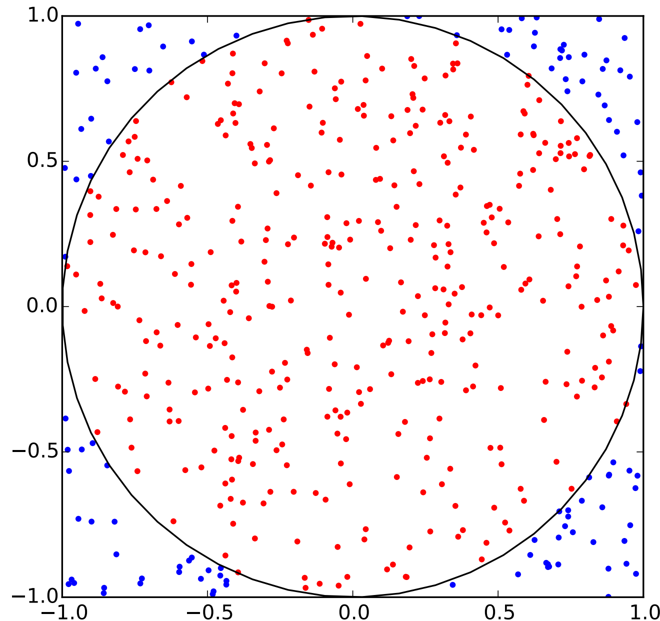

Figure 18.1: Finding the area of a circle using random points

Next we compute the number of points in the unit circle. The function `np.hypot(a, b)` returns the norm of the vector (a, b).

```
>>> # Create a mask of points in the circle
>>> circleMask = np.hypot(points[0,:], points[1,:]) <= 1
>>> # Count how many there are
>>> numInCircle = np.count_nonzero(circleMask)
```

Finally, we approximate the area.

```
>>> # Area is approximately (area of the square)*(num points in circle)/(total ←
    num points)
>>> 4.*numInCircle/numPoints
3.024
```

This differs from π by about 0.117.

We analyze the error of the MC method by repeating this experiment for many values of `numPoints` and plotting the errors. The result is the blue line in Figure 18.2. The error appears to be proportional to $1/\sqrt{N}$ where $N =$`numPoints` (the green line in Figure 18.2). This means that to divide the error by10, we must sample *100 times* more points.

This is a slow convergence rate, but it is independent of the number of dimensions of the problem. This dimension independence is what makes the MC method useful for multivariable integrals.

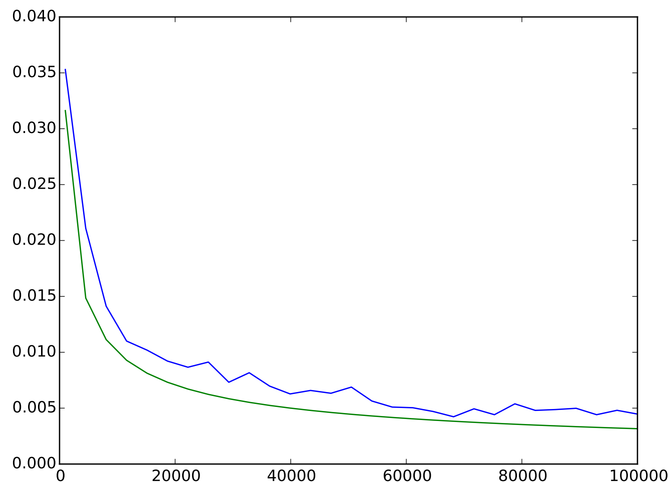

Figure 18.2: The Monte-Carlo integration method was used to compute the area of a circle of radius 1. The blue line plots the average error in 100 runs of the MC method on N sample points, where N appears on the horizontal axis. The green line is a plot of $1/\sqrt{N}$.

Monte-Carlo Integration

You can calculate the area of the unit circle with the following integration problem:

$$\text{Area of unit circle } = \int_{[-1,1]\times[-1,1]} f(x,y)dA$$

where

$$f(x,y) = \begin{cases} 1 & \text{if } x,y \text{ is in the unit circle} \\ 0 & \text{otherwise.} \end{cases} \tag{18.1}$$

We can use a random-points method as above to approximate any integral. Suppose we with to evaluate

$$\int_\Omega f(x)dV.$$

We can approximate this integral using the formula

$$\int_\Omega f(x)dV \approx V(\Omega)\frac{1}{N}\sum_{i=1}^{N} f(x_i), \tag{18.2}$$

where x_i are uniformly distributed random vectors in Ω and $V(\Omega)$ is the volume of Ω. This is the formula for Monte-Carlo Integration.

