Lab 3: Monte Carlo

Monte Carlo (MC) simulation is another of the key tools of financial modelling. In this lecture we will:

- apply MC simulation to the simple problem of estimating the **area of a circle and the value of pi** using both classical and quantum approaches
- explore how classical and quantum MC algorithms can be used for other type of problems such as **numerical integration**

Monte Carlo simulation is a tool that allow us to naturally integrate uncertainty into financial calculations (Glasserman, 2003). Its basic idea is very simple and can be represented in three steps:

Step 1: generate a number of scenarios for some unknown variable,

Step 2: for each scenario evaluate the price of the financial instrument under consideration,

Step 3: average the prices of the financial instruments to obtain an estimate of its price.

The Problem: Area of a Circle with Monte Carlo

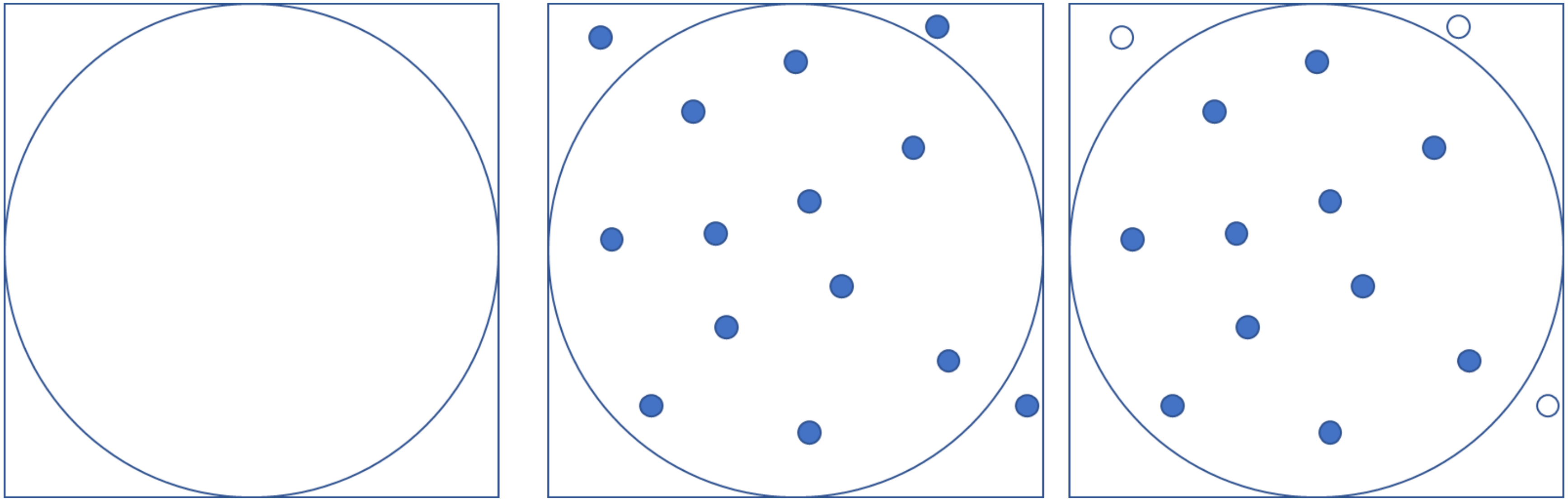
π

An Example from Math: estimating π

Imagine you would like to estimate $\boldsymbol{\pi}$. How can you do it?

Well, we can do it using Monte Carlo simulation.

The strategy would involve using random samples of a unit square, which can be imagined as throwing stones into a circle inscribed in a square and then counting the number of stones that landed inside the circle to the total amount of stones thrown , and that will give us an estimate of the proportion of the area of the circle to the area of the square as illustrated in the figure:



Monte Carlo sampling

$$\frac{Area\ Circle}{Area\ Square} = \frac{\pi r^2}{(2r)(2r)} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

So we can approximate it as an expectation via samples using the stones as:

$$E\left[\frac{Area\ Circle}{Area\ Square}\right] = \frac{\pi}{4} \approx \frac{Stones\ in\ Circle}{Stones\ in\ Square}$$

$$\pi \approx 4 \times \frac{Stones\ in\ Circle}{Stones\ in\ Square}$$

In the figure, we have used 14 samples or stones and we have thrown them into the square. Of these 11 have landed inside the circle. Thus out estimate corresponds to:

$$\pi \approx 4 \times \frac{11}{14} = 3.1428$$

Which happens to be close to the true value of π. Please see Sutor (2019) for an excellent alternative explanation of this problem in the context of quantum computing.

