EXPLORING TWO DIFFERENT METHODS OF OBTAINING RANDOM POINTS WITH UNIFORM DENSITY IN A CIRCLE AND COMPARING THEIR COMPUTATIONAL EFFICIENCY.

(draft)

Contents

Page. No

Background 2
MEASUREMENT OF COMPUTATIONAL EFFICIENCY 2
First Method: Rejection sampling 3
LIMITATION OF THE METHOD 5
The second method: Sampling on the polar coordinate system 6
POINT DISTRIBUTION IN DIFFERENT RADII 8
THE EXPECTED DISTRIBUTION 11
INVERSE TRANSFORM SAMPLING 14
Comparing the computational efficiency of the two methods 17
RELATION TO π APPROXIMATION WITH MONTE CARLO SAMPLING 18
References: 21
Appendix 1: Fewer instructions count $\nRightarrow$ faster program 22
Output: 22
Appendix 2: Python implementation of rejection sampling method 23
Appendix 3: Python implementation of sampling on polar coordinates 24
Appendix 4: Visualizing the relationship between points density and distance from the center given constant number of points per radius 25
Appendix 5: Comparing the executing speed 25
THE RESULTS OF EXECUTING TEST COMPARISON 26

Background

In my free time, programming is the primary way of expressing my creativity: programming electronic systems, building simple software, and most importantly, programming allows me to experiment and visualize different mathematical concepts.

Several months ago, I was working on developing a simple multiplayer online game of chance[1]. The game would prompt the players to put their fingers anywhere within the given circle, and the game will then generate a random point within the circle. The player whose chosen point has the shortest distance to the point selected by the game is the winner!

A crucial part of the game was to develop an algorithm that would generate uniform random points, such as all positions inside the circle having the same probability of being selected. While researching this, I came across different approaches I could use. However, I needed to find out which method was the most efficient.

MEASUREMENT OF COMPUTATIONAL EFFICIENCY

In software development, the computational efficiency of a program can be measured by using various factors such as CPU utilization, memory utilization, error rates, average load time, execution speed, and instruction count, to mention a few. But for the scope of this essay, I will only measure the efficiency in terms of execution speed and instructions count. It should be acknowledged that this analysis might not be very accurate as the efficiency of a program relies on numerous factors other than just execution speed. While fewer instruction

[1] A game of chance is a game whose outcome depends on some random influence.

count generally implies a faster program, this is not always the case – this is further explained in *appendix 1*.

In this essay, I will explore the mathematics behind two methods/algorithms of generating random points with uniform density in a circle and computational efficiency in terms of execution speed and instruction count.

First Method: Rejection sampling

To implement this method, we will first inscribe a circle inside a square, as shown in *fig.1*. Given the circle's radius is r, the side of the square is equal to the circle's diameter- **2r.**

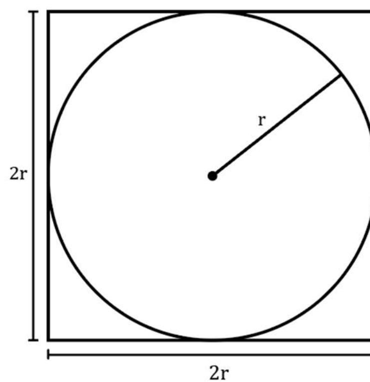

Fig.1

From here, we could assume the Horizontal and vertical sides of the square as the *x* and *y*-axis of a cartesian plane, respectively. We will then randomly select a point along the horizontal axis and on the vertical axis. This method will have two cases where the chosen point could either fall inside the circle-*fig.2*, or outside -*fig.3.*

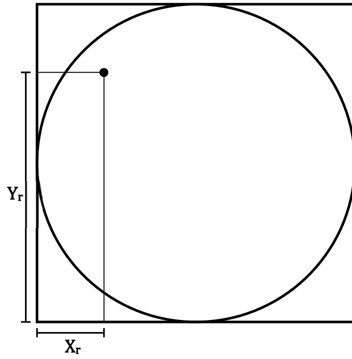

Fig 2: The selected point falls within the circle

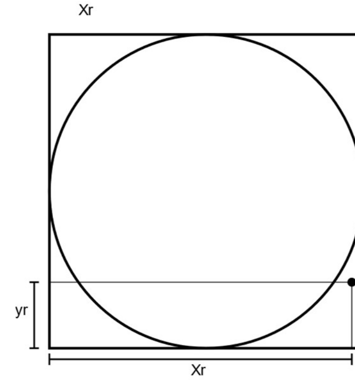

Fig 3: The selected point does not fall within the circle

To account for cases where obtained coordinates do not form a point within the circle, we should restrict our selection such that the distance from the point to the center of the circle is less than the radius.

From the equation of the circle;

$$x^2 + y^2 = r^2$$

This equation gives us all the points that are r units distant from a common point -the circle's center. So, for a point **(x, y)** to fall within the circle of radius **r**, the distance between the point and the center should be less than **r.**

$$\Rightarrow x^2 + y^2 < r^2$$

So, in our rejection sampling, we will accept all the random points **(x_r, y_r)** such that $\boldsymbol{x_r}^{\mathbf{2}} + \boldsymbol{y_r}^2 < \boldsymbol{r_{max}}^{\mathbf{2}}$ - where $\boldsymbol{r_{max}}$ is the radius of the circle in this case. Otherwise, the point will be rejected.