In our example, Ω was the box $[-1,1]\times[-1,1]$ and f was the function defined in (18.1). Then $\sum_{i=1}^{N} f(x_i)$ is the number of points in the unit circle, N is the total number of points, and (18.2) is the same as the formula we derived previously.

The intuition behind (18.2) is that $\frac{1}{N}\sum_{i=1}^{N} f(x_i)$ approximates the average value of f on Ω. We multiply this (approximate) average value by the volume of Ω to get the (approximate) integral of f on Ω.

As an 1-dimensional example consider the integral

$$\int_0^1 x dx \approx (1-0)\frac{1}{N}\sum_{i=1}^{N} x_i = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

The integral on the left-hand-side is $1/2$. In the approximation on the right-hand-side, x_i is drawn from a uniform distribution on $[0,1]$. The average of N such draws will converge to $1/2$.

Problem 1. Implement Monte-Carlo integration with the following function. Your implementation should run the Monte-Carlo algorithm several times and return the average of those runs.

```
def mc_int(f, mins, maxs, numPoints=500, numIters=100):
    '''Use Monte-Carlo integration to approximate the integral of f
    on the box defined by mins and maxs.

    INPUTS:
    f         - A function handle. Should accept a 1-D NumPy array
            as input.
    mins      - A 1-D NumPy array of the minimum bounds on integration.
    maxs      - A 1-D NumPy array of the maximum bounds on integration.
    numPoints - An integer specifying the number of points to sample in
            the Monte-Carlo method. Defaults to 500.
    numIters - An integer specifying the number of times to run the
            Monte Carlo algorithm. Defaults to 100.

    ALGORITHM:
    Run the Monte-Carlo algorithm `numIters' times and return the average
    of these runs.

    EXAMPLES:
    >>> f = lambda x: np.hypot(x[0], x[1]) <= 1
    >>> # Integral over the square [-1,1] x [-1,1] should be pi
    >>> mc_int(f, np.array([-1,-1]), np.array([1,1]))
    3.1290400000000007
    '''
```

Hints:

1. To create a random array of points on which to evaluate `f`, first create a random array of points in $[0,1] \times \ldots \times [0,1]$. Then multiply this array by the appropriate vector to stretch it the right amount in each direction. Finally, add the appropriate vector to shift the points to the right location.

2. You can evaluate `f` on an array of points in one line using `np.apply_along_axis()`.

One application of Monte Carlo integration is integrating probability density functions that do not have closed form solutions.

Problem 2. The joint normal distribution of N independent random variables with mean 0 and variance 1 is

$$f(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^N}} e^{-\frac{\mathbf{x}^T\mathbf{x}}{2}}.$$

The integral of $f(\mathbf{x})$ over a box is the probability that a draw from the distribution will be in the box. However, $f(\mathbf{x})$ does not have a symbolic antiderivative.

1. The integral of this function on $B = [-1,1] \times [-1,1] \times [-1,1] \subset \mathbb{R}^3$ can be computed in SciPy with the following code.

```
import scipy.stats as stats

# Define the bounds of the box to integrate over
mins = np.array([-1, -1, -1])
maxs = np.array([1, 1, 1])

# Each variable has mean 0 and variance 1
means = np.zeros(3)
covs = np.ones(3)

# Compute the integral
value, inform = stats.mvn.mvnun(mins, maxs, means, covs)
```

   Then `value` is the integral of $f(\mathbf{x})$ on B. Use SciPy to integrate $f(\mathbf{x})$ on $\Omega = [-0.5, 0.75] \times [0,1] \times [0,0.5] \times [0,1] \subset \mathbb{R}^4$.

2. Use the function `mc_int()` you wrote in Problem 1 to integrate $f(\mathbf{x})$ on Ω with 10, 100, 1000, and 10,000 sample points. Plot the errors of your estimates.

A caution

You can run into trouble if you try to use MC integration on an integral that does not converge. For example, we may attempt to evaluate

$$\int_0^1 \frac{1}{x}$$

with MC integragtion using the following code.

```
>>> k = 5000
>>> np.mean(1/np.random.rand(k,1))
21.237332864358656
```

Since this code returns a finite value, so we could assume that this integral has a finite value. In fact, the integral is infinite. We could discover this emprically by using larger and larger values of k, and noting that MC integration returns larger and larger values.