The Classical Solution

The Classical Solution

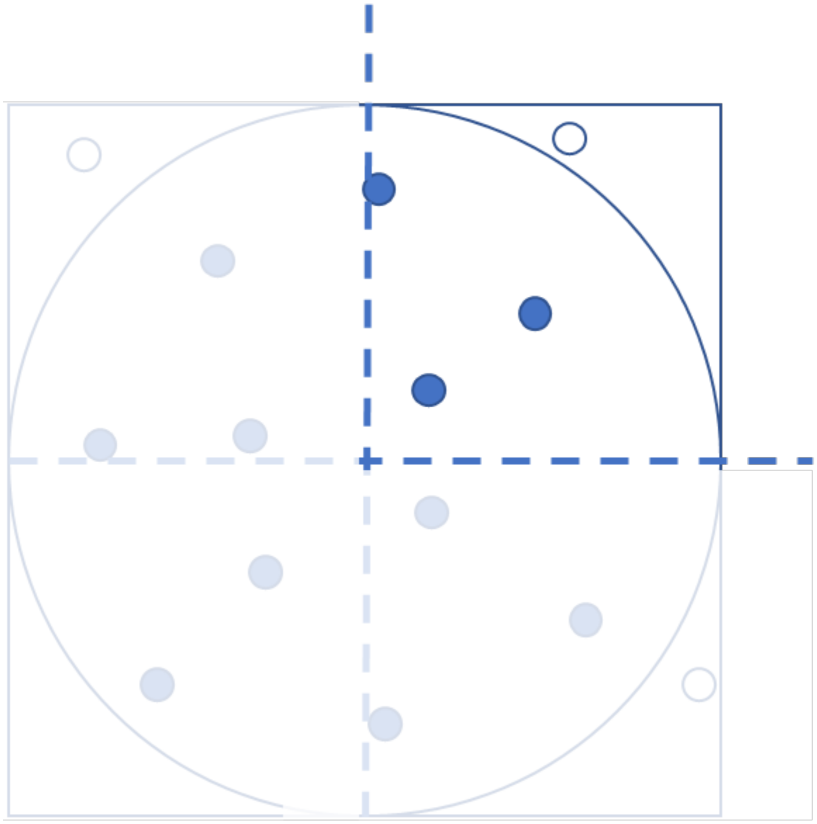
As before, the steps required in order to do in Monte Carlo simulation would be the following:

first, generate a series of random numbers following some specific rule (in our case, numbers that would represent the *xy* coordinates inside the square),

second, we would apply deterministic operation of determining if each set of coordinates represent a location inside or outside the circle;

finally, we can average these numbers to obtain an estimate of the area of the circle.

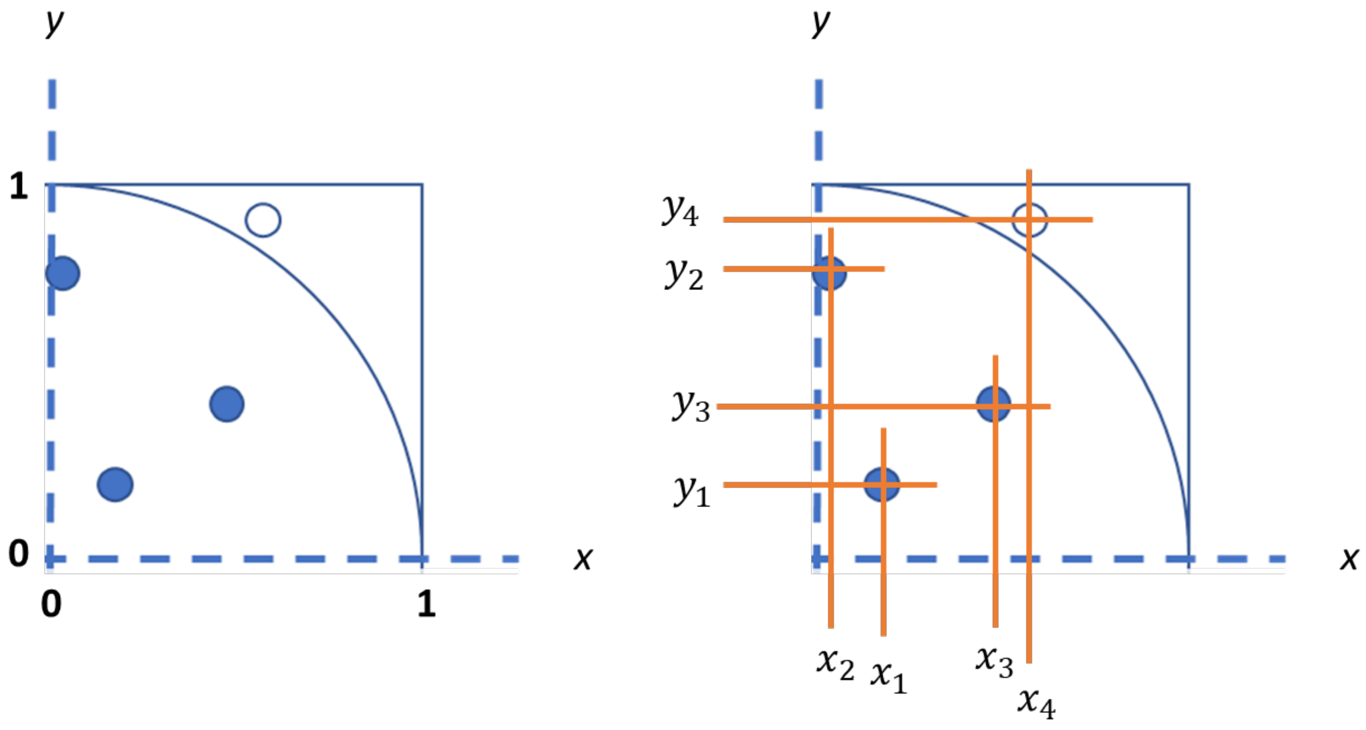
In the following analysis, due to symmetry considerations, we can concentrate only on one quarter of the domain as illustrated in the figure.



