Lab 1

Monte-Carlo Integration

Lab Objective: *Implement Monte-Carlo integration to estimate integrals. Use Monte-Carlo Integration to calculate the integral of the joint normal distribution.*

Some multivariable integrals which are critical in applications are impossible to evaluate 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*. In the next lab, we will approximate this integral using a modified version of Monte-Carlo Integration. In this lab, we address the basics of Monte-Carlo Integration.

Monte-Carlo (MC) integration is radically different from 1-dimensional techniques like Simpson's rule. Whereas Simpson's rule is purely computational, MC integration relies 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 1.1).

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)
```

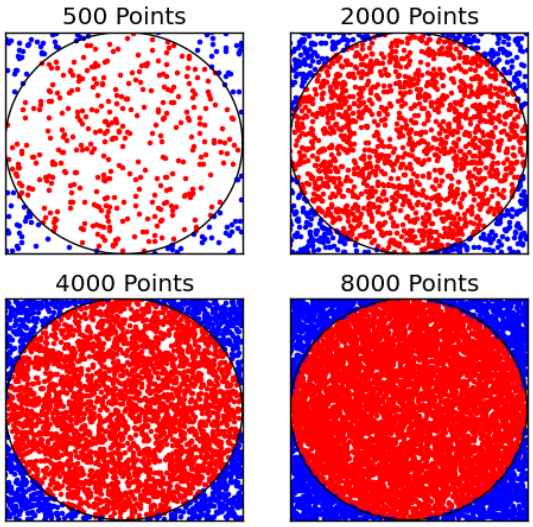

Figure 1.1: Finding the area of a circle using random points

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

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

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) where a and b are the x- and y-components, respectively.

```
>>> # 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.

Problem 1. Estimate the volume of the unit sphere using 1000000 sample points from $[-1,1]\times[-1,1]\times[-1,1]$. Your answer should be approximately 4.189.

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 1.2.

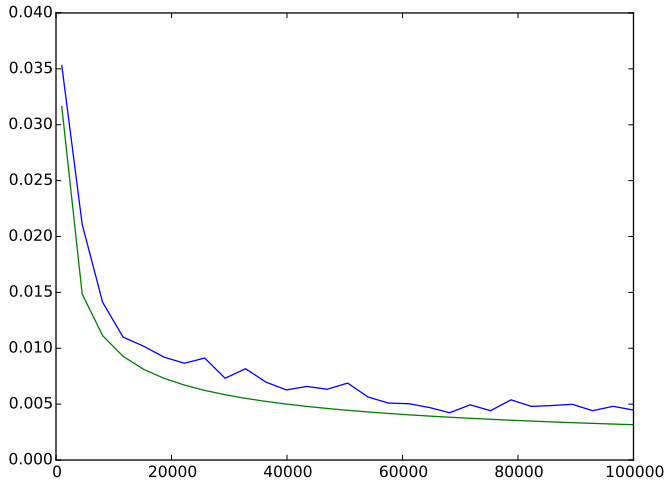

Figure 1.2: The Monte-Carlo integration method was used to compute the area of a circle of radius 1. The blue line plots the average error in 100 runs of the MC method on N sample points, where N appears on the horizontal axis. The green line is a plot of $1/\sqrt{N}$.

The error appears to be proportional to $1/\sqrt{N}$ where $N =$ `numPoints` (the green line in Figure 1.2). This means that to divide the error by 10, 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.

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

$$f(x,y) = \begin{cases} 1 & \text{if } (x,y) \text{ is in the unit circle} \\ 0 & \text{otherwise.} \end{cases} \tag{1.1}$$

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

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

We can approximate this integral using the formula

$$\int_\Omega f(x)dV \approx V(\Omega)\frac{1}{N}\sum_{i=1}^{N} f(x_i), \tag{1.2}$$

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 (1.1). 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 (1.2) is the same as the formula we derived previously.

The intuition behind (1.2) 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 2. Monte-Carlo Integration is particularly useful when trying to approximate integrals that would be difficult to calculate otherwise. Approximate the following integral:

$$\int_1^5 \left|\sin(10x)\cos(10x) + \sqrt{x}\sin(3x)\right| dx$$

Your answer should be approximately 4.502.

