CS: Model for plotting points on a circle

Circles: Plotting points on a circular graph with

manipulatable distribution

Table of Contents

Introduction

Over the quarantine, I have developed a profound enjoyment and interest in gaming. Gaming is an outlet, a place to have fun with friends, practice competition, and immerse yourself in the work of a game developer. This interest in gaming and my background in computer science was a catalyst in my game development journey. In particular, I was trying to make a dart game where the objective is to play against a bot (a type of artificial intelligence designed to play a video game in the place of a human) with selectable difficulties. Quickly, I encountered a problem: I did not know how to get the AI to throw these darts randomly and in a manipulatable fashion. The only way I could think of was rejection sampling; it would be quite difficult to affect the placement of the darts on the board with this method. I could have designed the AI to aim for specific areas on the board depending on the difficulty, but this would make the game boring and predictable. This roadblock made me want to explore different methods of plotting points on a circle to manipulate the distribution. Irregular patterns have always interested me, driving me to want to fully understand how to manipulate these patterns to my desired outcome.

This IA aims to find a method of randomly plotting points on a circle while giving control of the distribution. It will also find the most efficient algorithm possible to plot these points in python, a popular programming language I will use in this exploration. This paper will explore two methods: rejection sampling and polar coordinate system. A program will be made for each method to see how they work under realistic conditions. Uniform distribution will be derived, as once it is possible to create a uniform distribution, it is possible to create any distribution by simply adding some scale factor.

Rejection Sampling

The first method to plot points, which is also the most intuitive, is rejection sampling. Rejection sampling works by choosing two continuous values from 0 - 2 represented by x and y and plots these points on a graph. However, these points will not always be within the circle as the area of the circle does not encompass the whole area of the square (Fig. 1). To solve this problem, a method to only include points that exist within the circle is required. The center of the circle is at (1, 1), and by definition, the bounds of a circle are defined as exactly radius units from the center of the circle. This makes it so that any point within the radius of 1 unit from that point will be within the bounds of the circle. The distance between two points formula can check if a particular point in the square is also in the circle. Looking at the figure below (Fig. 1), the area that satisfies the condition is marked in green, while the area marked in red represents the area where a point would be rejected.

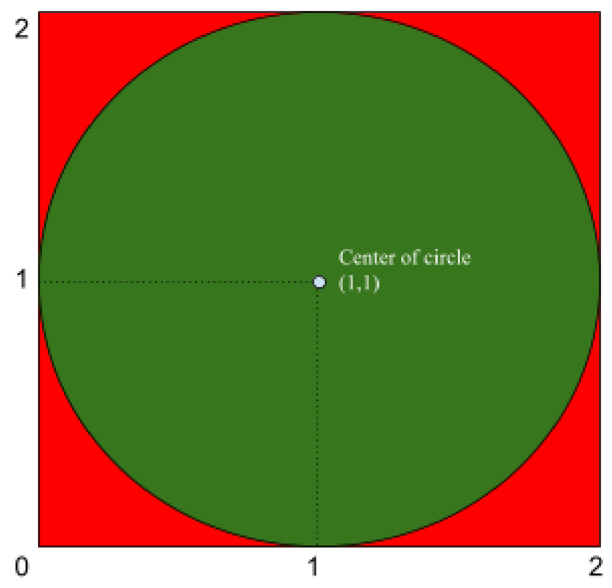

Fig. 1

Using this method to plot 3141 points looks like this:

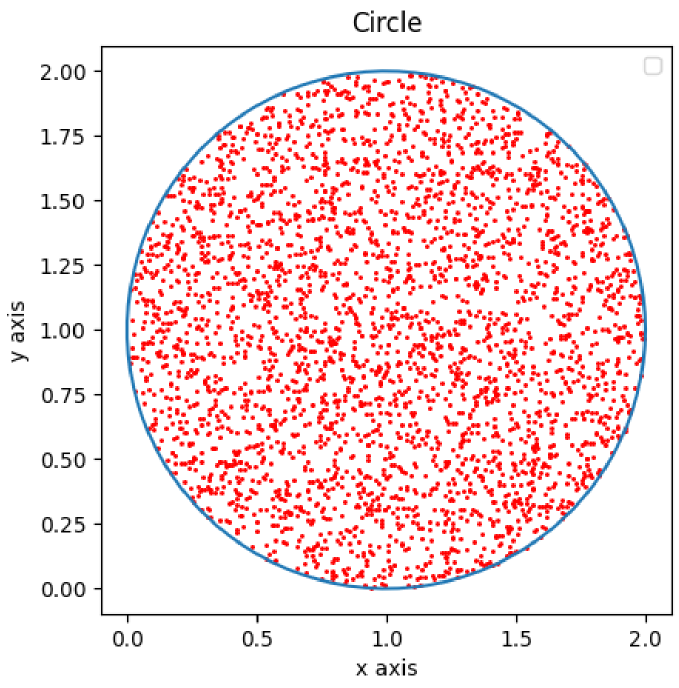

Fig. 2

As seen above (Fig. 2), all of the points are within the circle thanks to the employment of rejection sampling. Within the circle, there is clearly a uniform distribution. Rejection sampling would be suitable for an easy difficulty as there is an equal chance for the darts to land anywhere on the circle. The main issue is that it is inefficient, as not all the points will land in the circle on the first try; this is a source of inefficiency that would be better to remove. The success rate of this method can be found by dividing the area of the circle ($\pi * 1^2$) (the desired outcome), by the area of the square (2^2) (the total area in which a point could be). Using this method, solving for the success rate given that the radius of the circle is one and the square has side lengths two:

$$= \frac{\pi\ (1)^2}{2^2}$$

$$= \frac{\pi}{4} \approx 0.786$$

This means that the chance of a point landing within the circle on the first try is 78.6%. Also, because the chance of an event not occurring, is the one minus chance of it occurring, the chance this algorithm with fail *n* times can be written as:

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

Using this expression, a table for the probability of the algorithm having to loop *n* times can be created:

Table 1

Value of n	Probability algorithm will fail
1	$(1-\frac{\pi}{4})^n \approx 0.214$
2	0.046
3	0.0099

As can be seen in the table above (Table 1), the chance a single point will need to loop even two times is meagre, sitting at 4.6%, and this goes down to 1% when considering three loops. Now it is essential to find out how many times this algorithm will need to loop for each point on average. This number, times the number of points (*n)* would represent the total number of loops that would be expected for *n* points. To get this, the reciprocal of the success rate can be taken, which would be:

$$=\frac{1}{\frac{\pi}{4}}$$

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

$\approx\ 1.273$

This means that on average, each point will have to loop 1.273 times before that point can be plotted on the circle. This means that for n points, on average the computer will have to loop $1.273n$ times.

Polar coordinate system

While the rejection sampling method worked, it could not always choose a valid point on the first try. This would lead to higher uncertainty, and increase the amount of time required on average. To remedy this, the use of the polar coordinate system would be beneficial. This system plots points by selecting a distance from an origin and a radius about the polar axis, instead of using two perpendicular axes that meet at an origin, which lends itself more to rectangles. This means that as long as the distance from the origin is less than one unit, the point must be within the circle's bounds. From an intuitive perspective, the polar coordinate system makes much more sense to use. In this case, the aim is to produce points on a circle, so it makes sense to employ a coordinate system that works with the properties of a circle; radius and angles.

Using these principles, a program was made that chooses 3141 points randomly by selecting a distance from the origin of: $0 < \text{distance} < 1$; and an angle theta of: $0 < \text{Theta} < 2\pi$. The main issue with this is that there is no way to plot a point with polar coordinates directly, as the library used to plot the points in python does not support polar coordinates. To get around this, a method to convert polar coordinates to cartesian coordinates was required.