The domain for sampling in the MC problem

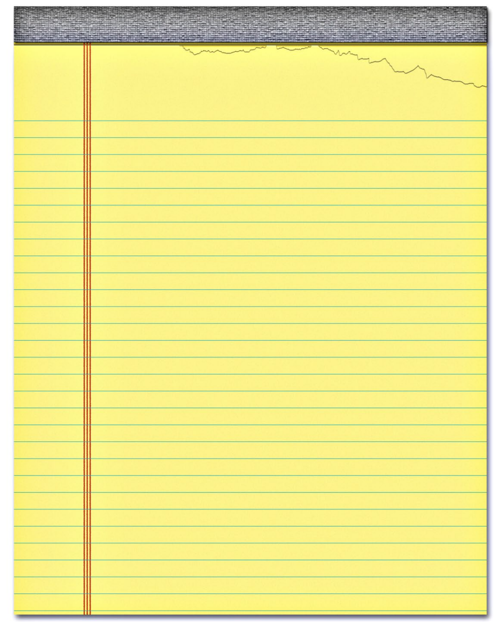
Step 1: Generate random scenarios

We generate two sets of independent random numbers from a uniform distribution, one representing the x coordinate and the other representing the y coordinate. The pair (x,y) will represent a sample coordinate, i.e. the throwing of the stone.



So we have four samples:

$$(x_1, y_1) = (0.20, 0.25)$$
$$(x_2, y_2) = (0.05, 0.65)$$
$$(x_3, y_3) = (0.50, 0.45)$$
$$(x_4, y_4) = (0.55, 0.90)$$



Step 2: Evaluate each scenario

For each sample (coordinate pair) we calculate the radius in order to determine if its higher than 1 (outside the circle) or less than 1 (inside the circle).

$$r_1 = f(x_1, y_1) = \sqrt{{x_1}^2 + {y_1}^2} = \sqrt{0.20^2 + 0.25^2} = 0.32 < 1$$
$$r_2 = f(x_2, y_2) = \sqrt{{x_2}^2 + {y_2}^2} = \sqrt{0.05^2 + 0.65^2} = 0.65 < 1$$
$$r_3 = f(x_3, y_3) = \sqrt{{x_3}^2 + {y_3}^2} = \sqrt{0.50^2 + 0.45^2} = 0.67 < 1$$
$$r_4 = f(x_4, y_4) = \sqrt{{x_4}^2 + {y_4}^2} = \sqrt{0.55^2 + 0.90^2} = 1.05 > 1$$

Thus three samples fall inside the circle and one sample is outside.

Step 3:: Average across the scenarios to obtain the estimate

Finally we compute the expectation that will give us our estimate for pi.

$$\pi \approx 4 \times \frac{3}{4} = 3$$

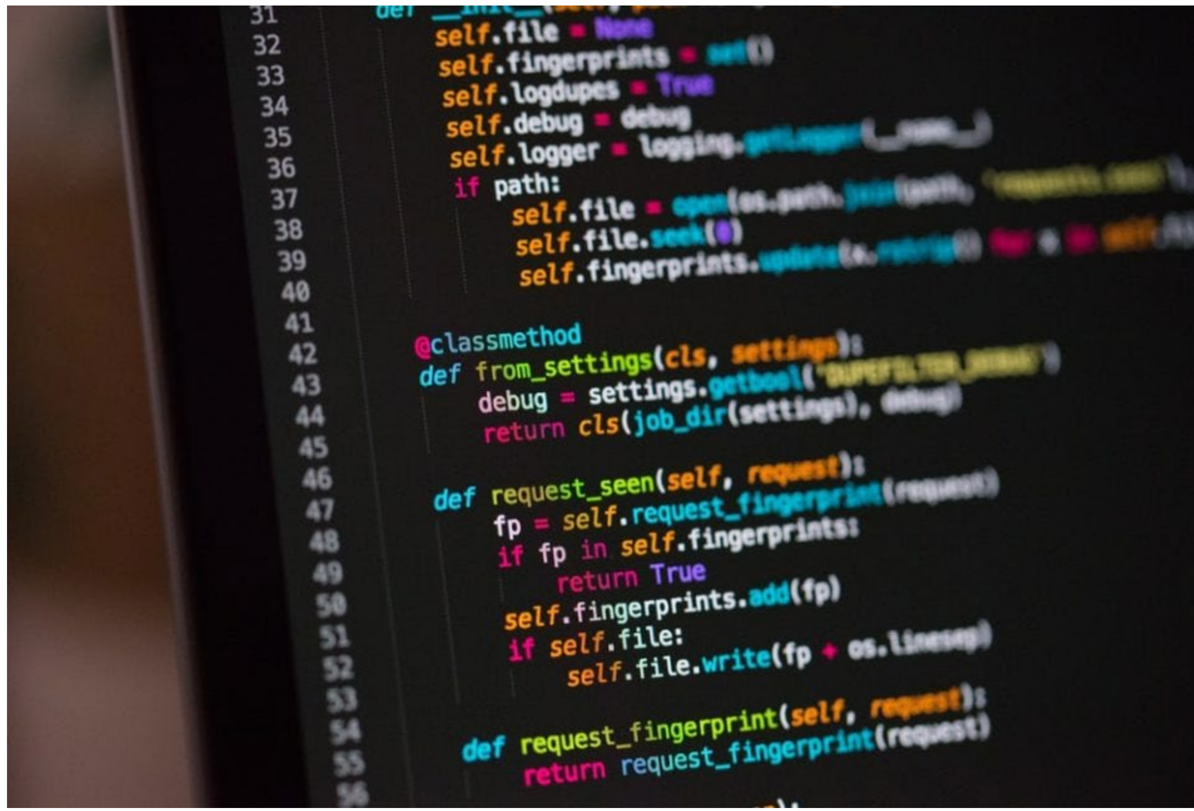
Clearly our estimate is not very good as we have used a very small number of samples. What we would like to do now is to throw more stones. In the following we will illustrate how we can implement this approach using classical tools from Python.