This mathematical information can be transcribed for python implementation, as shown and explained in the code in *appendix 2*. After executing the program *7000* times, the result is as shown in *fig.4.*

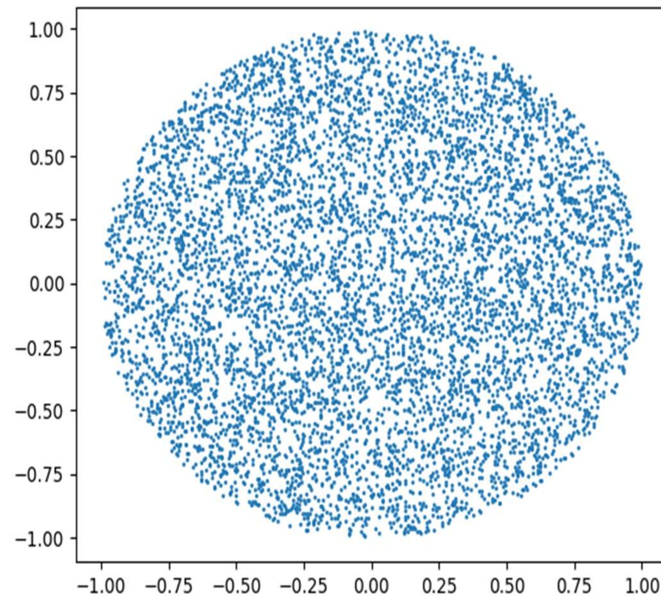

Fig 4: 7000 points uniformly distributed in a unit circle

LIMITATIONS OF THE METHOD

Rejection sampling is based on the fact that in a given event, failure is probable. The case is the same in our method, which is the main limitation. For instance, if we want to obtain $\boldsymbol{n}$ uniform random points, we will have to run the program at least **(n + k)** times, where **k** is the number of times the point obtained falls outside the circle – failure. This implies that the probability of the event where a point will fall inside the circle is given by;

$$\boldsymbol{P}(\boldsymbol{X}) = \frac{\boldsymbol{n}}{\boldsymbol{n}+\boldsymbol{k}}$$

This is similar to taking the circle's area and divide by the area of the square from *fig.1*;

$$\textit{The area of the square with side } s = \boldsymbol{s^2} = (\boldsymbol{2r})^2$$
$$= \boldsymbol{4r^2}$$

$$\textit{The are of the cirlce with radius } \mathrm{r} \textit{ given by} = \boldsymbol{\pi r^2 2}$$

$$\Rightarrow \textit{The probability of selecting a point withing a circle } = \frac{\boldsymbol{\pi r^2}}{\boldsymbol{4r^2}}$$

$$= \frac{\boldsymbol{\pi}}{\boldsymbol{4}}$$

Since this event represents a binomial distribution;

$$\Rightarrow \textit{ the Probabilty of the randomly obtained point being outside the circle} = \left(\boldsymbol{1 - \frac{\pi}{4}}\right)$$

This implies that the probability of not obtaining a point inside the circle $\boldsymbol{n}$ times in a row is given by;

$$\left(\boldsymbol{1 - \frac{\pi}{4}}\right)^{\boldsymbol{n}}$$

For instance, if the program was to be executed four times, the probability of not obtaining a single point within the circle is about 0.00212. In other words, there is a 99.78% chance that at least one point will fall within the circle.

Therefore, this limitation does not broadly undermine the computational significance of this method because the probability of failure becomes very small as the number of trials becomes very big. Another way to imagine this will be by limits;

$$\Rightarrow \lim_{\boldsymbol{n \to \infty}} \left(\boldsymbol{1 - \frac{\pi}{4}}\right)^{\boldsymbol{n}} = \boldsymbol{0}$$

The second method: Sampling on the polar coordinate system

In the previous rejection sampling method, we used the cartesian plane to specify the position of a point relative to the circle by a pair of numerical coordinates. In a polar coordinate system, any point can be defined by a distance from a reference point – the pole, analogous to the origin in the cartesian system, and an angle from a reference direction. So, the position of a point can be denoted as $(\boldsymbol{r}, \boldsymbol{\theta})$

Where;

$\boldsymbol{r}$ = *distance from the pole-the origin.*

$\boldsymbol{\theta} = The\ angle\ from\ the\ reference\ direction.$

As we have seen previously, when doing the sampling with the cartesian system, we have to take into account the possibility of selecting points outside the circle hence rejection sampling. Using polar coordinates removes the necessity of doing rejection sampling because, for a circle with a radius of 1, we can select a random number in the interval **(0, 1)** = $\{x \mid 0 < x < 1\}$ and a random angle in the range **$[0, 2\pi]$ =** $\{x \mid 0 \leq x \leq 2\pi\}$and, Of course, given that the sample space is only defined within an interval, the probability of an outcome outside the interval is zero.

Obtaining uniform random points within a circle by using this method takes the following steps.

$1. By\ using\ a\ python\ random\ function\ select\ a\ random\ number\ between\ \mathbf{0}\ and\ \mathbf{2\pi}$

$2. Select\ a\ random\ number\ betwee\ \mathbf{0}\ and\ \boldsymbol{r}, where\ \boldsymbol{r}\ is\ the\ radius\ of\ the\ cirlce$

$3. Convert\ the\ randomly\ selected\ polar\ coordinates(\boldsymbol{r_r}, \boldsymbol{\theta_r})\ into\ the\ cartesian\ form(\boldsymbol{x}, \boldsymbol{y});$

$$\boldsymbol{x = r_r * cos(\theta_r)} \quad \boldsymbol{and} \quad \boldsymbol{y = r_r * sin(\theta_r)}$$

These mathematical instructions can be adapted as pseudo instructions for a python implementation, as shown in the code in *appendix 3.* After running the program *7000* times, we expect to get uniform random points within the circle, but the results are pretty different.