Problem 3. Implement Monte-Carlo integration with the following function. Your implementation should be robust enough to integrate any function f in $\mathbb{R}^n$ over any interval in $\mathbb{R}^n$. Your implementation should run the Monte-Carlo algorithm several times and return the average of those runs. Test your function by recalculating Problem 1 and 2.

```
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 (function) - The function to integrate. This function should
            accept a 1-D NumPy array as input.
        mins (1-D np.ndarray) - Minimum bounds on integration.
        maxs (1-D np.ndarray) - Maximum bounds on integration.
        numPoints (int, optional) - The number of points to sample in
            the Monte-Carlo method. Defaults to 500.
        numIters (int, optional) - An integer specifying the number of
            times to run the Monte Carlo algorithm. Defaults to 100.

    Returns:
```

```
        estimate (int) - The average of 'numIters' runs of the
            Monte-Carlo algorithm.

    Example:
        >>> 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 a vector of the lengths of the sides 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 using `np.apply_along_axis()` as follows:

```
# to evaluate the function f using the rows of vecs as input
>>> f = lambda x: la.norm(x)
>>> vecs = np.array([[1,1,1],[0,2,1],[0.5,0.5,0.5],[1,0,1]])
>>> np.apply_along_axis(f,1,vecs)
array([ 1.73205081,  2.23606798,  0.8660254 ,  1.41421356])
```

   In this example, we chose the axis parameter to be 1 to evaluate the rows of the matrix. If you would like a refresher on axes, see Lab **??**.

Problem 4. The exact value of the integral of

$$f(w,x,y,z) = \sin(x)y^5 - y^3 + zw + yz^3$$

on $[-1,1] \times [-1,1] \times [-1,1] \times [-1,1]$ is 0. Run the function `mc_int()` you wrote in Problem 3 on f with 100, 1000, and 10000 sample points. Use the default value of 500 iterations for your approximations. 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, we could assume that this integral has a finite value. In fact, the integral is infinite. We could discover this empirically by using larger and larger values of k, and noting that MC integration returns larger and larger values.

Lab 2

Importance Sampling and Monte Carlo Simulations

Lab Objective: *Use importance sampling to reduce the error and variance of Monte Carlo Simulations.*

Introduction

Using the traditional methods of Monte Carlo integration as discussed in the previous lab are not always the most efficient means to estimate an integral. For example, assume we were trying to find the probability that a randomly chosen variable X from the standard normal distribution is greater than 3. We know that one way to solve this is by solving the following integral:

$$P(X > 3) = \int_3^\infty f_X(t)\,dt = \frac{1}{\sqrt{2\pi}} \int_3^\infty e^{-t^2/2}\,dt \tag{2.1}$$

If we define the function $h : \mathbb{R} \to \mathbb{R}$ as

$$h(t) = \begin{cases} 1 & \text{if } t > 3 \\ 0 & \text{if } t \leq 3 \end{cases},$$

we can rewrite this integral as,

$$\int_3^\infty f_X(t)\,dt = \int_{-\infty}^\infty h(t) f_X(t)\,dt.$$

By the Law of the Unconscious Statistician (see Volume 2 §3.5), we can restate the integral above as,

$$\int_{-\infty}^\infty h(t) f_X(t)\,dt = E[h(X)].$$

Being able to write integrals as expected values is an essential tool in this lab.

Monte Carlo Simulation

In the last section, we expressed the probability of drawing a number greater than 3from the normal distribution as an expected value problem. We can now easily estimate this same probabilty using Monte Carlo simulation. Given a random i.i.d. sample $x_1, x_2, \cdots, x_N$ generated by f_X, we can estimate $E[h(X)]$ using

$$\widehat{E}_n[h(X)] = \frac{1}{N}\sum_{i=1}^{N} h(x_i) \tag{2.2}$$

Now that we have defined the estimator, it is now quite manageable to approximate Equation 2.1. By the Weak Law of Large Numbers (see Volume 2 §3.6), the estimate will get closer and closer to the actual value as we use more and more sample points.

Problem 1. Write a function in Python that estimates the probability that a random draw from the standard normal distribution is greater than 3 using Equation 2.2. Your answer should approach 0.0013499 for sufficiently large samples. What estimate do you get when you use a sample with 10^7 points? Print your results to the terminal.

Though this approach gets the job done, it turns out that this isn't very efficient. Since the probability of drawing a number greater than 3 from the standard normal distribution is so unlikely, it turns out we need many sample points to get a good approximation.

Importance Sampling

Importance sampling is one way to make Monte Carlo simulations converge much faster. We chose a different distribution to sample our points to generate more *important* points. With our example, we want to choose a distribution that would generate more numbers around 3 to get a more reliable estimate. The theory behind importance sampling boils down to the following result. In these equations, the random variable X is generated by f_X and the random variable Y is generated by g_X. We X and Y in this way for the remainder of the lab.

$$\begin{aligned} E[h(X)] &= \int_{-\infty}^{\infty} h(t) f_X(t)\, dt \\ &= \int_{-\infty}^{\infty} h(t) f_X(t) \left(\frac{g_X(t)}{g_X(t)}\right) dt \\ &= \int_{-\infty}^{\infty} \left(\frac{h(t) f_X(t)}{g_X(t)}\right) g_X(t)\, dt \\ &= E\left[\frac{h(Y) f_X(Y)}{g_X(Y)}\right] \end{aligned} \tag{2.3}$$

The corresponding estimator is,

$$\begin{aligned}\widehat{E}[h(X)] &= \widehat{E}\left[\frac{h(Y)f_X(Y)}{g_X(Y)}\right] \\ &= \frac{1}{N}\sum_{i=1}^{N}\frac{h(y_i)f_X(y_i)}{g_X(y_i)}\end{aligned}$$

The function f_X is the p.d.f. of the *target distribution*. The function g_X is the p.d.f. of the *importance distribution*. The fraction $\frac{f_X(X)}{g_X(X)}$ is called the *importance weight*. This allows us to draw a sample from any distribution with p.d.f. g_X as long as we multiply $h(X)$ by the importance weight. We will solve the same problem as in Problem 1 using importance sampling. We will choose g_X to be the normal distribution with $\mu = 4$ and $\sigma = 1$. We have chosen this distribution for g_X because it will give us more points closer to and greater than 3.

