Monte Carlo Method (History)

In the 1930s, **Enrico Fermi** first experimented with the Monte Carlo method while studying neutron diffusion, but he did not publish this work.[17]

In the late 1940s, **Stanislaw Ulam** invented the modern version of the Markov Chain Monte Carlo method while he was working on nuclear weapons projects at the Los Alamos National Laboratory.

Immediately after Ulam's breakthrough, **John von Neumann** understood its importance. Von Neumann programmed the ENIAC computer to perform Monte Carlo calculations. In 1946, nuclear weapons physicists at Los Alamos were investigating neutron diffusion in fissionable material.[17] Despite having most of the necessary data, such as the average distance a neutron would travel in a substance before it collided with an atomic nucleus and how much energy the neutron was likely to give off following a collision, **the Los Alamos physicists were unable to solve the problem using conventional, deterministic mathematical methods**. Ulam proposed using random experiments. He recounts his inspiration as follows:

John von Neumann developed a way to **calculate pseudorandom numbers**, using **the middle-square method**.

A colleague of **von Neumann and Ulam**, **Nicholas Metropolis**, suggested using the name Monte Carlo, which refers to the **Monte Carlo Casino in Monaco** where Ulam's uncle would borrow money from relatives to gamble.

Monte Carlo methods were central to the simulations required for the **Manhattan Project.**

https://en.wikipedia.org/wiki/Monte_Carlo_method

Monte Carlo Method
(in a simplistic term)

Sometimes calculating this probability can be mathematically complex or highly intractable. But we can always run a computer simulation to simulate the whole game many times and see the probability as the number of wins divided by the number of games played.

What is the probability that you will get <H H T T> in 4 coin flips (fair coin)? The order matters.

Calculation

$1/16 = ½ * ½ * ½ * ½$

Since the order matters.

What is the probability that you will get {H, H, T, T} in 4 coin flips (fair coin)? The order does not matter.

Calculation

H	H	T	T
T	T	H	H
T	H	H	T
H	T	T	H
H	T	H	T
T	H	T	H

6 (= 4! / (2!*2!)) many cases to worry of.

$P = 6 * 1/16$

Since the order does not matters.

MC → Just do it 1000 times and see what happens → You do need a counting strategy.

Pseudorandom generator: Middle-square method

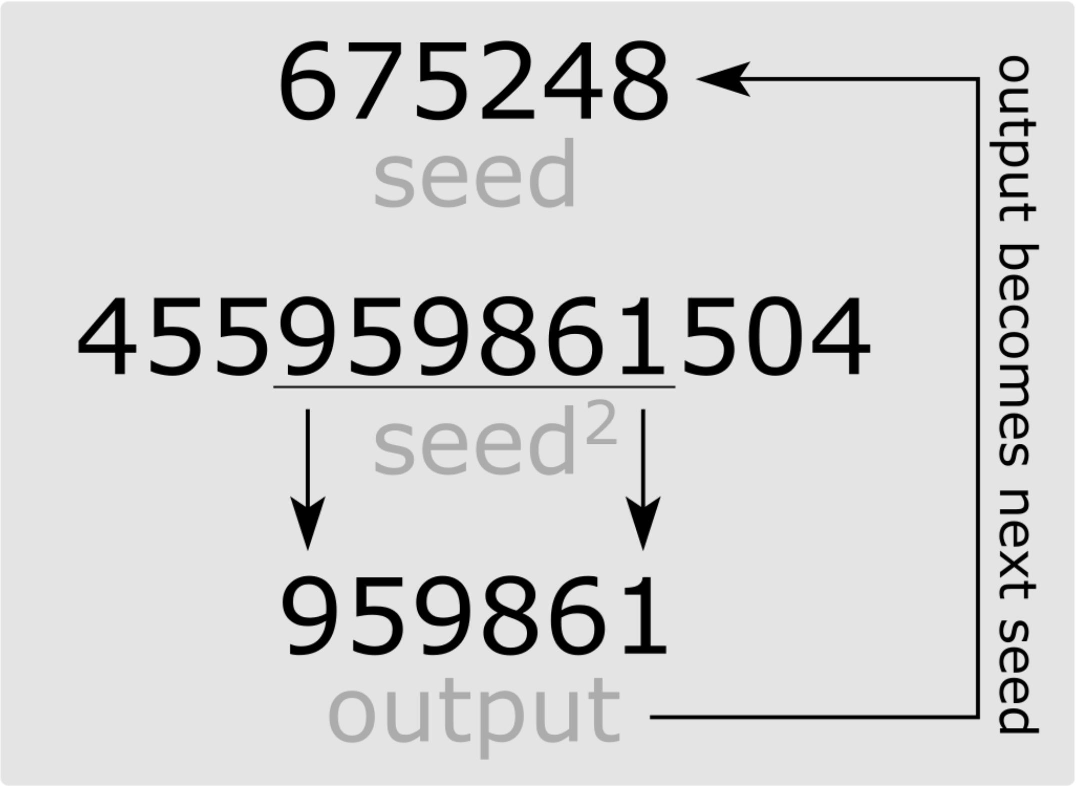
The middle-square method is a method of generating pseudorandom numbers. In practice **it is not a good method**, since its period is usually very short and it has some severe weaknesses; repeated enough times, the middle-square method will either begin repeatedly generating the same number or cycle to a previous number in the sequence and loop indefinitely.

```
seed_number = int(input("Please enter a four digit number:\n[####] "))
number = seed_number
already_seen = set()
counter = 0

while number not in already_seen:
   counter += 1
   already_seen.add(number)
   number = int(str(number * number).zfill(8)[2:6])  # zfill adds padding of zeroes
   print(f"#{counter}: {number}")

print(f"We began with {seed_number}, and"
     f" have repeated ourselves after {counter} steps"
     f" with {number}.")
```

Please enter a four digit number:
[####] 7395
#1: 6860
#2: 596
#3: 3552
…

https://en.wikipedia.org/wiki/Middle-square_method

Six middle digits



Speed is the key: Nevertheless he found **these methods hundreds of times faster** than reading "truly" random numbers off punch cards, which had practical importance for his ENIAC work.

Now → Cryptographically secure PRNG

random — Generate pseudo-random numbers

Source code: Lib/random.py

Computers don't generate truly random numbers—they are deterministic, meaning they operate by a set of rules.