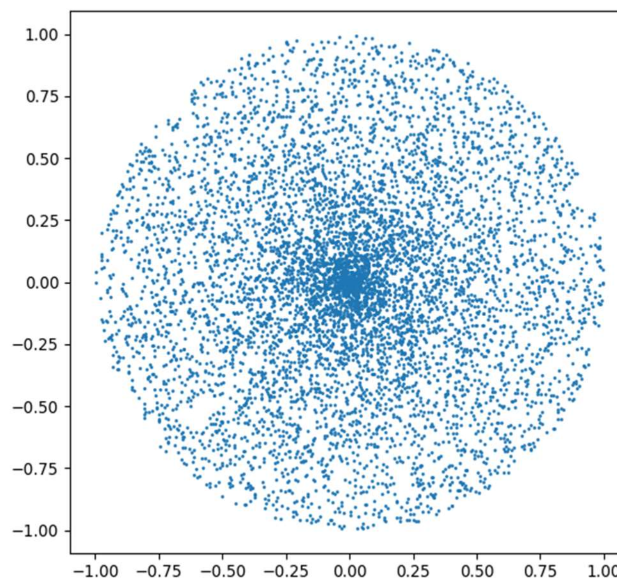

Fig 5: Visualization of random points distribution obtained by the polar-coordinates method

At first, it is surprising that the density does not appear uniform. The points are much more densely packed towards the center and less densely packed as you move outwards.

POINT DISTRIBUTION IN DIFFERENT RADII

To understand the situation, I will create a histogram to show the frequency of the points in a given radius. A histogram gives us a good approximate representation of the corresponding

probability distribution function (***PDF***). The only disparity is that a histogram involves discrete data (individual classes or bins). On the other hand, a *PDF* involves continuous data, hence a smooth curve in *PDF* as opposed to a histogram.

The probability of selecting any radius is uniform, and so is the probability of selecting any angle. This implies that on average all the radii will have the same number of points.

To generate the histogram, I will calculate the distances to the center of every point generated from *fig.5*, but because I am using polar coordinates the distance of a given point $(\boldsymbol{r_r}, \boldsymbol{\theta_r})$ is simply the magnitude of $\boldsymbol{r_r}$**.** All these distances are then stored in a python list and then a histogram is plotted, this is shown and explained in the code snippet in *appendix 3*. The resulting histogram from the 7000 points in *fig.5*, is shown below.

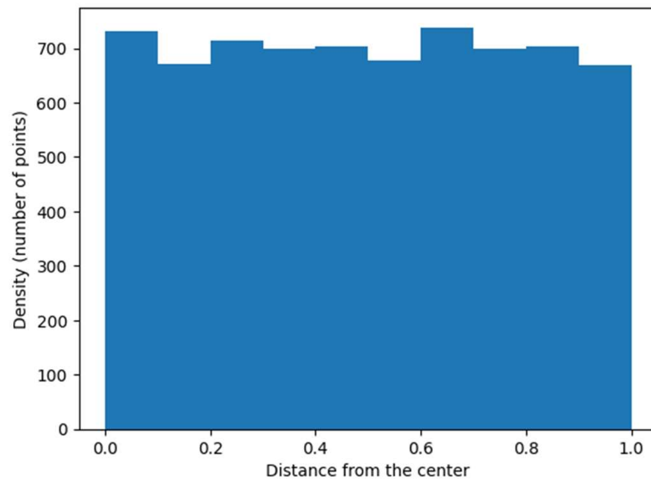

Fig 6: Histogram showing frequency distribution of distances of the 7000 points from the center after sampling on polar coordinates

The histogram in *fig.6* approximates the probability distribution of points in *fig.5*. To get an even better approximation of the PDF of the point distribution, we can generate a larger sample of the points. For instance, if we generate 1 million points, the results are as follows.

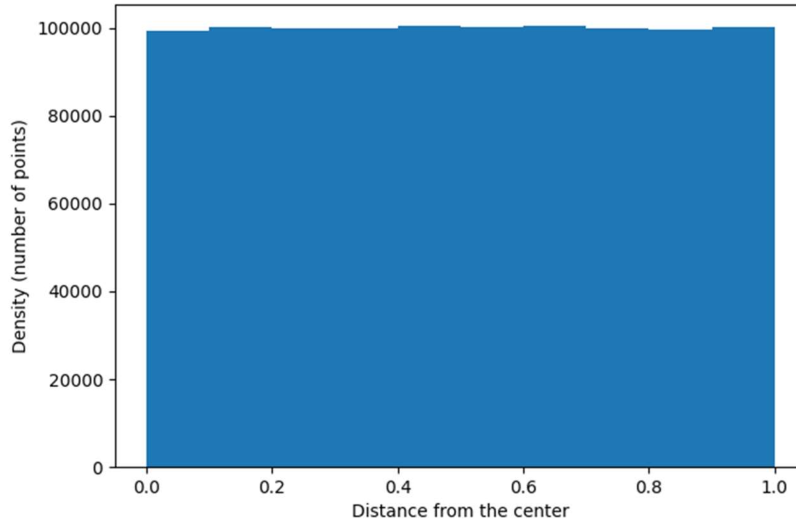

Fig 7: A histogram showing the distribution of 1 million points on different radii from 0 to 1 (figure produced by the candidate in Python Matplotlib)

The distribution looks more rectangular as the number of samples increases, distributions with this behavior are uniform distributions, so the PDF of this distribution is simply a straight line enclosing a rectangular area.