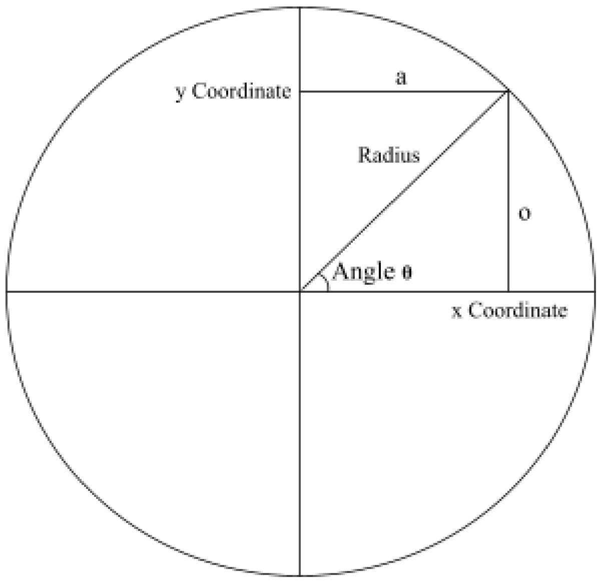

Fig. 3

Looking at the diagram above (Fig. 3), the radius is known, as it is equal to the distance from the origin, and in this case, is equal to the hypotenuse. The angle θ is also known, as it is one also of the randomized variables. Using the sin and cos functions the x adjacent and y opposite coordinates can be solved for:

x- Coordinate, solving for a:

$cos\theta = \frac{a}{h}$

$hcos\theta = a$

y- Coordinate, solving for o

$sin\theta = \frac{o}{h}$

$hsin\theta = o$

Using these expressions, the polar coordinates generated by the polar algorithm can now be converted to cartesian coordinates and plotted in python. Plotting 3141 points on a graph looks as follows:

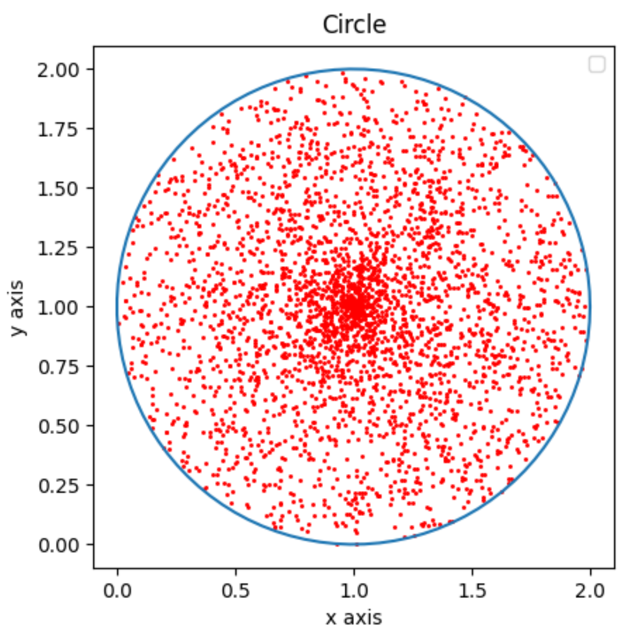

Fig. 4

One defining feature of this graph (Fig. 4) is the concentration of points near the center. While this is good for higher difficulties knowing how to manipulate the distribution would make the algorithm more versatile. To better understand how to manipulate this distribution, it will help to see how to make it uniform first. The issue is not the angle theta, as there is an even distribution of points around the circle. This means that the problem must lie in the distance from the origin. The random number generator that is being used for this in python has a uniform distribution. Meaning there is an equal chance for any number in the range to be chosen. Using this logic, it makes sense that each ring on the circle would also have an equal number of points on it.