Bookkeeping functions

random.seed(a=None, version=2)
Initialize the random number generator.

If a is omitted or None, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the os.urandom() function for details on availability).

Real-valued distributions

random.random()
Return the next random floating point number in the range [0.0, 1.0).

random.uniform(a, b)
Return a random floating point number N such that a <= N <= b for a <= b and b <= N <= a for b < a.

random.gauss(mu, sigma)
Normal distribution, also called the Gaussian distribution. mu is the mean, and sigma is the standard deviation. This is slightly faster than the normalvariate() function defined below.

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

random.py

```
import random

random.seed(10)
print(random.random())

random.seed(10)
print(random.random())
```

0.5714025946899135
0.5714025946899135

```
import random

random.seed(10)
print(random.random())

random.seed()
print(random.random())
```

0.5714025946899135
0.8364074283863471

If a is omitted or None, the current system time is used.

https://www.w3schools.com/python/ref_random_seed.asp

Plot a graph to observe the gaussian distribution.

```
# import the required libraries
import random
import matplotlib.pyplot as plt

# store the random numbers in a
# list
nums = []
mu = 100
sigma = 50
```

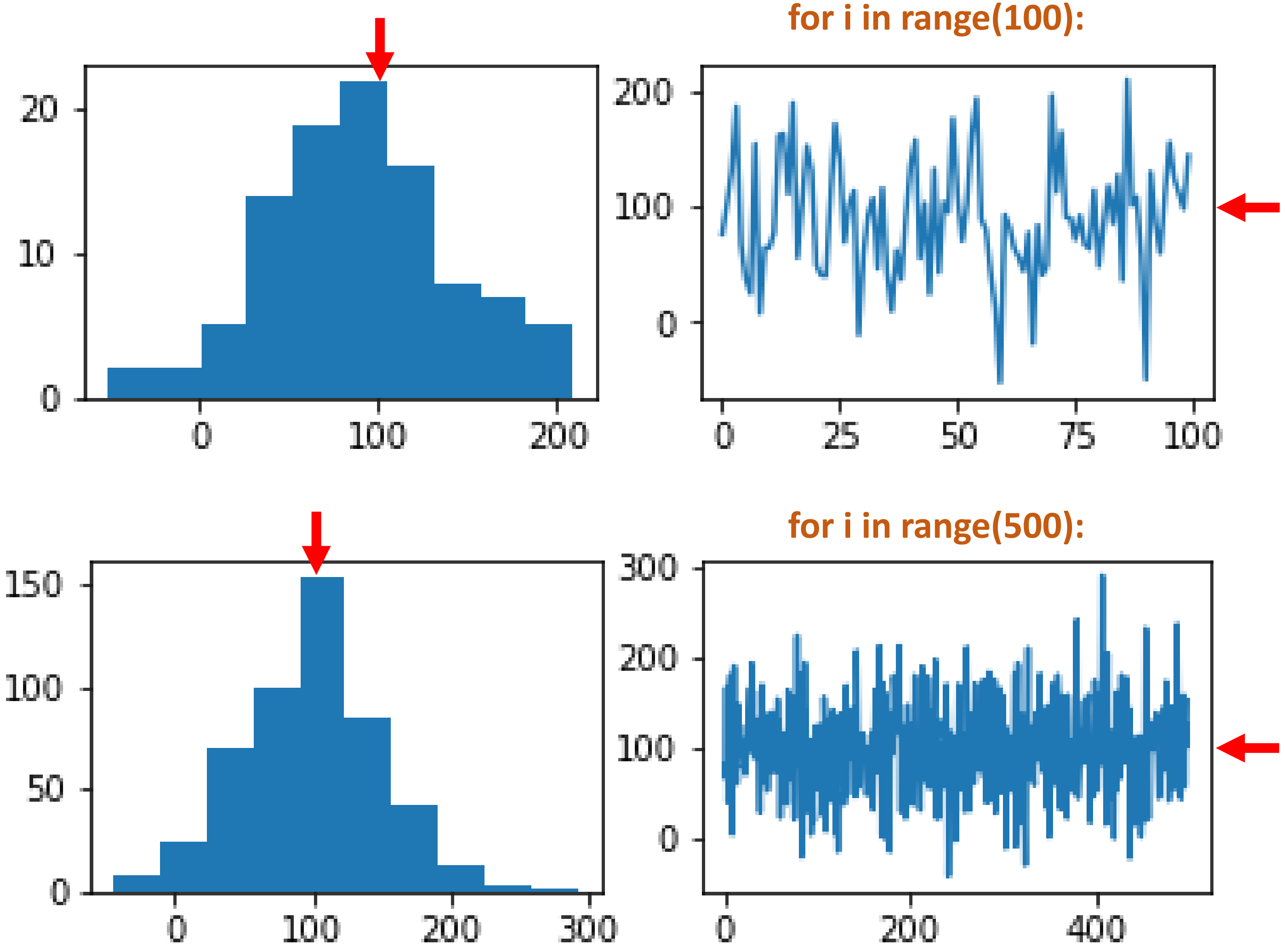
Loop

```
for i in range(500):
    temp = random.gauss(mu, sigma)
    nums.append(temp)
```

```
# plotting a graph
plt.subplot(2, 2, 1)
plt.hist(nums)

plt.subplot(2, 2, 2)
plt.plot(nums)
```

for i in range(100):

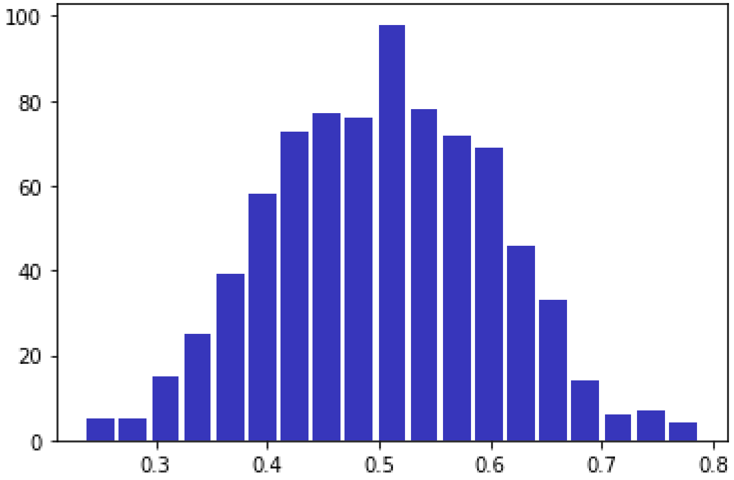
for i in range(500):