The convergence of the distribution to a uniform one can also be explained by the law of large numbers (LLN) where the average of the results of an experiment converges to the expected value as the number of trials becomes very large, that is;

$$\lim_{n\to\infty}\sum_{i=1}^{n}\frac{x_i}{n}=\bar{x}$$

Uniform distribution implies an equal probability of a randomly selected point within a radius between 0 and 1. So on average, all radii will have the same number of points. Point density in a radius can be interpreted as the number of points in that radius divided by the circumference;

$$points\ density = \frac{number\ of\ points\ in\ a\ given\ circumference}{circumference}$$

$$points\ density = \frac{\boldsymbol{n}}{\boldsymbol{2\pi r_r}}$$

From the equation above, we can deduce that, given a constant $\boldsymbol{n}$ number of points in every radius, points density is inverse proportional to the radius $\boldsymbol{r_r}$. This justifies that the points will be much more spaced or less densely packed for a bigger radii. An illustration below visualizes this in a case where 70 points are equally spaced around the circumference of 30 different rings-radii (code explanation is in *appendix 4*), it shows; Toward the center, points are more densely packed as supposed by the random points generated in *fig.5*.

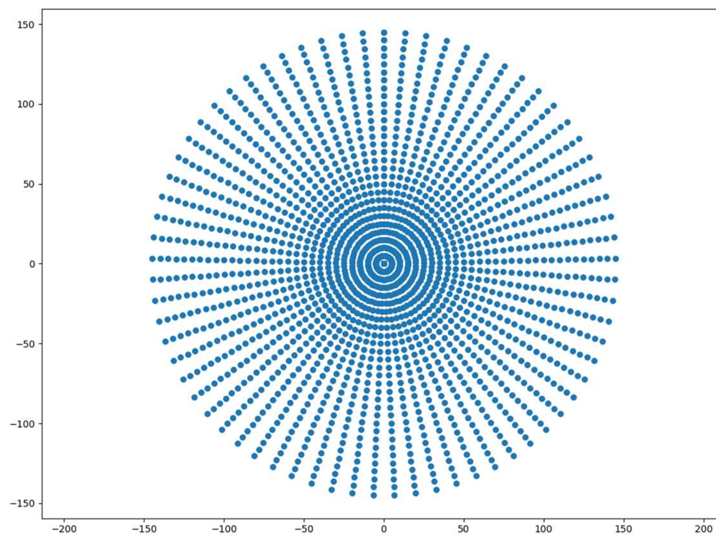

Fig 8: A Visualization of how points density is inverse proportional the distance from the center for constant number of points (figure generated in Matplotlib by the candidate)

It follows that the circumference grows linearly with the radius, so the number of points should also grow linearly with the radius to keep the density of the points uniform. It is a bit unintuitive, but in other words, to have uniform density, The point distribution must not be uniform.

THE EXPECTED DISTRIBUTION

As we have already attained the desired results from the previous rejection sampling method, we can generate a histogram to visualize the desired PDF. To do this, we can make a python implementation -as explained within the code in *appendix 2* - which calculates the distance from the center of all the points in *fig.4* and store it in a python list, the list can then be used to generate a histogram which shows the frequency distribution of points in all distances between 0 and 1. The result is shown below.

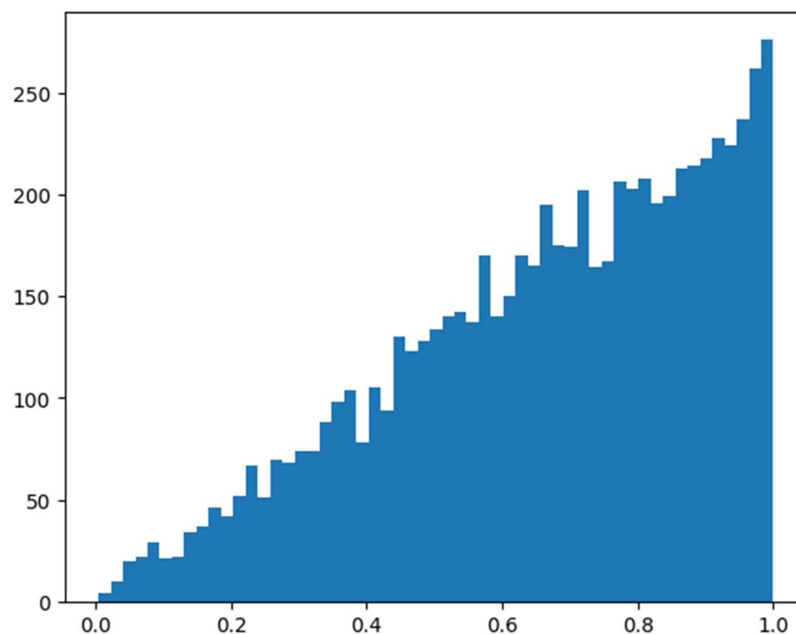

Fig 9: An illustration showing how density varies on different radii given a constant number of points

