Introducing AI-DS-ML Skills

Python Programming

Chapter 13: Monte Carlo Simulation

By Fred Annexstein, PhD

PYTHON PROGRAMMING

13. MONTE CARLO SIMULATION

BY FRED ANNEXSTEIN, PHD

CHAPTER 13 OVERVIEW

In this module we will be covering an important algorithmic method for data science called the Monte Carlo method. Monte Carlo methods rely on access to a fast and statistically sound random number generators. Python provides such a generator in the random module. We will consider a variety of problems that we can apply Monte Carlo methods to achieve accurate simulation and estimation results.

CHAPTER 13 OBJECTIVES

By the end of this chapter, you will be able to... .

Understand the use cases of the Monte Carlo methods

Understand the reliance of such methods on effective random number generators.

Understand how to use seed values for reproducibility.

Understand how to use Monte Carlo methods for various ball and bin simulations.

PSEUDO-RANDOM NUMBERS

Several Python libraries will produce

pseudorandom number generators (PRNG). The standard modules random and numpy have implementations of efficient pseudo-random number generators for various distributions. For integers, there is a generator for uniform selection from a range. For sequences, there is generator for uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement, as others.

Almost all module functions depend on the basic function random(), which generates a random float uniformly in the range [0.0, 1.0). Python uses the Mersenne Twister algorithm as the core generator. This produces 53-bit precision floats and has an enormous period of 2**19937-1. The Mersenne Twister is one of the most extensively tested random number generators in existence.

A COMPUTATIONAL EXPERIMENT

Let us run an experiment with random number generator random(). We will compute the mean and the variance, which is the average of the squared deviations from the theoretical mean of 0.5. Our experiment uses N=10,000 random samples and calculates the sample mean and sample variance.

```
import random
sum, sumdev = 0.0,0.0
N= 10000
for i in range(N):
    next = random()
    sum += next
    dev = (next - 0.5)**2
    sumdev += dev
mean = sum / N
print(f"Mean:{mean}")
var = sumdev/(N-1)
print(f”Variance:{var}")

Output:
Mean:0.49933762112884134
Variance:0.08385481699150024
```

Theory tells us that for a standard uniform distribution U(0,1) of a random variable between 0 and 1, the expected value is 1/2 and the variance is 1/12. These are close to the computed sample values.

SEEDING THE RANDOM-NUMBER GENERATOR FOR REPRODUCIBILITY

The function random() is deterministic and generates pseudorandom numbers, based on an internal calculation that begins with a numeric value known as a seed. Repeatedly calling random produces a sequence of numbers that appear to be random, because each time you start a new interactive session or execute a script that uses the random() function, Python internally uses a different seed value. When you are debugging logic errors in programs that use randomly generated data, it can be helpful to use the same sequence of random numbers until you have eliminated the logic errors, before testing the program with other values.

To do this, you can use the random module's seed function to seed the random-number generator yourself—this forces random() to begin calculating its pseudorandom number sequence from the seed you specify.

In the next example, we use the int() function along with random() to generate a random integer number between 1 and 6 to simulate a dice roll. The output will be the same for each run, since we fixed the seed value.

```
from random import random,seed
seed(43)
for roll in range(20):
    print(1 + int(random()*6) , end=' ‘)

Output:
1 5 1 3 5 5 3 3 1 3 3 6 4 5 3 2 6 5 3 1
```

MONTE CARLO ESTIMATION OF PI

One simple demonstration of the Monte Carlo algorithm is the estimation of the number Pi. Here we use the idea of sampling random (x, y) points in a 2-D unit square using a standard uniform distribution. All the points in this square whose distance from the origin is at most 1.0 lies inside or on top the quarter-circle. For a Monte Carlo estimation of pi we can calculate the ratio of the number points that fall inside the circle versus the total number of generated points. This ratio will tend to pi/4 since we are using a quarter of a circle with radius 1, that is quarter of a circle whose area is Pi. Here is the code to carry out this Monte Carlo estimation:

```
from random import random

N = 10000
circle_hits = 0
for i in range(N):
  x,y = random(), random()
  if x**2 + y**2 <=1:
    circle_hits +=1


pi_est = 4 * circle_hits / N
print("Final Estimate of Pi=", pi_est)

Output:
Final Estimation of Pi= 3.1424
```

MONTE CARLO INTEGRATION

Integration or summing the area under a curve lends itself to solution by Monte Carlo techniques. This is possible since we can view integration as calculating the average value of a function along an interval [a,b] and then multiplying by the length of the interval which is simply b-a.

Let us consider integrating the sin() function. In the following code we choose a large number of random x values in the interval [a,b] and compute the sum of the function values at those points. This large sum yields an average value for the function, and thus provides the integral.