Python Lab

Python Lab

We are going to re-use some of the material from Lecture 2 about generating random numbers. In particular, we will make use of the module **random**. In the following demonstrations we will use this module to generate random numbers in a classical computer.

For more details see



https://docs.python.org/3/library/random.html

LABORATORY 1: Classical Monte Carlo π Estimate

In this laboratory we will implement the MC algorithm discussed above to estimate the value of $\boldsymbol{\pi}$.

Code 3.1 Classical Monte Carlo π Estimate

In the following code, we present the use of method random() from module **random** to generate random numbers from a uniform distribution. These will represent the x and y coordinates that we ought to generate.

```
# CODE_3_1_CLASSICAL_MONTECARLO_PI_ESTIMATE

# import the required libraries
import matplotlib.pyplot as plt
import random as rn
import numpy as np
import math as m
rn.seed(12345) # set seed of random sequence to 12345

N = 100 # number of simulations
count = 0 # initialize number of samples inside circle quadrant
x_vector = np.zeros(N)
y_vector = np.zeros(N)
r_vector = np.zeros(N)

# Step 1: generate random scenarios
for i in range(N):
    x_vector[i] = rn.random()
    y_vector[i] = rn.random()

# Step 2: operations on each scenario
for i in range(N):
    r_vector[i] = m.sqrt(x_vector[i] ** 2 + y_vector[i] ** 2)
    if r_vector[i]<1: count=count+1

# Step 3: calculate output estimates
# plot individual radii (first 100)
print(r_vector[0:100])
plt.plot(r_vector[0:100])
plt.show()

#estimate pi
pi_approx = 4 * count / N
print('Pi estimate: \t%.4f' % pi_approx)
```

```
[0.41674396 0.87758279 0.41621135 0.58864947 0.45041766 0.58849631
 0.65727358 0.96240467 1.06187506 0.52526007 0.74364163 0.34236935
 1.02532659 1.2297851  0.94024485 1.16293624 0.20468513 0.97775866
 0.93634225 0.5729459  1.0879247  0.83989111 0.89960058 0.97527538
 0.61724572 1.08977331 0.16931297 1.28818842 0.27399739 0.56152744
 0.32749564 0.71600047 1.35524405 1.10402076 0.77193512 0.90036699
 0.62911961 0.69161318 1.0413508  0.79484621 0.8931691  0.18334796
 0.99439952 0.33145244 0.74732622 0.00460923 0.52938941 1.00329481
 0.51418436 0.88868699 1.04176939 0.91153424 0.33158957 0.43304234
 0.85493354 0.48243918 0.76439789 0.46545371 0.6512138  0.24288121
 0.95302502 0.61250718 0.96489156 0.8396426  0.56097254 0.77233063
 0.63294664 0.79991571 0.74707389 0.79491952 0.52118216 0.87993424
 0.26674937 0.98351443 0.92116795 0.93438225 0.63109368 0.43661065
 0.96663057 1.11463158 0.31016855 0.78520329 0.20662806 0.8499038
 0.42451797 0.52641452 0.86166795 0.32194579 0.17882961 0.60602802
 1.04151561 0.9021824  0.5355271  0.85671472 0.66541445 0.79244898
 0.79275403 0.79865618 0.41531372 1.0919962 ]
```