From the frequency distribution shown by the histogram, it is evident that the number of points per radius increases as the radius increases from 0 to 1, the outermost rings of points should contain more points than their corresponding innermost. So, our desired PDF should be a linear function. The PDF can therefore be expressed as an equation of a straight line;

$$\boldsymbol{f_X(r) = mr + C, \qquad 0 < X < 1}$$

Because our domain excludes radii of 0, the frequency of points whose distance is 0 units will be zero. In other words, the probability of getting a radius of 0 is zero, this implies that the line traced by the equation passes through the origin, so the y-intercept **(C)** is zero;

$$\Rightarrow \boldsymbol{f_X(r) = mr + 0}$$

$$= \boldsymbol{f_X(r) = mr}$$

Therefore, the graph of the desired PDF will look as follows;

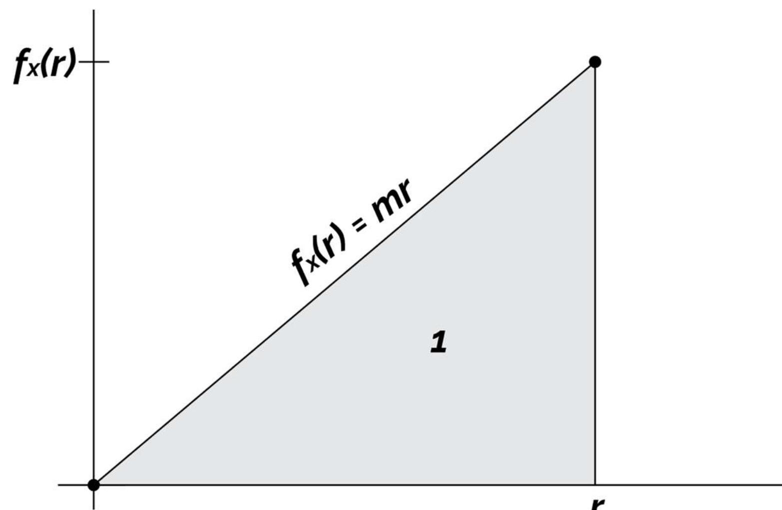

Fig 10: The graph shape of the expected pdf

But from our distribution function $\boldsymbol{f_X(r)}$, the probability ***P(X),*** of picking a radius between **0** and the maximum radius ***1*** is ***100%,*** that is;

$$\boldsymbol{P(0 < X < 1) \ = \ 1}$$

This probability is represented by the area under the graph, as shown in the above figure. This is the cumulative probability which can also be found by integrating the probability distribution function;

$$\Rightarrow \int_{0}^{r_{max}} \boldsymbol{f_X(r)\, dr = 1}$$

$$\Rightarrow \int_{0}^{r_{max}} mr\,dr = 1$$

The maximum possible radius r_{max} can be assumed to be simply the circle's radius – it is an approximation because r_{max} itself is excluded. In our case, the radius is r_{max} is 1.

$$\Rightarrow m\left[\frac{r^2}{2}\right]_0^1 = 1$$

$$\Rightarrow m = 2$$

$$\therefore f_X(r) = 2r\,,\ (0 < X < 1)$$

INVERSE TRANSFORM SAMPLING

Because we already know the required PDF, we can use the inverse transform sampling method, which can help us randomly generate our sample radius according to our PDF given the cumulative distribution function, but the CDF is simply the integral of the PDF

$$\Rightarrow The\ CDF\ F_X(r) = \int f_X(r)\,dr$$

$$= \int 2r\,dr$$

$$\therefore F_X(r) = r^2$$

A cumulative distribution function $F_X(r)$ gives us the probability of the random variable X being less or equal to its input r, that is;

$$F_X(r) = P(X \leq r)$$

While the random variable X does not have a uniform distribution, its corresponding probabilities do have a uniform distribution. To visualize this let us sketch the graph of $F_X(r)$;

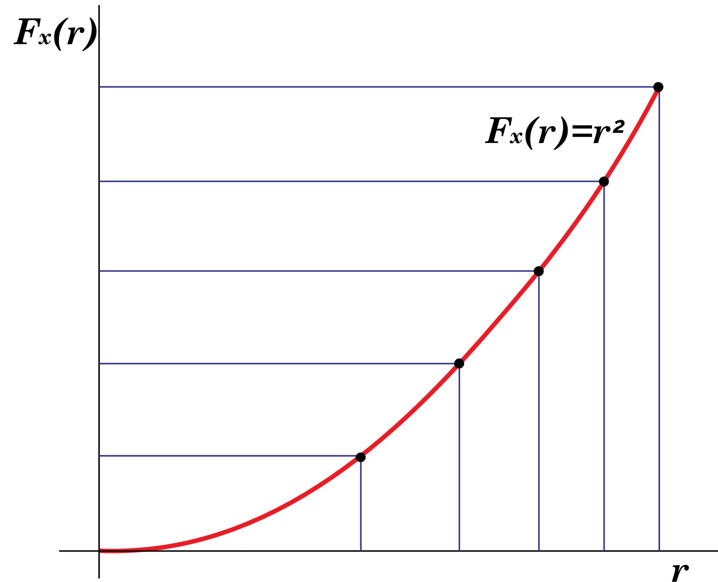

Fig 11: Showing ununiform distribution with uniform probabilities

As the graph of $F_X(r)$ shows, if we treat the probabilities (*Y-values*) as the input and take for example 5 uniformly distributed points, the results shown on the *x-axis* reflect our desired results where points with a greater value of r are more packed compared to the points on the lesser value of r, this is as opposed to the distribution frequency from figure xx. This implies that to get our desired distribution, we should take the inverse of our CDF-inverse transforming;

$$F_X(r) = r^2$$

$$\Rightarrow r = \left[F^{-1}{}_X(r)\right]^2$$

$$\therefore F^{-1}{}_X(r) = \sqrt{r}$$

This implies that after obtaining a random number between 0 and 1, we will take the square root of the number as the new random radius to obtain our desired distribution. The python implementation is as shown in the following code snippet (further explained in the code in *appendix 2*.)

```
 while(points_obtained<number_of_points): #conditions for the loop to terminate
        r = math.sqrt(select.random()*radius) # using the relationship obtained in inverse transforming to obtained uniform density
```

With the new change in how we sample the radius, we get our desired results as shown below.