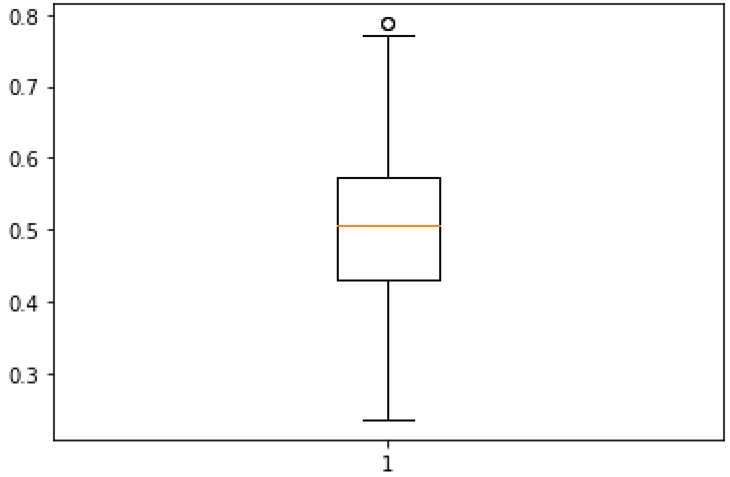
https://www.geeksforgeeks.org/random-gauss-function-in-python/

Numpy Random Normal Distribution

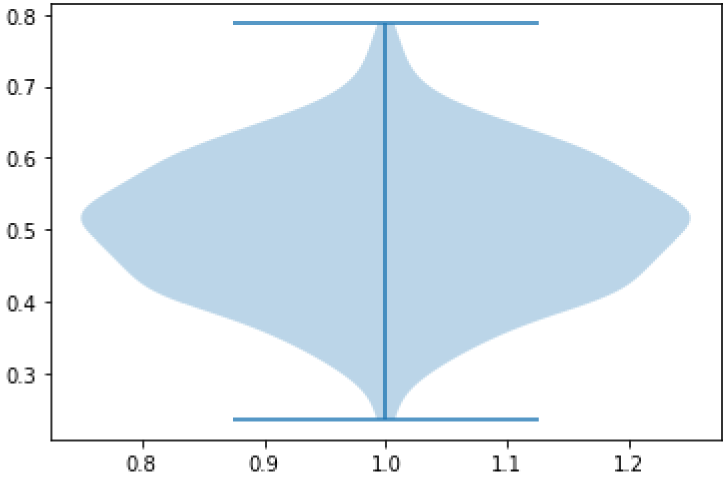
```
import pymc3 as pm
import numpy as np
import matplotlib.pyplot as plt

obs_y = np.random.normal(0.5, 0.1, 800)
n, bins, patches = plt.hist(x=obs_y, bins='auto', color='#0504aa',
                            alpha=0.8, rwidth=0.85)
```




```
n, bins, patches = plt.boxplot(x=obs_y)
```

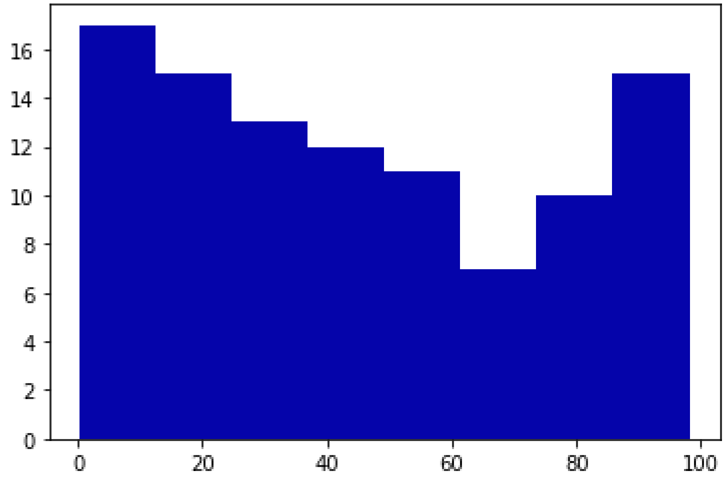



```
n, bins, patches = plt.violinplot(obs_y)
```

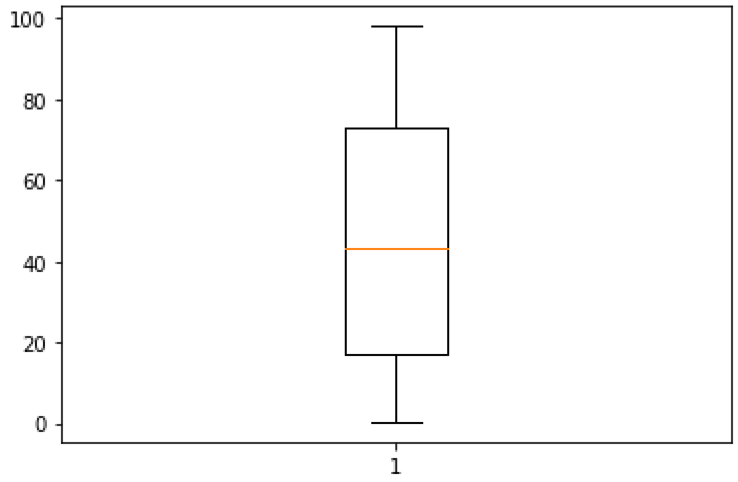


Numpy Random Uniform Distribution

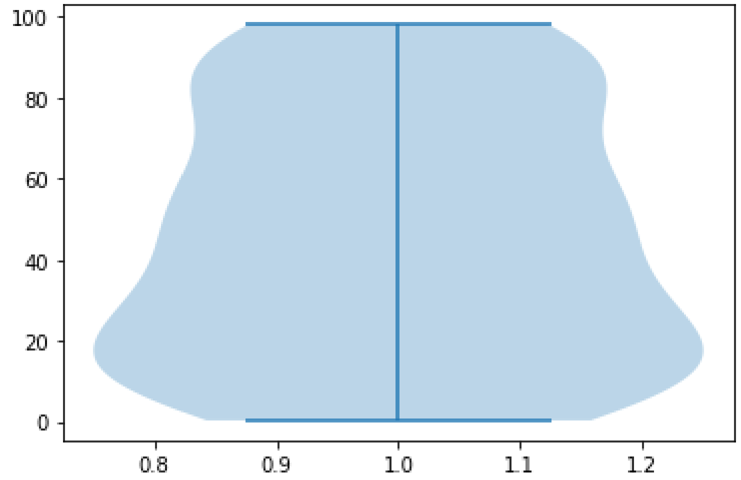
```
import pymc3 as pm
import numpy as np
import matplotlib.pyplot as plt

obs_y = np.random.uniform(0, 100, 100)
n, bins, patches = plt.hist(x=obs_y, bins='auto', color='#0504aa')
```




```
n, bins, patches = plt.boxplot(x=obs_y)
```




```
n, bins, patches = plt.violinplot(obs_y)
```