Figure2: output of code 3.1: 100 estimates of the radius

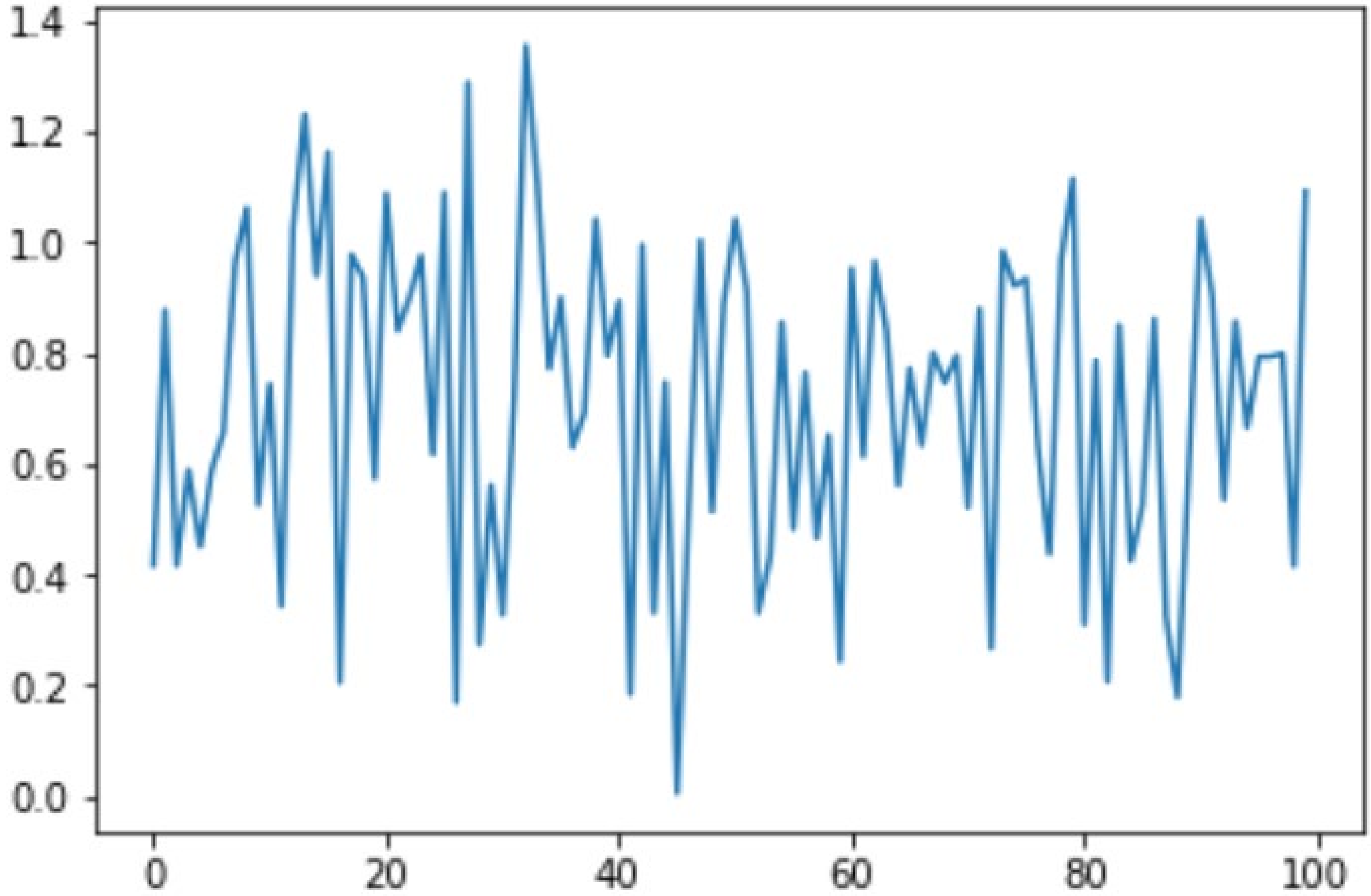


Figure 3: graphical representation of the 100 estimates for r from the table

Code 3.2 Classical Monte Carlo Pi Estimate Convergence

We are going now to take the code 3.1 and put it into a function called *classical_pi_estimate(N)* that depends on the number of simulations N. We are then going to explore what happens by increasing the number of simulations in terms of convergence. This is illustrated in the code 3.2, as described below.

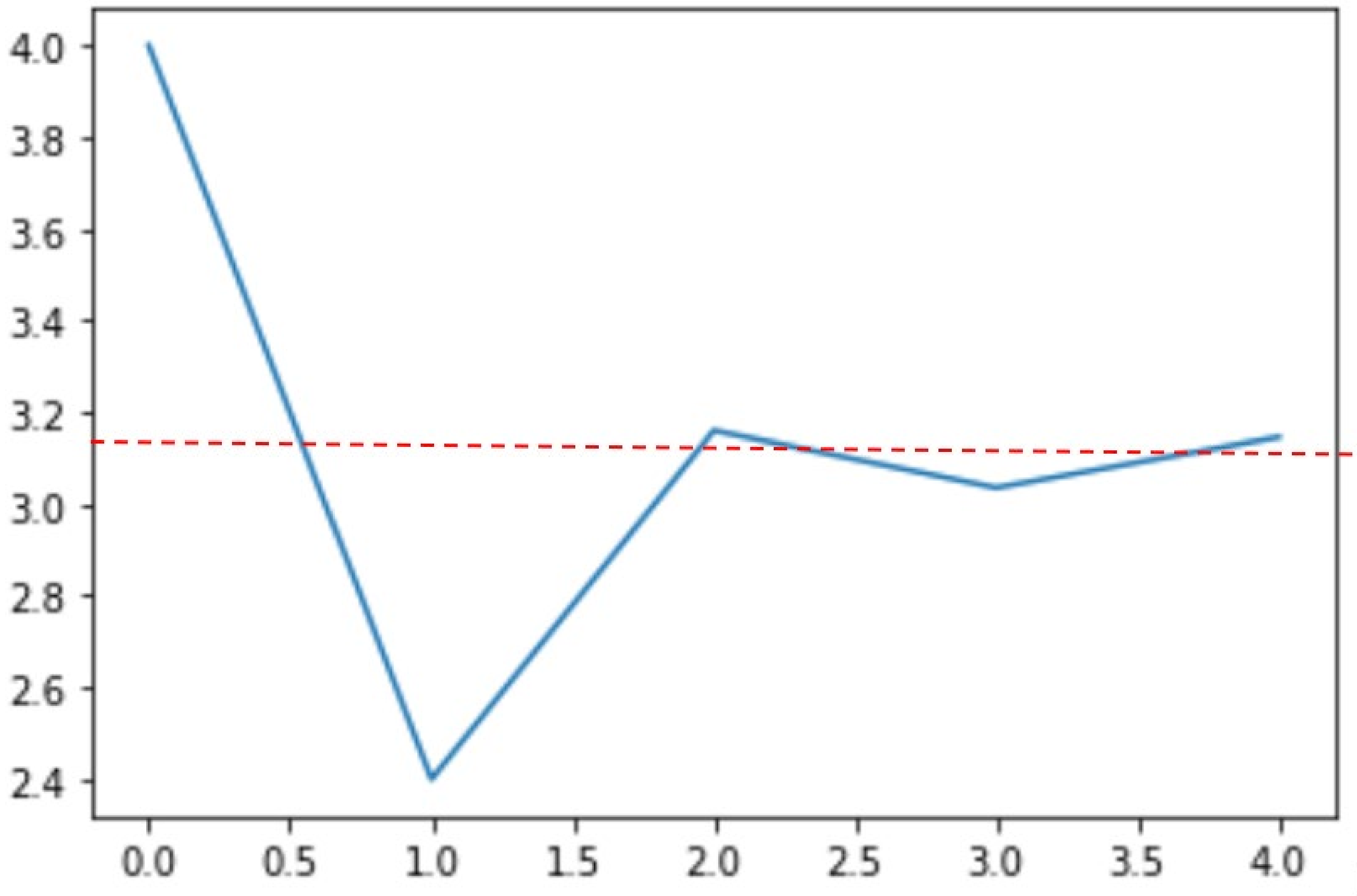
```
# CODE_3_2_CLASSICAL_MONTECARLO_PI_CONVERGENCE
```