Uniform Distribution with Polar system

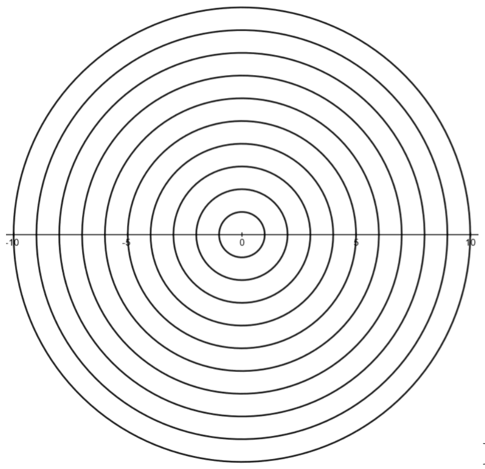

Fig. 5

Notice on the diagram above (Fig. 5), even though the radius itself is uniform, the area of each ring is different. Specifically, as the ring goes farther from the origin (radius increases), the area in which all the points are distributed also increases, decreasing the density of points in the outer rings. This means, naturally, as one looks outwards in the circle, there would be a diminishing effect, even though the number of points on each ring is equal.

Knowing the issue lies in the selection of radius, or r, the probability density of r can be graphed using a probability density function or PDF. Taking the integral of this function will give the probability that the random continuous variable r will be in a given interval.

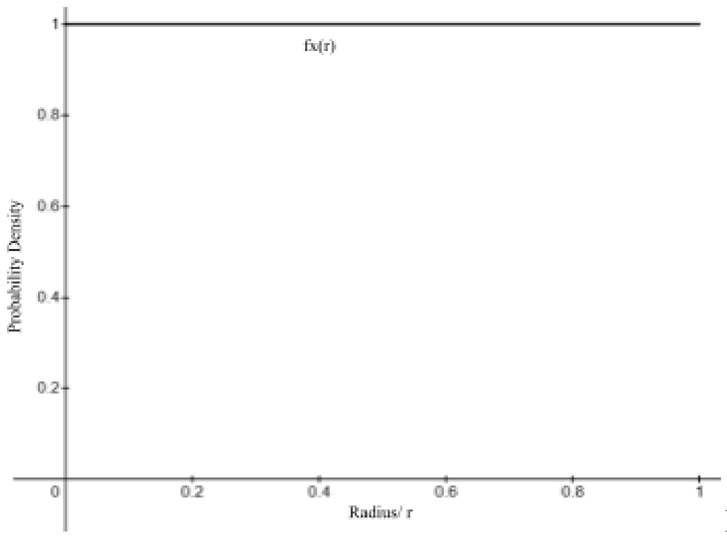

Fig. 6

Currently, this graph (Fig. 6) is a horizontal line from 0 to 1 as, due to the uniform random variable generator, there is an equal probability that any value of r will be chosen. As has been established earlier, this probability density function (PDF) doesn't work. Therefore a working PDF must be derived such that the points will be uniformly distributed.

Returning to the ring explanation, it is known that each interval of radius currently has an equal number of points. The current distribution doesn't consider the circumference of these rings, which can be thought of as infinitely small rings. Knowing that the circumference grows linearly with radius $C = 2\pi r$ it can be deduced that the number of points should also grow linearly with the radius. Knowing this, the PDF can be written as, $fx(r) = mr$. Now, the slope of this graph must be found.

The probability for any given radius to exist in the interval 0 to 1 is 100%; this can be notated as $P\{0 \leq r \leq 1\} = 1$. This is known because the only values r can possibly take, lie in this range (0 - 1). Given this, the integral of $fx(r) = mr$ in the range $\{0 \leq r \leq 1\}$ must be equal to 1, as the definite integral of a PDF function is equal to the probability that a point will fall in the range of the bounds of the integral.