Monte Carlo Method

Monte Carlo methods, or Monte Carlo experiments, are a broad class of computational algorithms that rely on **repeated random sampling to obtain numerical results**. The underlying concept is to use randomness to solve problems that might be deterministic in principle. They are often used in physical and mathematical problems and are most useful **when it is difficult or impossible to use other approaches.** Monte Carlo methods are mainly used in three problem classes:[1] optimization, numerical integration, and generating draws from a probability distribution.

Despite its conceptual and algorithmic simplicity, **the computational cost associated with a Monte Carlo simulation can be staggeringly high**. In general the method **requires many samples to get a good approximation**, which may incurs an arbitrarily large total runtime if the processing time of a single sample is high.[11]

Although this is a severe limitation in very complex problems, the embarrassingly parallel nature of the algorithm allows this large cost to be reduced (perhaps to a feasible level) through **parallel computing strategies** in local processors, clusters, cloud computing, GPU, FPGA etc.[12][13][14][15]

GPU machines?

Monte Carlo methods were central to the simulations required for the **Manhattan Project**, though severely limited by the computational tools at the time. In the 1950s they were used at **Los Alamos** for early work relating to the development of the hydrogen bomb, and became popularized in the fields of physics, physical chemistry, and operations research.

https://en.wikipedia.org/wiki/Monte_Carlo_method

We know π but can you find this value through calculation?

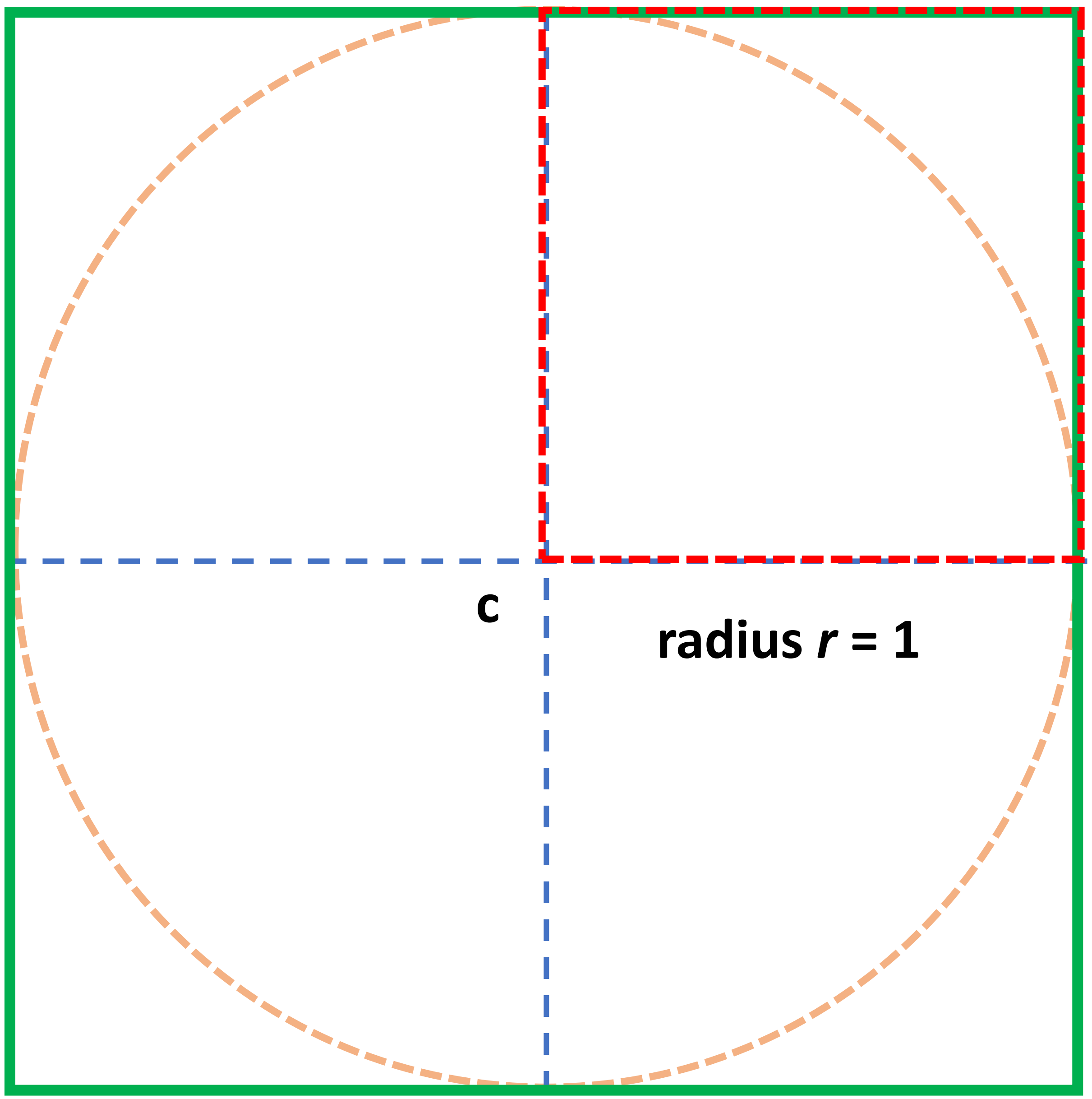
$$\frac{\text{Area of the circle}}{\text{Area of the square}} = \text{ratio}$$

$$= \frac{\pi r^2}{4 \times r \times r}$$

$$\frac{\text{Points inside circle}}{\text{Points inside square}} = \text{ratio}$$

$$\frac{\text{Points inside quadrant}}{\text{Points inside quarter N}} = \text{ratio}$$

$$= \frac{\pi/4 * N}{N}$$



Monte Carlo Method

Example 1. π calculation

Monte Carlo methods vary, but tend to follow a particular pattern:

- Define a domain of possible inputs.
- Generate inputs randomly from a probability distribution over the domain.
- Perform a deterministic computation on the inputs.
- Aggregate the results.