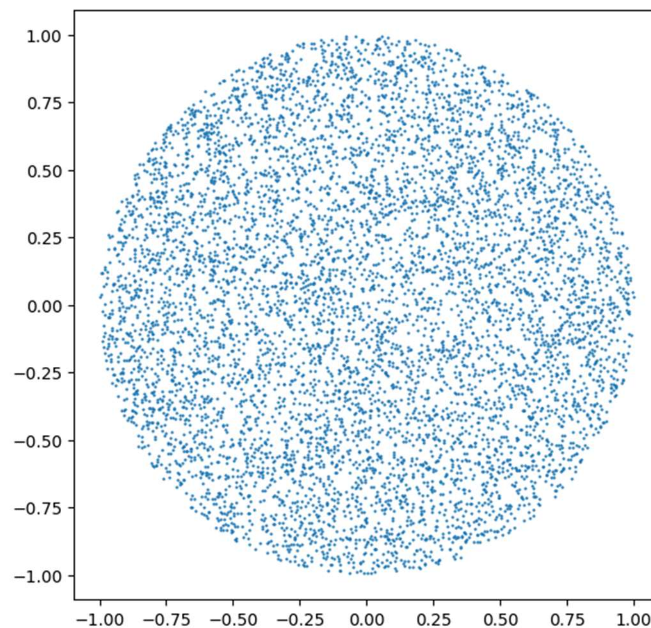

Fig 12: Points distribution with uniform density obtained by sampling on polar coordinates

To visualize the distribution, we can again use 1 million points, with their distances plotted on a histogram, as follows;

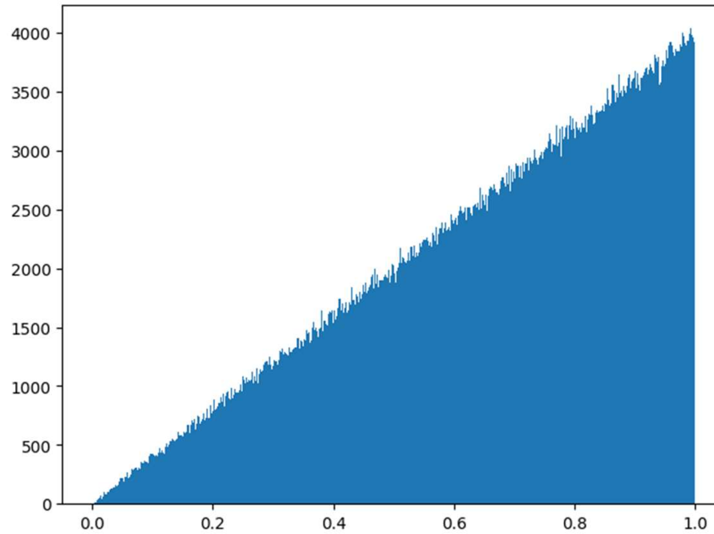

Fig 13: Histogram drawn with 1 thousand bins to show Frequency distribution of distances of 1 million points.

The results obtained by sampling 1 million times look almost the same as the PDF obtained previously - ***f(x) = 2x*** in *fig.10*. This histogram gives us an excellent approximation, so our results are correct.

Comparing the computational efficiency of the two methods

As previously mentioned, I will perform an execution speed test of the two methods and use it to assess the computational efficiency. However, as already mentioned, this method can not accurately assess computational efficiency.

8 million uniformly distributed points in a circle with a radius of 1 were generated by both methods. The python code implementation for this is explained in *appendix 5.* After running this experiment, the result is as shown below;

Method	Number of points	Execution time (s)
Rejection sampling	8000000	16.05
Polar coordinates method	8000000	15.61
Rejection sampling	1000	0.00021
Polar coordinates method	1000	0.00026

The polar second method seems to be performing averagely faster compared to the rejection sampling method, but with a few numbers of points, for instance 1000, the rejection sampling method performs faster than the polar coordinate method. The reason for rejection sampling to become slower as the number of samples increases could be; as the number of points to be generated increases, the number of times a point is outside (the points to be rejected) the circle increases, so the program becomes slower. This is also proof by counterexample to the previously stated mentioned statement (analyzed in *appendix 1)*, that fewer instructions do not necessarily imply a faster program. As we have seen, the rejection sampling method sometimes runs faster despite having to repeat the instructions several times.

It is safe to say that the polar coordinate method is generally faster. However, the application of one method over the other depends on how many points we intend to generate. The rejection sampling method has proved to be faster when generating a fewer number of uniform points in a circle but loses its advantage as the required number of points increases.

