Lab 18

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 integration is radically different from techniques like Simpson's rule. Whereas Simpson's rule is purely computational and deterministic, Monte Carlo integration uses randomly chosen points in the domain to calculate the integral. Although it converges slowly, Monte Carlo integration is frequently used to evaluate multivariable integrals because the higher-dimensional analogs of methods like Simpson's rule are extremely 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 the area by randomly choosing points in a 2×2 square. The percentage 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 $[0, 1] \times [0, 1]$.

```
>>> N = 500 # Number of sample points
>>> points = np.random.rand(2, N)
```

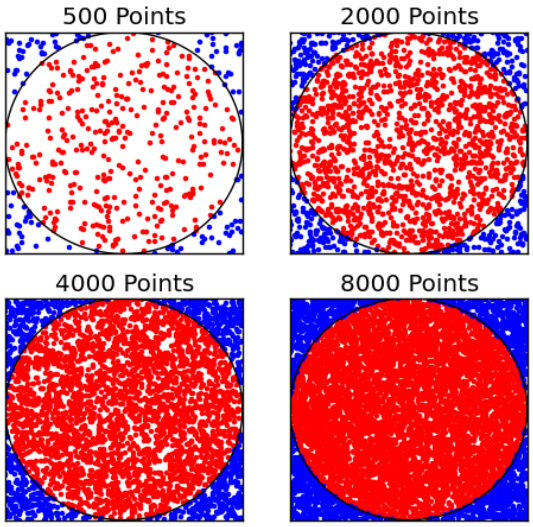

Figure 18.1: Finding the area of a circle using random points

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

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

Next we determine the number of points in the unit circle. We compute the Euclidean distance from the origin for each point, then count the points that are within a distance of 1 from the origin.

```
>>> # Compute the distance from the origin for each point
>>> pointsDistances = np.linalg.norm(points,axis=0)
>>> # Count how many are less than 1
>>> numInCircle = np.count_nonzero(pointsDistances < 1)
```

The fraction of points inside the circle is `numInCircle` divided by `N`. By multiplying this fraction by the square's area, we can estimate the area of the circle.

```
>>> circleArea = 4.*(numInCircle/N)
>>> circleArea
3.024
```

This differs from π by about 0.117.

Problem 1. Write a function that estimates the volume of the unit sphere. Your function should have a keyword argument `N` that defaults to 10^5. Your function should draw `N` points uniformly from $[-1,1]\times[-1,1]\times[-1,1]$ to make your estimate. The true volume is $\frac{4}{3}\pi \approx 4.189$.

Monte Carlo Integration

In the examples above, we drew a bounding box around a volume, then used random points drawn from the box to estimate that volume. This is easy and intuitive when the volume is a circle or sphere. But it's hard to generalize this to any arbitrary integral - in order to draw the box, we have to already know something about the volume we are estimating. Instead, given an arbitrary function $f(x) : \mathbb{R}^n \mapsto \mathbb{R}$ and a region $\Omega \subset \mathbb{R}^n$ in the domain of f, we would like to use random points drawn from Ω to estimate the integral $\int_\Omega f(x)\, dV$, *without* having to specify a bounding box around the volume of integration.

We can estimate this integral using the approximation.

$$\int_\Omega f(x)\, dV \approx V(\Omega)\frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{18.1}$$

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

The intuition behind (18.1) 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.

For further intuition, compare (18.1) to the Average Value Theorem from single-variable calculus. By the Average Value Theorem, the average value of $f(x) : \mathbb{R} \mapsto \mathbb{R}$ on $[a, b]$ is given by

$$f_{avg} = \frac{1}{b-a}\int_a^b f(x)\, dx. \tag{18.2}$$

If we let $\Omega = [a, b]$ in (18.2) (noting that $V(\Omega) = b - a$) and replace f_{avg} with the approximation $\frac{1}{N}\sum_{i=1}^{N} f(x_i)$, then we get precisely the Monte Carlo integration formula in Equation (18.1)!

As it turns out, we can refactor the circle-area problem slightly so that it uses Equation (18.1). Let f be defined by

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

and let $\Omega = [-1, 1] \times [-1, 1]$. The area of the circle is given by $\int_\Omega f(x)\, dV$, which we can estimate with the Monte Carlo integration formula:

$$\text{Area of unit circle } \approx V(\Omega)\frac{1}{N}\sum_{i=1}^{N} f(x_i) = \frac{4}{N}\sum_{i=1}^{N} f(x_i).$$

To summarize, we have the following steps to estimate the integral of any function f over a region Ω in the domain of f:

1. Draw N random points uniformly distributed in Ω.
2. Find the image of each point under f, and take the average of these images.
3. Multiply by the volume of Ω.

Problem 2. Write a function that performs 1-dimensional Monte Carlo integration. Given a function $f : \mathbb{R} \mapsto \mathbb{R}$, an interval $[a, b]$, and the number of random points to use, your function should return an approximation of the integral $\int_a^b f(x)\,dx$. Let the number of sample points default to 10^5. Test your function by estimating integrals that you can calculate by hand.