```
>>> import scipy.stats as ss
>>> f = lambda x : x > 3
>>> h = lambda x : ss.norm().pdf(x)
>>> g = lambda x : ss.norm(loc=4,scale=1).pdf(x)

# Sample from the N(4,1).
>>> N = 10**7
>>> X = np.random.normal(4,scale=1,size=N)

# Calculate estimate.
>>> 1./N * np.sum(f(X)*h(X)/g(X))
0.00134921134631
```

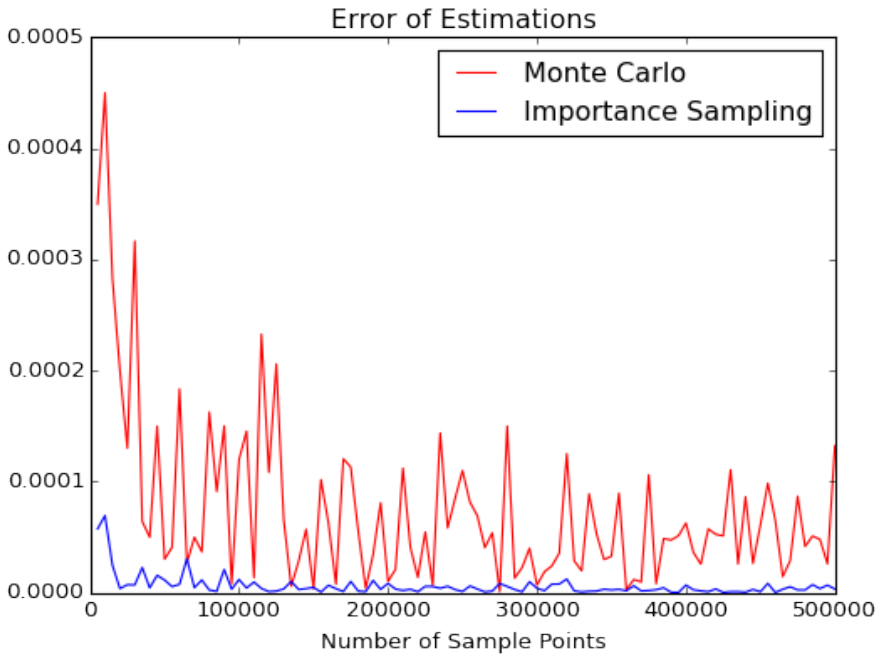

Figure 2.1: Comparison of error between standard method Monte Carlo and Importance Sampling method of Monte Carlo.

Problem 2. A tech support hotline receives an average of 2 calls per minute. What is the probability that they will have to wait at least 10 minutes to receive 9 calls? Implement your estimator using importance sampling. Calculate estimates using 5000, 10000, 15000, $\cdots$, 500000 sample points. Your answers should approach 0.00208725.

Hint: In Volume 2 §3.5, the gamma distribution is defined as,

$$f_X(x) = \frac{b^a x^{a-1} e^{-xb}}{\Gamma(a)}.$$

The version of the gamma distribution in `scipy.stats` is determined by the shape (a) and the scale (θ) of the distribution.

$$f_X(x) = \frac{1}{\Gamma(a)\theta^a} x^{a-1} e^{-x/\theta}$$

You can switch between these representations this with the fact that $\theta = 1/b$.

Problem 3. In this problem, we will visualize the benefits of importance sampling. Create a plot of the error of the traditional methods of Monte Carlo and the importance sampling methods of Monte Carlo. What do you observe? Your answers should resemble Figure 2.1.

Hint: The following code solves Problem 2 using traditional methods of Monte Carlo:

```
h = lambda x : x > 10
MC_estimates = []
for N in xrange(5000,505000,5000):
    X = np.random.gamma(9,scale=0.5,size=N)
    MC = 1./N*np.sum(h(X))
    MC_estimates.append(MC)
MC_estimates = np.array(MC_estimates)
```

Hint: The following code returns the actual value of Equation 2.1:

```
1 - ss.gamma(a=9,scale=0.5).cdf(10)
```

Generalizing the Principles of Importance Sampling

The examples we have explored to this point in the lab were merely educational. Since we have a simple means of calculating the correct answer to Problem 2, it doesn't make much sense to use methods of Monte Carlo in this situation. However, as discussed in the previous lab, there are not always closed-form solutions to the integrals we want to compute.

We can extend the same principles we have discussed thusfar to solve many types of problems. For a more general problem, we can implement importance sampling by doing the following:

1. Define a function h where, $h(t) = \begin{cases} 1 & \text{if condition is met} \\ 0 & \text{otherwise} \end{cases}$.
2. Define a function f_X which is the p.d.f. of the target distribution.
3. Define a function g_X which is the p.d.f. of the importance distribution.

Problem 4. The joint normal distribution of N independent random variables with mean 0 and variance 1 is

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

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

antiderivative.

Use what you have learned about importance sampling to estimate the probability that a given random variable in $\mathbb{R}^2$ generated by f_X will be less than -1.75 in the x-direction and greater than 1.25 in the y-direction.

Treat f_X as the p.d.f. of your target distribution. Use the function `stats.multivariate_normal` to create a multivariate normal distribution to serve as your importance distribution. This function accepts a numpy array `mean` and a numpy array `cov`. The parameter `mean` is an array of the mean of each direction. The parameter `cov` is the covariance matrix. In this case, the covariance matrix will be a diagonal matrix with the variance of each variable down the diagonal.

Unnormalized Target Densities

The methods discussed so far are only applicable if the target density is normalized, or in other words, has an integral of 1. If the target density is not normalized, Equation 2.3 becomes,

$$\begin{aligned} E[h(X)] &= \frac{\int h(t)f(t)\,dt}{\int f(t)\,dt} \\ &= \frac{\int h(t)f(t)\left(\frac{g_X(t)}{g_X(t)}\right)\,dt}{\int f(t)\left(\frac{g_X(t)}{g_X(t)}\right)\,dt} \\ &= \frac{\int \left(\frac{h(t)f(t)}{g_X(t)}\right) g_X(t)\,dt}{\int \left(\frac{f(t)}{g_X(t)}\right) g_X(t)\,dt} \\ &= \frac{E\left[\frac{h(Y)f(Y)}{g_X(Y)}\right]}{E\left[\frac{f(Y)}{g_X(Y)}\right]} \end{aligned}$$

The corresponding estimator becomes,

$$\begin{aligned} \widehat{E}_n[h(X)] &= \frac{E\left[\frac{h(Y)f(Y)}{g_X(Y)}\right]}{E\left[\frac{f(Y)}{g_X(Y)}\right]} \\ &= \frac{\frac{1}{N}\sum_{i=1}^{N}\frac{h(y_i)f(y_i)}{g_X(y_i)}}{\frac{1}{N}\sum_{i=1}^{N}\frac{f(y_i)}{g_X(y_i)}} \end{aligned}$$