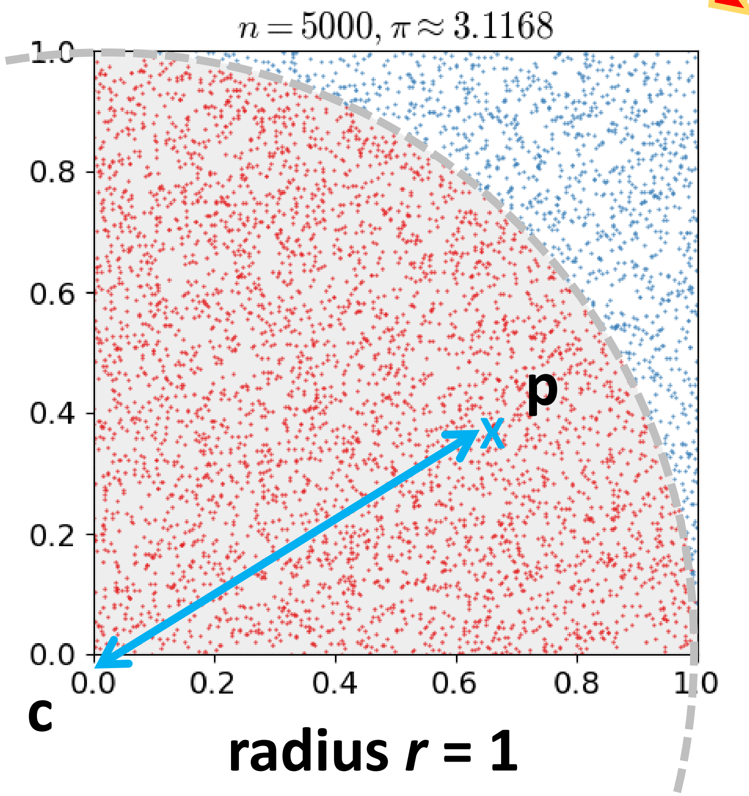
Demonstration: Approximate the value of π.

Consider a quadrant (circular sector) inscribed in a unit square. Given that the ratio of their areas is π/4, the value of π can be approximated using a Monte Carlo method.

Draw a square, then inscribe a quadrant within it.
Uniformly scatter points over the square. → 1,000,000
Count the points inside the quadrant
→ for p and the center c, d(p, c) < 1

$$\frac{\text{Points inside-quadrant}}{\text{the total-sample-count N}} = \frac{\pi/4 * N}{N}$$

Multiply the result by 4 to estimate π.



Area of circle with its radius *r* : $\pi * r^2$

Quadrant area = $\pi * r^2 / 4 = \pi / 4$

Square area = 1 * 1

https://en.wikipedia.org/wiki/Monte_Carlo_method

Examine how the π estimate changes over size of N

Monte Carlo methods vary, but tend to follow a particular pattern:

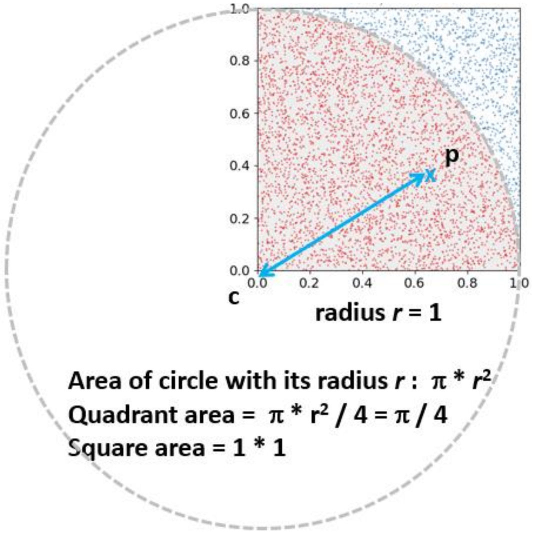
1. Define a domain of possible inputs
2. Generate inputs randomly from a probability distribution over the domain
3. Perform a deterministic computation on the inputs
4. Aggregate the results

Monte Carlo method applied to approximating the value of π.

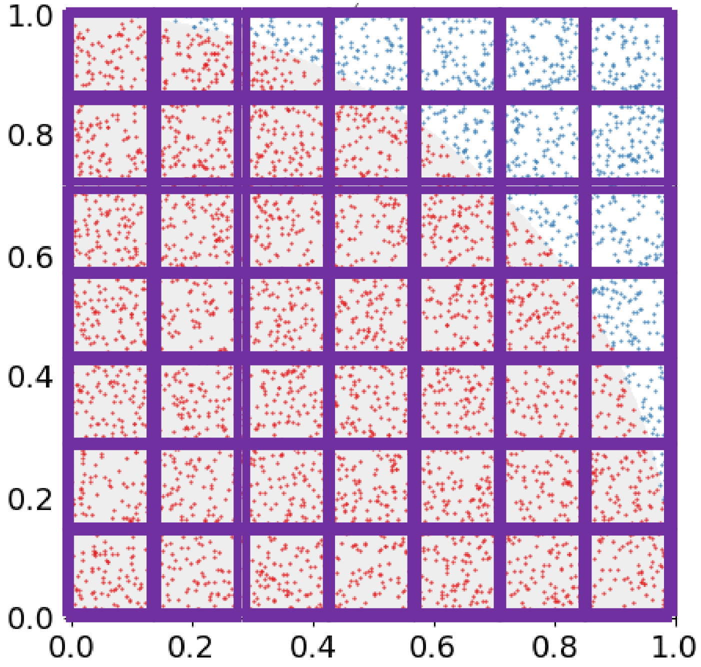
1. Draw a square, then inscribe a quadrant within it
2. Uniformly scatter a given number of points over the square
3. Count the number of points inside the quadrant, i.e. having a distance from the origin of less than 1
4. The ratio of the inside-count and the total-sample-count is an estimate of the ratio of the two areas, π/4. Multiply the result by 4 to estimate π.