RELATION TO π APPROXIMATION WITH MONTE CARLO SAMPLING

While working on writing programs for this exploration, I remembered a method we studied in class, which estimates the value of π by the Monte Carlo method. It follows that given a circle inscribed in a square as shown in *fig.1* if particles were randomly scattered on the surface of that figure, the probability that a particle falls inside a circle is $\frac{\pi}{4}$, proof for this has been explained in the previous section.

In the case of our program, where we use rejection sampling, we have seen that if we want to generate exactly $\boldsymbol{n}$ points inside the circle, we will have to run the experiment at least **n +** $\boldsymbol{k}$ times, where $\boldsymbol{k}$ is the number of times the program fails;

$$\Rightarrow \text{probability of success } \frac{\boldsymbol{\pi}}{\mathbf{4}} = \frac{\boldsymbol{n}}{\boldsymbol{n}+\boldsymbol{k}} = \frac{number\ of\ successfull\ \ trials}{Total\ number\ of\ trials}$$

So, we can simply record the number of trials taken to generate $\boldsymbol{n}$ points, and the quotient multiplied by **4** will give us an approximation of $\boldsymbol{\pi}$**.** The following code snippet shows the implementation in python, where 7000 points are generated.

```
quotient = points_obtained/trials
print(f'Pi = {pi}')
```

Output

```
C:\Users\PC\Desktop\Math\randompoints\venv\Scripts\python.exe C:/Users/PC/Desktop/Math/randompoints/distribution.py
The quotient = 3.1410857948321285
```

$$\Rightarrow \frac{\boldsymbol{\pi}}{\mathbf{4}} = \mathbf{0.7852714487080321}$$

$$\therefore \boldsymbol{\pi} = \mathbf{3.1410857948321285} \approx \mathbf{3.141}$$

This gives us a good approximation of $\boldsymbol{\pi}$ to the accuracy of 3 decimal places!

Conclusions

In this analysis, I was able to explore the mathematical aspects of 2 different methods of generating random points with uniform density in a circle. I was able to empirically deduce which method is relatively superior in terms of executing speed, and depending on the required number of points to be generated, for instance for the game I am developing rejection sampling method will be suitable as there will be only a few players at a given time – realistically less than 100. This exploration allowed me to appreciate the interdependence

of programming and mathematics. When I expected the least, I found out my program could be used to estimate the value of pi to an accuracy of 3 decimal places.

Limited by the computing power of my laptop, I could not properly take advantage of the law of large numbers, mainly because I could not generate more than 10 million samples. In the future when I have access to more powerful devices, I will be able to obtain results of higher accuracy. But for this scope of exploration, this limitation does not undermine the mathematical significance of my results.

In general, this exploration was a great opportunity to apply my programming and data science knowledge from a mathematical perspective. The end of this exploration marks the beginning of the newly discovered math exploration possibilities.

References:

aioobe. (2018a, June 7). Answer to “Generate a random point within a circle (uniformly).” Stack Overflow. https://stackoverflow.com/a/50746409

aioobe. (2018b, June 8). Generate a random point within a circle (uniformly) [Forum post]. Stack Overflow. https://stackoverflow.com/q/5837572

alvas. (2017, May 23). How to measure time taken between lines of code in python? [Forum post]. Stack Overflow. https://stackoverflow.com/q/14452145

Marengo, J. E., Farnsworth, D. L., & Stefanic, L. (2017). A Geometric Derivation of the Irwin-Hall Distribution. International Journal of Mathematics and Mathematical Sciences, 2017, e3571419. https://doi.org/10.1155/2017/3571419

nubDotDev. (2021, August 21). The BEST Way to Find a Random Point in a Circle | #SoME1 #3b1b. https://www.youtube.com/watch?v=4y_nmpv-9lI

Appendix 1: Fewer instructions count ⇏ faster program

With modern pipelined[2] processors, the number of instruction counts cannot be used to estimate performance or speed of a program, to put this into context let us compare the time taken by a computer program to find the cosine of π and compare to the time taken to find the square of π n times. On average, time taken to find the cosine of π n times is almost twice the time taken to find the square of π $2n$ times, this is a simple example of why fewer number of instructions does not necessarily imply a faster program. Program implementation is as shown below.

```
import time #Importing time library enable time measurement to 10-6 of a second
import math #Math library allows implementation of advanced mathematical operation

def squaring_pi(n): # defining a function which find the square of pi n times (n as
the input)
    for i in range(n): #looping the execution n times
        3.14**2

def cosine_pi(n): # defining a function F(n) to find the cosine of pi n times (n as
the input)
    for i in range(n): #looping the execution n times
        math.cos(3.14)

start = time.perf_counter()  #recording the time the execution started
squaring_pi(10000000)       #Calling the squaring function
end = time.perf_counter()   #recording the time the execution ended
print(f"time to square pi 2 million times: {end-start}s") #Calculating and printing
squaring function execution time

start = time.perf_counter() #recording the time the execution started
cosine_pi(5000000)    #Calling the cosine function
end = time.perf_counter()  #recording the time the execution ended
print(f"time to find cosine pi 1 million times: {end-start}s")#Calculating and
printing cosine function execution time
```

Output:

```
C:\Users\PC\Desktop\Math\randompoints\venv\Scripts\python.exe C:/Users/PC/Desktop/Math/randompoints/speed.py
time to square 3.14 2 million times: 0.06104279999999994s
time to find cosine 3.14 1 million times: 0.2338947s
```

[2] Pipelining is a computational techniques where multiple instructions are paralleled during execution.