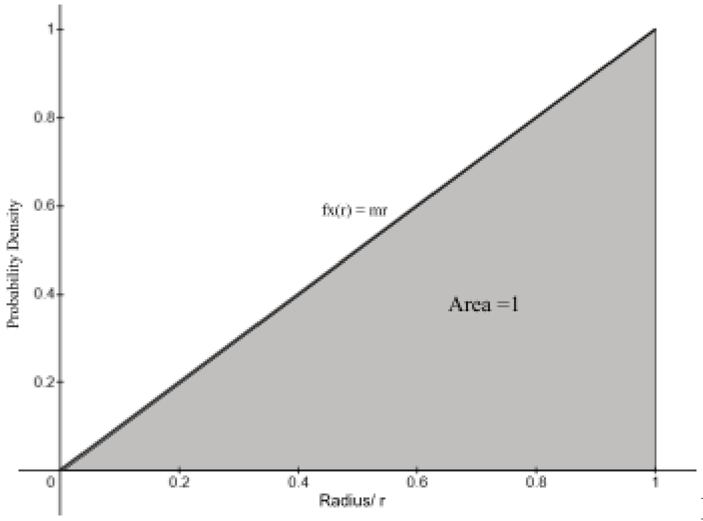

Fig. 7

Now, to solve for the slope, an integral can be set up solving for m:

$$\int_0^1 mr\,dr = 1$$

$$m\int_0^1 r\,dr = 1$$

$$[(\tfrac{1}{2})r^2]_0^1 = \frac{1}{m}$$

$$\frac{1}{2} = \frac{1}{m}$$

$$m = 2$$

Given $m = 2$, $fx(r)$ can now be written as $fx(r) = 2r$. An interesting side note is that this integral will always equal 1, meaning it can be used for a distribution of any degree. Also, there will never be a constant of integration that isn't zero as this function must pass through (0,0). This would be done by entering a function $fx(r)$ of different degrees, resulting in an altered distribution. This would be very useful to create PDFs for different difficulties, for example.

Now that the PDF that would make this distribution uniform has been derived, a transformation is required that would make the uniform random number generator provided by python give this linear distribution. To do this, a cumulative distribution function or CDF that will be denoted as $Fx(r)$ will be used. This function represents the integral of the previously derived PDF function, knowing this, the CDF function can be derived as:

$$Fx(r) = \int fx(r)dx$$

$$= \int 2rdr$$

$= r^2$

$= P(X \leq r)$

This CDF function $Fx(r) = r^2$ represents the probability that any given output X will be less than or equal to its input. Using this now a method called inverse transform sampling can be implemented to find this final transformation. This method uses a uniform distribution, such as the one provided by python, and transforms it into one that has a different distribution. While the random variable X does not have a uniform distribution, the probabilities themselves, marked on the y-axis, do. As such, let's see what happens when those y-values are treated as the input. Essentially taking the inverse of the CDF function. Then by graphing the CDF function, and taking 10 evenly spaced y-values and looking at their corresponding x-values the effect can be seen.

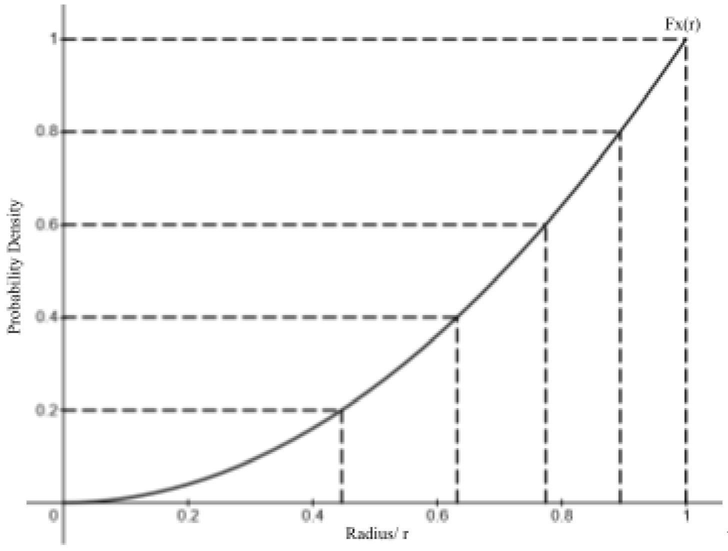

Fig. 8

As is evident in the graph above (Fig. 8), this distribution has had the desired effect of concentrating higher densities of points at the higher radii. Beginning at the lower radii, the cumulative distribution is much less, increasing with radius. While this works, there is better proof for why plugging a uniform random variable into the inverse of the CDF yields the desired distribution.