There are two important considerations:
If the points are not uniformly distributed, then the approximation will be poor.
On average, the approximation improves as more points are placed.

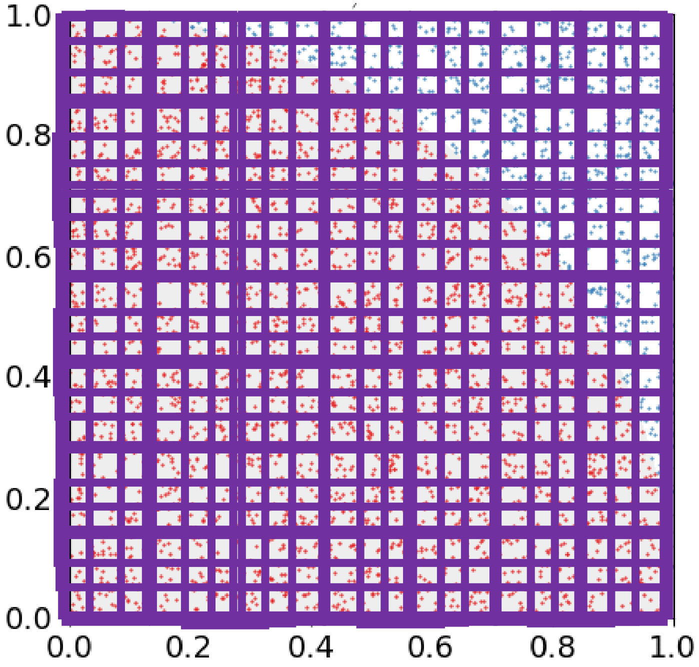
Uses of Monte Carlo methods require large amounts of random numbers, and it was their use that spurred the development of **pseudorandom number generators**, which were far quicker to use than the tables of random numbers that had been previously used for statistical sampling.



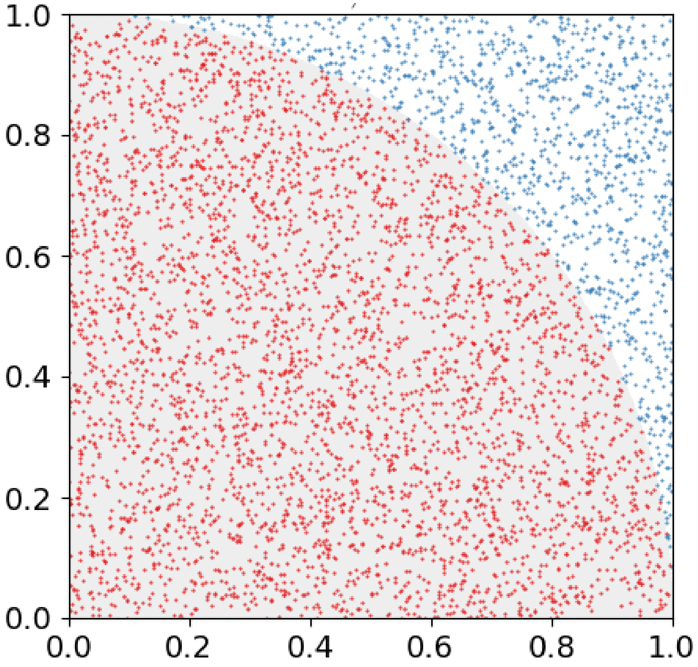
Examine how the π estimate changes over size of N



N = 7 x 7 = 49



N = 21 x 21 = 441



N = 5000

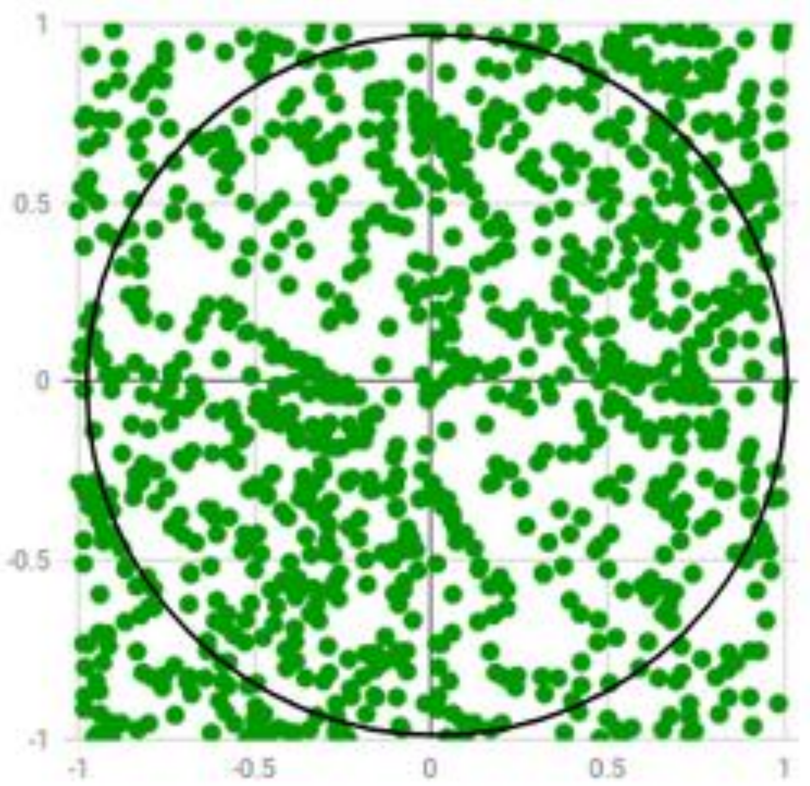
How accurate would be your π estimates?

What if you plot the π value as a function of N?

What types of visualizations can you produce to show the MC dynamics?
(e.g., Given N, show the distribution of π values with 100 repeats.)