Running the code above in Jupyter Lab generates the results illustrated below. Here we see the first 100 integer random numbers from the set generated.

```
1        4.0000
10       2.4000
100      3.1600
1000     3.0360
10000    3.1460
```

OUTPUT code 2.2: increasing number of simulations and resulrting improved estimates of pi

The numerical results can be also visualized in terms of a plot graph.



4.0
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

The Quantum Solution

The Quantum Computing Solution

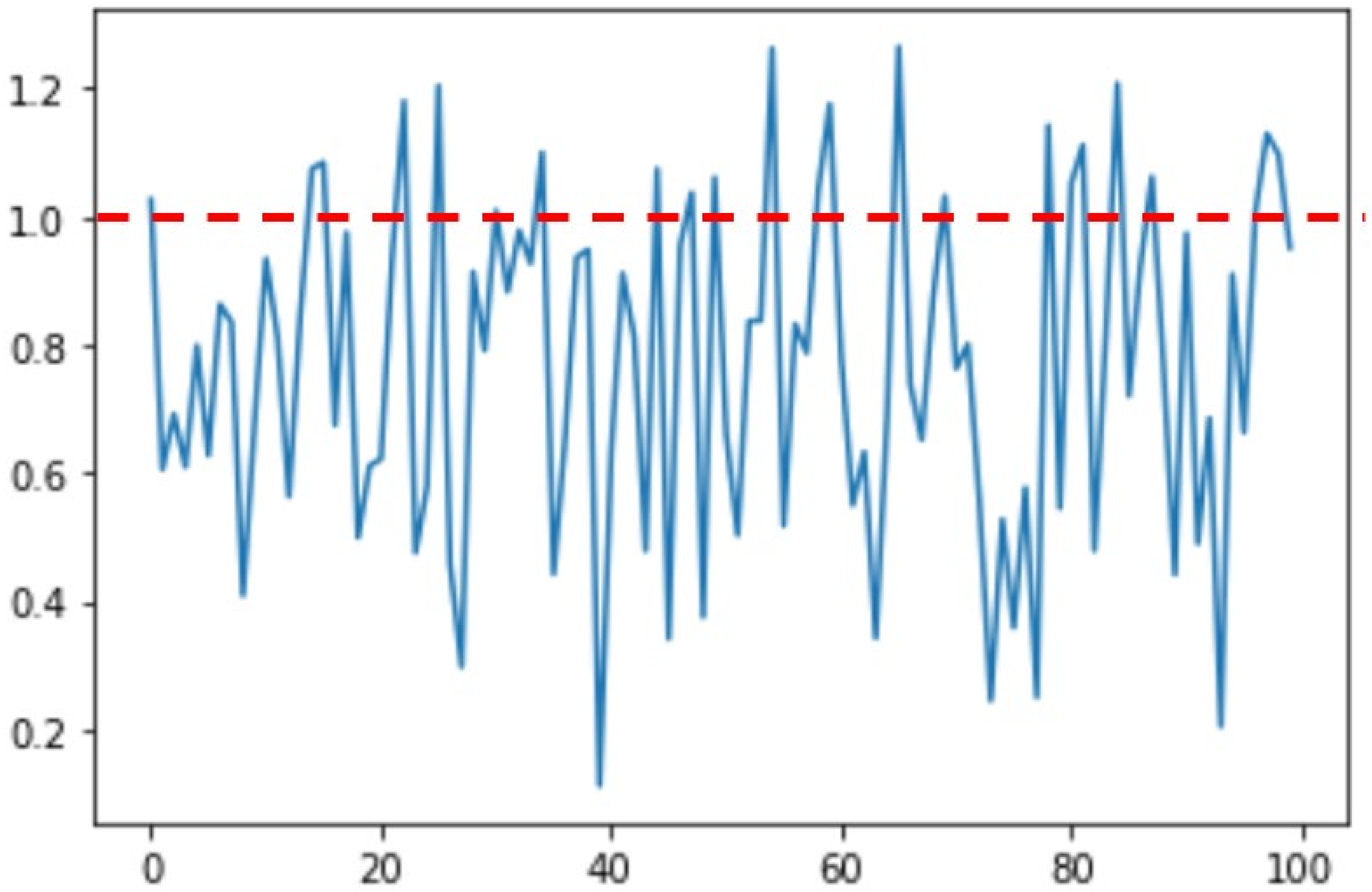
In this section we will repeat the two problems explored before but using random numbers generated using quantum computing. As you will see, the structure of the algorithms is fundamentally identical but now we use quantum circuits to generate high quality random numbers. For this purpose, we will use some of the quantum computing functions we developed in Chapter 2, in particular the *uniform_8bits()* function.