Appendix 2: Python implementation of rejection sampling method

```
import random  #random library simulates randomness to high precision.
import math
import matplotlib.pyplot as plt #Matplotlib is a python plotting library for data
visualization
select = random.Random()       #redefining random function with a variable 'select'

def rejection_sampling(number_of_points, radius): #define a function to uniformly generate n
random points in a circle given the radius
    distances = []    #creating an empty list ‘distances’ where will record the distance to
the center for every point
    x = [] # creating an empty list where we will store our x-coordinates
    y = [] # creating an empty list where we will store our y-coordinates
    points_obtained = 0   #At the start the points obtained is set zero
    trials = 0            #At the start the number of trials is set to zero

    while (points_obtained < number_of_points): # conditions for the loop to stop once the
desired number of points have been generated
        x_random = select.random() * 2*radius-radius  # using random python library to
generate a random x-coordinate between -radius and radius (-1 and 1)
        y_random = select.random() * 2*radius-radius # Subtracting radius from the results
enable us to obtain points in all 4 quadrants
        trials += 1                       # recording the number of trials

        if x_random ** 2 + y_random ** 2 < radius**2: # restricting to points within the
circle by using the equation of the circle
            points_obtained += 1           # recording the number of points obtained
            x.append(x_random), y.append(y_random) # storing the obtained coordinates of the
point into the corresponding x and y lists
            distances.append((x_random ** 2 + y_random ** 2)**0.5) #recording the distance of
the obtained point from the center

    #The following two lines are executed once the desired number of points have been
obtained, # x and y coordinated are transformed into rows and a scatter plot is then produced
for visualization
    plt.scatter(x, y,1) #plotting the points obtained on a scatter plot for visualization
    plt.show()         # Showing the plot

    plt.hist(distances, 55)  #plotting the histogram to show the expected frequency
distribution of distances of the points
    plt.show()               #showing the histogram

rejection_sampling(7000, 1) # Calling the function to produce 7000 points within a circle with
a radius of 1 units, by rejection sampling
```

Appendix 3: Python implementation of sampling on polar coordinates

```
def sampling_on_polar(number_of_points, radius): #defining a function to generate
uniform points on polar coordinates

    x = []    # creating an empty list where we will store our x-coordinates
    y = []    # creating an empty list where we will store our y-coordinates
    distances = []  # creating an empty list ‘distances’ where will record the
distance to the center for every point
    points_obtained=0 #At the start the obtained points is set to zero

    while(points_obtained<number_of_points): #conditions for the loop to stop once
the desired number of points have been generated
        r = select.random()  #selecting a random radius between 0 and 1
        # r = math.sqrt(select.random()*radius) # using the relationship obtained in
inverse transforming to obtained uniform density
        theta =  (select.random()*2*math.pi)    #selecting random angle theta between
zero and pi

        x_random = r*math.cos(theta)         #Converting to x cartesian coordinate
from polar coordinates
        y_random = r*math.sin(theta)         #Converting to x cartesian coordinate
from polar coordinates
        x.append(x_random), y.append(y_random) #storing the obtained coordinates into
their corresponding y and x lists
        points_obtained+=1

        distances.append(r) #recodring the distance from the point obtained to the
center

    plt.scatter(x, y,0.5)   # Visualizing the point distribution in a scatter plot
    plt.show()              # displaying the plot in a graphic window

    plt.xlabel("Distance from the center") #labelling the x-axis
    plt.ylabel("Density (number of points)") #labelling the y-axis
    plt.hist(distances)                      #plotting the histogram of distances
frequency distribution
    plt.show()                              #displaying the plot in a graphic window
#
sampling_on_polar(1000000, 1)     #Executing the function to sample of polar
coordinates
```

Appendix 4: Visualizing the relationship between points density and distance from the center given constant number of points per radius

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

def circle_points(r, n):
    points = []
    for r, n in zip(r, n):
        t = np.linspace(0, 2*np.pi, n, endpoint=True) #finding evenly
spaced points from 0 to n
        h = (2*np.pi*r)/n #space between each point in meters
        x = r * np.sin(t) #converting to circle coordinate on x-axis
        y = r * np.cos(t)
        for i in range(n):
            points.append([x[i],y[i]])
    return points
r = []
n = []
for i in range(0, 30):
    r.append(i*5)
for i in range(0, 30):
    n.append(70)

print(len(r))
print(len(n))

x,y=zip(*circle_points(r,n))
plt.scatter(x,y,30)
plt.axis('equal')
plt.show()
```

Appendix 5: Comparing the executing speed

Calculating the speed in seconds taken to generate 8 million uniform points in a circle by rejection sampling method.

```
start = time.perf_counter() #recording the starting time
rejection_sampling(8000000, 1) # producing 7000 points within a circle with a
radius of 1 units, by rejection sampling
end = time.perf_counter() #recording the finishing time
print(f"Time take to reject-sample 8 million points:{end-start}s") #The execution
speed is the difference between the end and starting time
```

Calculating the speed in seconds taken to generate 8 million uniform points in a circle sampling on polar coordinate method.

```
start = time.perf_counter() #recording the starting time
sampling_on_polar(8000000, 1)     #Executing the function to sample of polar
coordinates
end = time.perf_counter()#recording the finishing time
print(f"Time take sample 8 million points with polar coordinate method:{end-
start}s")#The execution speed is the difference between the end and starting time
```

THE RESULTS OF EXECUTING TEST COMPARISON