Monte Carlo π calculation

```
import random

INTERVAL = 1000
circle_points= 0
square_points= 0

# Total Random numbers generated= possible x
# values * possible y values
for i in range(INTERVAL**2):

    # Randomly generated x and y values from a
    # uniform distribution
    # Rannge of x and y values is -1 to 1
    rand_x = random.uniform(-1, 1)
    rand_y = random.uniform(-1, 1)

    # Distance between (x, y) from the origin
    origin_dist= rand_x**2 + rand_y**2

    # Checking if (x, y) lies inside the circle
    if origin_dist<= 1:
        circle_points+= 1

    square_points+= 1
 # Estimating value of pi,
    # pi= 4*(no. of points generated inside the
    # circle)/ (no. of points generated inside the square)
    pi = 4* circle_points/ square_points

##    print(rand_x, rand_y, circle_points, square_points, "-", pi)
##    print("\n")

print("Final Estimation of Pi=", pi)
```



Random points are generated only few of which lie outside the imaginary circle

Final Estimation of Pi= 3.143916

https://en.wikipedia.org/wiki/Monte_Carlo_method

https://www.geeksforgeeks.org/estimating-value-pi-using-monte-carlo/

Monte Carlo integration in Python

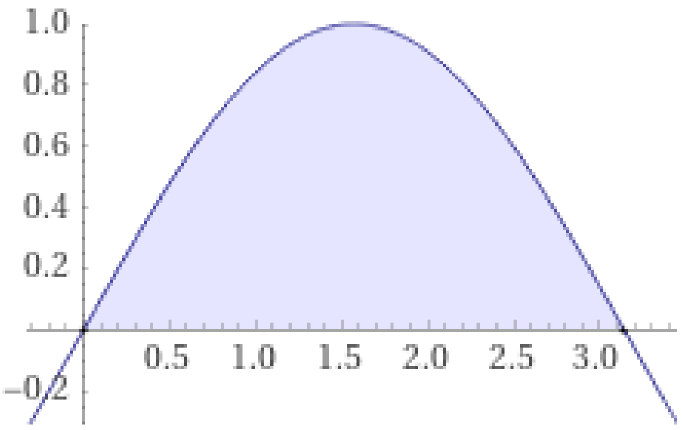
```
# importing the modules
from scipy import random
import numpy as np

# limits of integration
a = 0
b = np.pi # gets the value of pi
N = 1000

# array of zeros of length N
ar = np.zeros(N)

# iterating over each value of area and filling
# it with a random value between the limits a and b
for i in range (len(ar)):
    ar[i] = random.uniform(a,b)

# variable to store sum of the functions of different values of x
integral = 0.0

# function to calculate the sin of a particular value of x
def f(x):
    return np.sin(x)

# iterates and sums up values of different functions of x
for i in ar:
    integral += f(i)

# we get the answer by the formula derived adobe
ans = (b-a)/float(N)*integral

# prints the solution
print ("The value calculated by monte carlo integration is {}.".format(ans))
```

$$\int_0^{\pi} sinx\, dx = 2.0$$

Definite integral

$$\int_0^{\pi} \sin(x)\, dx = 2$$

Visual representation of the integral



→ The value calculated by monte carlo integration is 1.9584041688188545.

Monte Carlo integration in Python

```
# importing the modules
from scipy import random
import numpy as np
import matplotlib.pyplot as plt

# limits of integration
a = 0
b = np.pi # gets the value of pi
N = 1000

# function to calculate the sin of a particular value of x
def f(x):
    return np.sin(x)

# list to store all the values for plotting
plt_vals = []

# we iterate through all the values to generate multiple results and show whose intensity is the most.
for i in range(N):

    #array of zeros of length N
    ar = np.zeros(N)

    # iterating over each Value of ar and filling it with a random value between the limits a and b
    for i in range (len(ar)):
        ar[i] = random.uniform(a,b)

    # variable to store sum of the functions of different values of x
    integral = 0.0

    # iterates and sums up values of different functions of x
    for i in ar:
        integral += f(i)

    # we get the answer by the formula derived abobe
    ans = (b-a)/float(N)*integral

    # appends the solution to a list for plotting the graph
    plt_vals.append(ans)

# details of the plot to be generated sets the title of the plot
plt.title("Distributions of areas calculated")

# 3 parameters (array on which histogram needs
plt.hist (plt_vals, bins=30, ec="black")

# to be made, bins, separators colour between the beams; sets the label of the x-axis of the plot
plt.xlabel("Areas")
plt.show() # shows the plot
```