The example below estimates the integral of the sin() function for the range between a=0 and b=np.pi =3.141592653589793. Since math says that integrating sin is -cos, and so the Monte Carlo estimate shows correctness to 3 decimal places.

```
import numpy as np
from random import random

# limits of integration are a, b
a = 0
b = np.pi # gets the value of pi
N = 10000

integral = 0.0
```

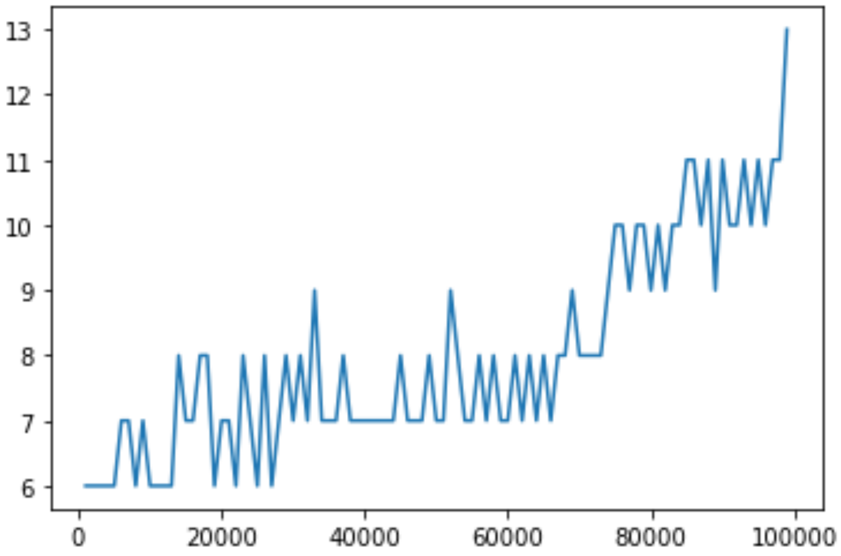
```
def f(x):
    return np.sin(x)

for i in range (N):
    x  = a + (b−a)∗rng()
    integral += f(x)

ans = (b−a) ∗ integral/N
print (f"{ans} is the estimated value.”)

Output:
2.0040349814696534 is the estimated value.
```

BALLS AND BIN ESTIMATION

There are many examples in Computer Science application we model collision problem. It is popular to use what is called balls and bins analysis to investigate such collision problems. Let us consider a container with N bins, and we throw balls at random, each ball falling into any one bin with equal chance. We would like to know how many balls will be in the largest bin after throwing N balls at random. Surely the number is greater than 1 and less than N, but where in that range is the answer likely to be.

We run a Monte Carlo simulation to answer that question. In the following code we run experiments for each N up to 1,000,000. To speed up the simulation we only pick values of N divisible by 100. We will produce a line plot of the results. To do this we store a list maxbin values for the size of the largest bin in each experimental value of N.

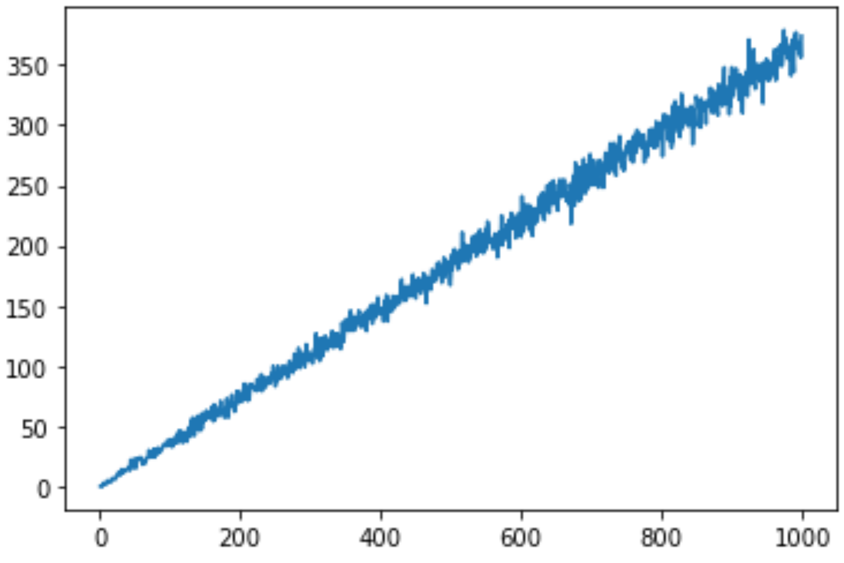
```
import numpy as np
from random import random

balls = np.arange(100,100000,100)
maxbin = []
for N in balls:
  bins = np.zeros(N)
  for b in range(N):
    bins[int(N * random())] +=1
  maxbin.append(max(bins))

import matplotlib.pyplot as plt
plt.plot(balls, maxbin)
plt.show()
```



We see from the output that throwing N balls into N bins never causes any bin to be loaded with more than 13 balls, and the larges bins are never smaller than 6 balls.

MODULE 13 LAB

For this lab you will run a Monte Carlo simulation for a different balls and bins problem. For this problem we are interested in knowing the number of empty bins when N balls are thrown at random into N bins. For this assignment you are to write code to run a simulation for each N from 1 to 1000, and plot the results using the matplotlib plot function. When you do this you should see a striking linear relationship in the output.



We would like to know the slope of this

this linear relationship. Write code to run a linear regression to find the best fit line for your dataset. Your output should be similar to the following:

```
SciPy Linear Regression Solution
 slope: 0.36837836634229415
 intercept: -0.02602000798390236
 rvalue: 0.997749733854947
```

Notice that the rvalue is very close to 1, and the slope is very close to the number 1/e = 0.36787944117. We leave this as an optional exercise to show why that is so.