So far, this paper has discussed that the CDF represents the probability of some random variable u being equal to or less than the input r. To make this more clear, consider the PDF of a uniform random variable. The graph of the PDF, $fu(r) = 1$ is a horizontal line from 0 to 1 at $y = 1$. It makes sense then that the CDF, the integral of the PDF, would be $Fu(r) = r$. The definition of a CDF dictates that the probability of u being less than or equal to r is r, notated as $P(u \leq r) = r$. If 0.5 is substituted in for r, the following occurs:

$fu(0.5) = 1$

$Fu(0.5) = 0.5$

$P(u \leq 0.5) = 0.5$

This makes sense because the probability of a uniform random variable being less than or equal to 0.5 is 0.5 since it is exactly in the middle of 0 and 1. Returning to the inverse transform sampling, it can now be said that $Fx(r)$ is equal to the probability that a uniform random variable u will be less than or equal to $Fx(r)$. Then the inverse of the CDF can be applied to both sides of the inequality leading to two inequalities that are both less than or equal to r. Therefore X, the random variable with the desired distribution, is equal to the inverse of the CDF being applied to a uniform random variable:

$$\begin{aligned} Fx(r) &= P(X \leq r) \\ &= P(u \leq Fx(r)) \\ &= P(Fx^{-1}(u) \leq r) \\ X &= Fx^{-1}(u) \end{aligned}$$

The inverse of the derived CDF $Fx(r) = r^2$ can be found by doing the following:

$Fx(r) = r^2$

$y = r^2$

$r = y^2$

$\sqrt{r} = y$

Finally all that is left to do is to check if this worked by plugging this transformation into python to see the plot of 3141 points:

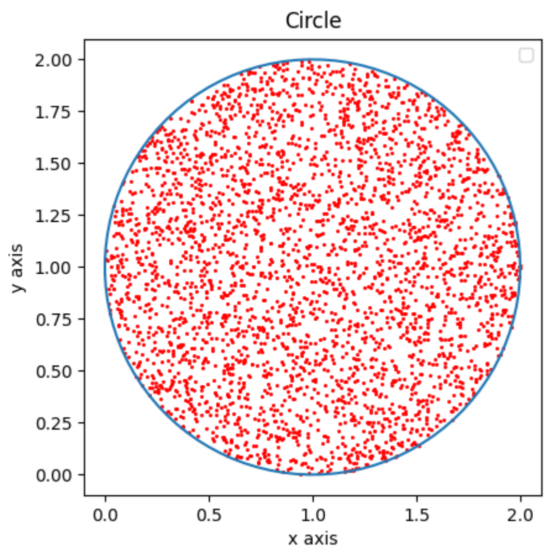

Fig. 9

From this plot (Fig. 9), it is clear that this method of selecting the desired PDF to solve for a transformation worked. In the plot every point seems to be equally distributed along the angles, as well as the radii, due to the transformation applied through the inverse transform sampling. Building on this, the question arises, is it possible to to use this method to create a manipulatable distribution?

Using Polar system for manipulatable distribution

This means this method of making a PDF of a desired distribution with degree n, then solving for a CDF by setting the integral equal to 1, and finally using the CDF to solve for a transformation leads to the desired result. Using this same method different transformations can be solved for with different densities. For example solving for a PDF with degree 2 and 3, the following can be calculated:

$\int_0^1 mr^2\,dr = 1$ $m\int_0^1 r^2\,dr = 1$ $[(\frac{1}{3})r^3]_0^1 = \frac{1}{m}$ $((\frac{1}{3})1^3) - ((\frac{1}{3})0^3) = \frac{1}{m}$ $\frac{1}{3} = \frac{1}{m}$ $m = 3$	$\int_0^1 mr^3\,dr = 1$ $m\int_0^1 r^3\,dr = 1$ $[(\frac{1}{4})r^4]_0^1 = \frac{1}{m}$ $((\frac{1}{4})1^4) - ((\frac{1}{4})0^4) = \frac{1}{m}$ $\frac{1}{4} = \frac{1}{m}$ $m = 4$
$\int_0^r 3r^2 dr$ $= r^3 - 0^3$ $= r^3$ Inverse: $y = r^3$ $r = y^3$ $y = \sqrt[3]{r}$	$\int_0^r 4r^3 dr$ $= r^4 - 0^4$ $= r^4$ Inverse: $y = r^4$ $r = y^4$ $y = \sqrt[4]{r}$