Loop

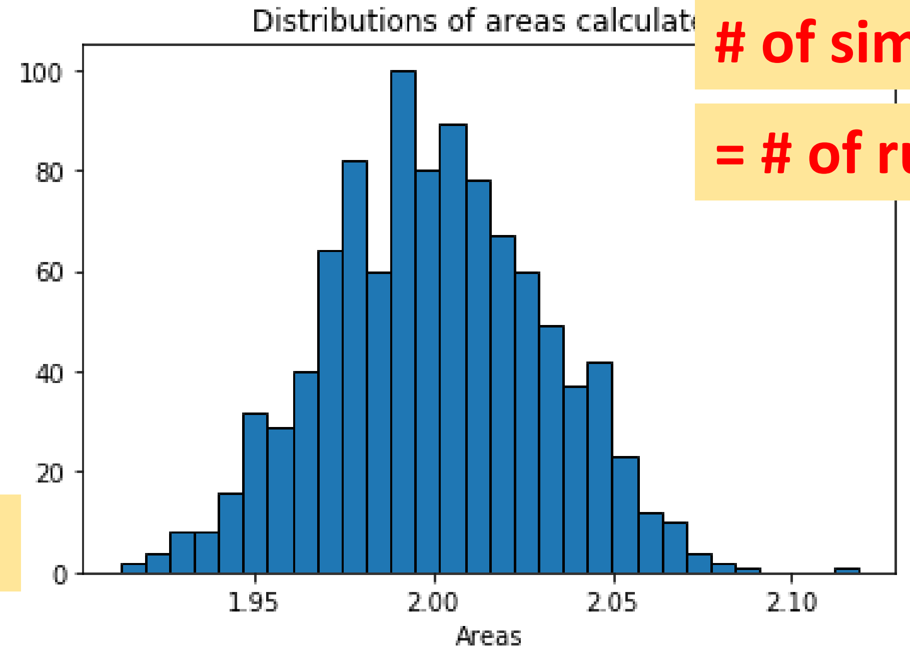
$$\int_0^{\pi} sinx\, dx = 2.0$$

# of samples

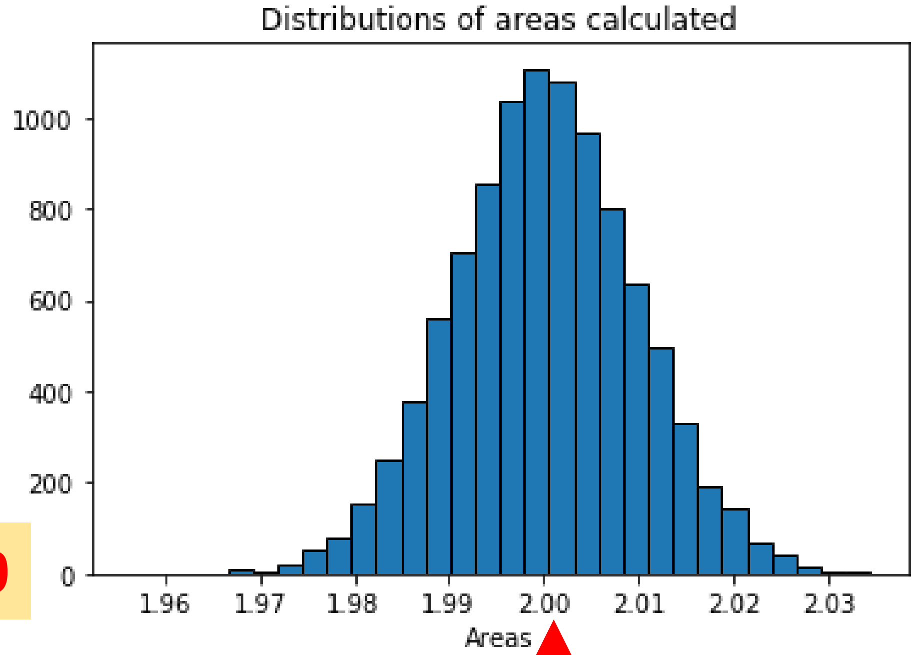
# of simulations

= # of runs

N=1,000



N=10,000

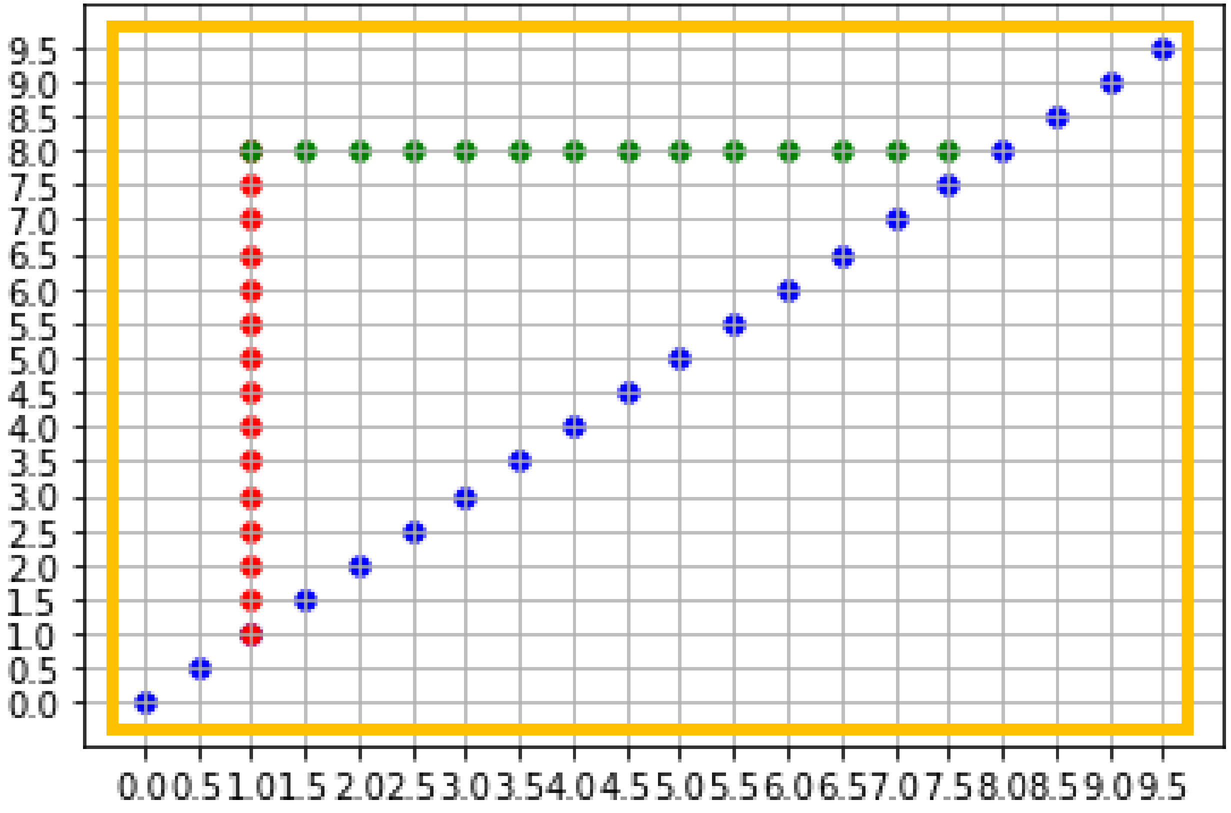


HW 6

Can you find the approximate area ratio (AR) surrounded by blue, red and green dots using Monte Carlo sampling method, i..e, AR = 120/400 = 0.3 where 120 is inclusive of the surrounding-colored dots?

```
# Sample code placing dots on grid
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import image as image
from matplotlib.patches import Rectangle

# Place diagonal dots: y = x
x = np.arange(0, 10, 0.5)
y = np.arange(0, 10, 0.5)

# Place 1st rise dots: x=1.0
xt = np.array([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0])
yt = np.array([1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0])

# Place 2nd hozizontal dots: y = 8.0
xf = np.array([1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5])
yf = np.array([8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0])

fig = plt.figure()
ax = fig.gca()
ax.set_xticks(np.arange(0, 10, 0.5))
ax.set_yticks(np.arange(0, 10, 0.5))
plt.scatter(x, y, color = "b", marker = "o", s = 30)
plt.scatter(xt, yt, color = "r", marker = "o", s = 30)
plt.scatter(xf, yf, color = "g", marker = "o", s = 30)
plt.grid()
```

Repeat the sampling 500 time and find μ.
How close is the estimated μ to the true area?
What if the grid size becomes finer (e.g., 0.5 → 0.1) or real numbers?



Create a grid that can place 400 (=20x20) dots (i.e., inside the orange bounding box).