Problem 3. Generalize Problem 2 to multiple dimensions. Write a function that accepts a function handle f to integrate, the bounds of the interval to integrate over, and the number of points to use. Let the number of sample points default to 10^5. Your implementation should be robust enough to integrate any function $f : \mathbb{R}^n \mapsto \mathbb{R}$ over any interval in $\mathbb{R}^n$.

Hints:

1. To draw a random array of points from the given interval, first create a random array of points in $[0, 1] \times \ldots \times [0, 1]$. Multiply this array by the dimensions of the interval to rescale it, then add the lower bounds of integration to shift it. Think about using array broadcasting.

2. You can use `np.apply_along_axis()` to apply a function to each column of an array. Here is an example of applying a function to points in $\mathbb{R}^2$:

```
>>> points = np.random.rand(2,4)
>>> points
array([[ 0.33144631,  0.52558001,  0.67766158,  0.45570083],
       [ 0.70935864,  0.20985475,  0.25917177,  0.19431292]])
# Apply the norm function to each point
>>> np.apply_along_axis(np.linalg.norm,0,points)
array([ 0.78297275,  0.565927  ,  0.72553099,  0.49539959])
```

   This is especially useful for functions that don't work nicely with array inputs. For example, the code below uses a simple thresholding function `f`; calling `f(points)` would throw an error, but using `np.apply_along_axis` gives the expected result. (Note that `points` must have at least 2 dimensions for this to work.) Refer to the NumPy docs for more information.

```
# Simple function that returns a 0 if x is less than 0.5
>>> f = lambda x: x if x > 0.5 else 0.
# Get 4 random points. The 1 forces the array to be 2-D.
>>> points = np.random.rand(1,4)
>>> points
array([[ 0.2144746 ,  0.02490517,  0.86593995,  0.86401139]])
# Evaluate f at each element of points
>>> np.apply_along_axis(f,0,points)
array([ 0.        ,  0.        ,  0.86593995,  0.86401139])
```

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

Problem 4. 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. This is an important distribution in statistics. However, $f(\mathbf{x})$ does not have a symbolic antiderivative.

1. Let $\Omega = [-1.5, 0.75] \times [0, 1] \times [0, 0.5] \times [0, 1] \subset \mathbb{R}^4$. Use the function you wrote in Problem 1 to integrate $f(\mathbf{x})$ on Ω. Use 50000 sample points.

2. SciPy has a built in function specifically for integrating the joint normal distribution. The integral of $f(\mathbf{x})$ on $B = [-1, 1] \times [-1, 1] \times [-1, 1] \subset \mathbb{R}^3$ can be computed in SciPy with the following code.

```
>>> from scipy import 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
>>> means = np.zeros(3)

# The covariance matrix of N independent random variables
#    is the NxN identity matrix.
>>> covs = np.eye(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 Ω.

3. Return your Monte Carlo estimate, SciPy's answer, and (assuming SciPy is correct) the relative error of your Monte Carlo estimate.

Convergence

The error of the Monte Carlo method is proportional to $1/\sqrt{N}$, where N is the number of points used in the estimation. 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. The error converges at the same rate whether integrating a 2-

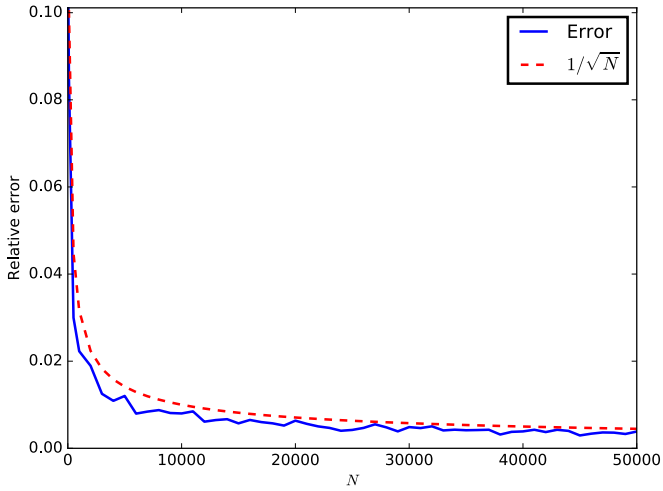

Figure 18.2: The Monte Carlo integration method was used to compute the volume of the unit sphere. The blue line plots the average error in 50 runs of the Monte Carlo method on N sample points. The red line is a plot of $1/\sqrt{N}$.

dimensional or a 20-dimensional function. This gives Monte Carlo integration an advantage over other methods, and makes it especially useful for estimating integrals in high dimensions.

Problem 5. In this problem we will visualize how the error in Monte Carlo integration depends on the number of sample points.

Run Problem 1 with `N` equal to 50, 100 and 500, as well as 1000, 2000, 3000, ..., 50000; having some additional small values of `N` will help make the visualization better.

For each value of `N`:

1. Estimate the volume of the unit sphere using Problem 1, and use the true volume to calculate the relative error of the estimate.

2. Repeat this multiple times to get an average estimate of the relative error. Your function should accept a keyword argument `numEstimates` that defaults to 50.

3. Calculate and store the mean of the errors.

Plot the mean relative error as a function of `N`. For comparison, plot the function $1/\sqrt{N}$ on the same graph. Your plot should resemble Figure 18.2).

A Caution

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

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

with Monte Carlo 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 Monte Carlo integration fails to converge.