LABORATORY 3: Quantum Monte Carlo Pi Estimate

In this lab, we will repeat the method to estimate the value of pi but now we will do it using random numbers obtain from a quantum computer. What do we need to change in order to do this? The answer is very little. In fact, the only change required is in the source of the random numbers to be used.

Code 3.5 Quantum Monte Carlo Pi Estimate in QISKIT

```
# CODE_3_5_QUANTUM_MONTECARLO_PI_ESTIMATE
```

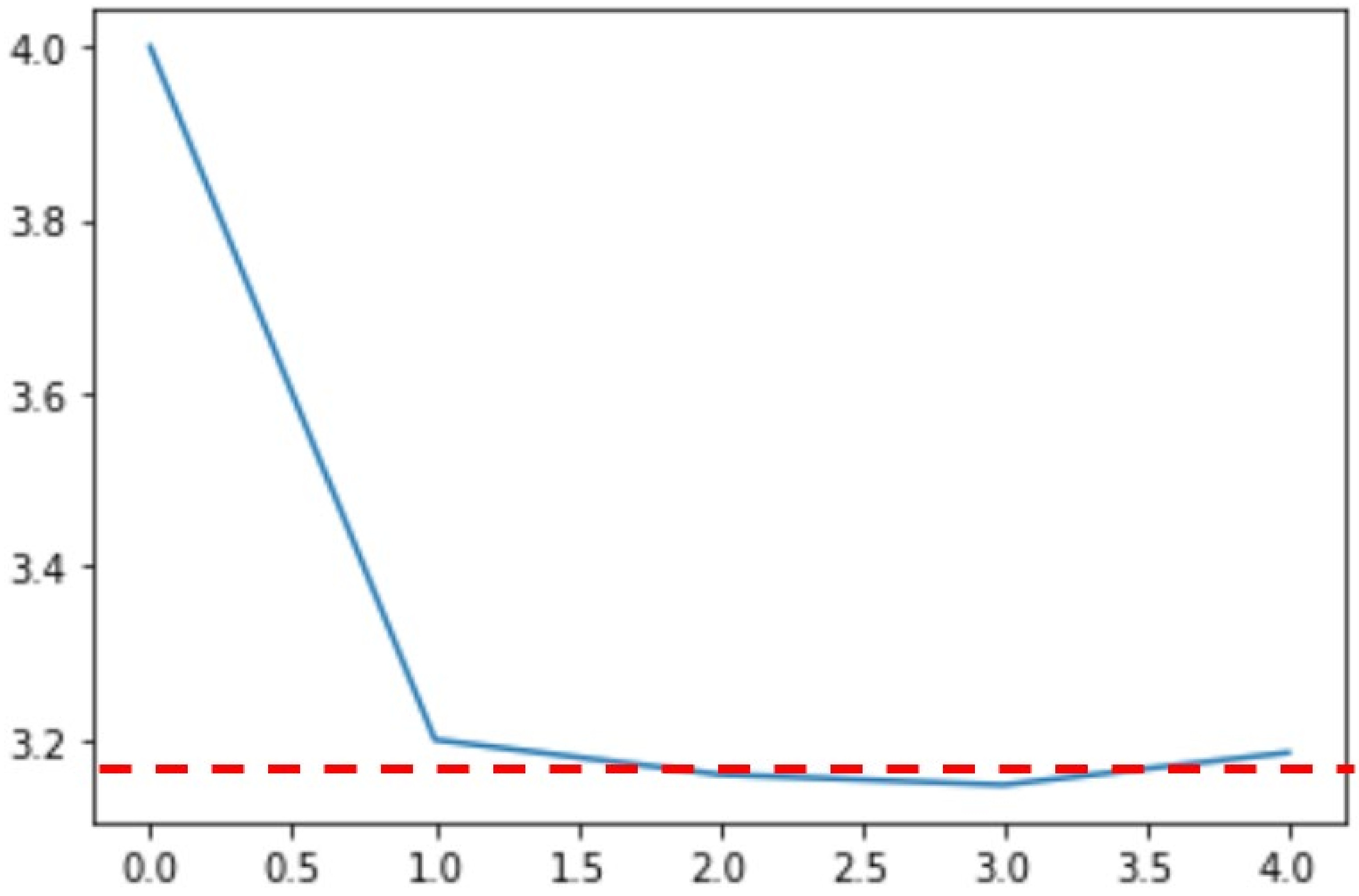


OUTPUT Code 3.5: 100 samples of radius r. The red dashed line indicated the condition inside the circle or outside the circle.

Code 3.6 Quantum Monte Carlo Pi Estimate in QISKIT Convergence

Let’s now explore what would happen if we increase the number of simulations, or the number of shots, in our analysis. We do this by putting Code 3.5 into a function and running it several times for an increasing number of shots.

```
# CODE_3_6_QISKIT_MONTECARLO_PI_CONVERGENCE
```



OUTPUT Code 3.6: results

So what? . . .


GODZILLA vs. KONG

COMPARISON: Classical versus Quantum

In the above results it is difficult to distinguish if quantum computing offers a distinct advantage in terms of accuracy with respect to classical computing. As a first example, we have done a simple comparison between the convergence of the algorithms to estimate pi, i.e. Code 3.2 versus Code 3.6. In the following figure we can observe two consecutive runs of the algorithms obtained the data collected in Code 3.8.

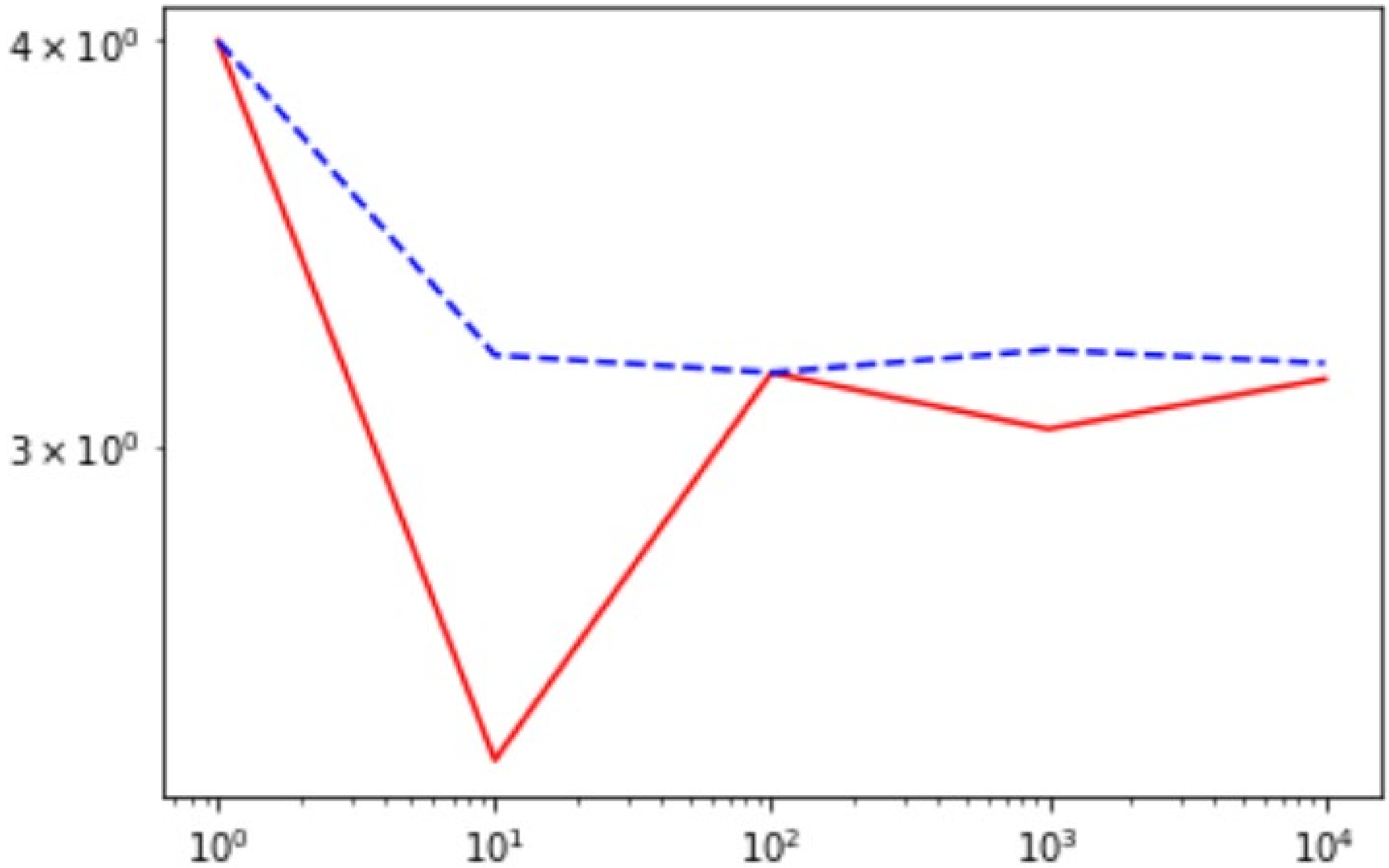


Figure: Code 3.8 convergence comparison between classical (red continuous) and quantum (blue dashed) algorithms to estimate pi.

References

Glasserman, Paul. Monte Carlo Methods in Financial Engineering. Springer, 2003.

Jaeckel, Peter. Monte Carlo Methods in Finance. Wiley, 2002.

Pena, Alonso. Advanced Quantitative Finance with C++. Packt Publishing, 2014.

Montanaro A, Quantum speedup of Monte Carlo methods, Proc. Roy. Soc. Ser. A, vol. 471 no. 2181, 20150301, 2015.

Robert S. Sutor , Dancing with Qubits: How quantum computing works and how it can change the world, Pack Publishing, 2019.