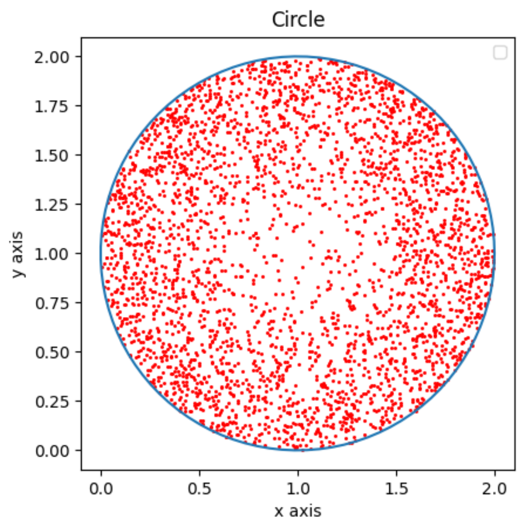

Fig. 10

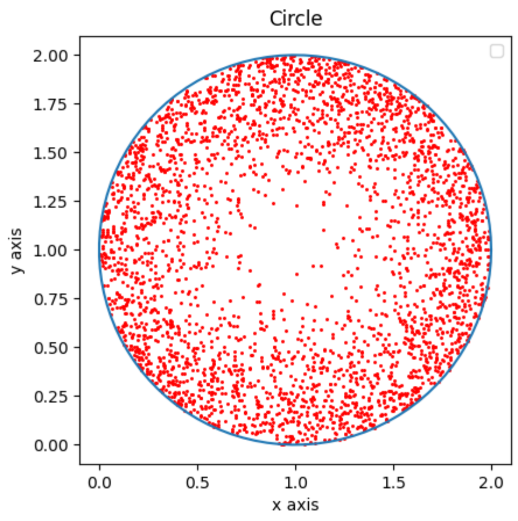

Fig. 11

As can be seen by the calculation of the PDF, CDF, and final transformation, there is a pattern in the input degree of the PDF and the final transformation. With this data a hypothesis can be made, for each PDF with degree n, the final transformation is $\sqrt[(n+1)]{r}$. Also, as the degree of the PDF increases, the points get concentrated increasingly towards the outside of the circle as can be seen in Figure 10 and 11.

Proof

This is true for n = 1 as shown earlier in this paper. First a general term for the slope can be derived by the following:

$$\int_0^1 mr^n \, dr = 1 \qquad n \in R$$

$$m\int_0^1 r^n \, dr = 1$$

$$m\left[\frac{1}{n+1}r^{n+1}\right]_0^1 = 1$$

$$((\frac{1}{n+1})1^{n+1}) - ((\frac{1}{n+1})0^{n+1}) = \frac{1}{m}$$

$$\frac{1}{n+1} = \frac{1}{m}$$

$$m = n + 1$$

Using this slope the integral can be taken with respect to r to derive the CDF:

$$\int (n + 1)r^{n}dr$$

$$= r^{n+1}$$

Inverse:

$$Fx(x) = y = r^{n+1}$$
$$Fx^{-1}(x) = r = y^{n+1}$$
$$Fx^{-1}(x) = \sqrt[n+1]{r}$$

Therefore it has been directly proven that for any degree n of radius r, where $n \in R,$ the transformation derived from the CDF of the PDF, the transformation is $\sqrt[n+1]{r}$.

Evaluation

In any programming project, efficiency is critical, as it is responsible for which type of computers will be able to run any given program. With this in mind, it's essential to evaluate the available options to see which methods lead to better results. In this paper, there were two methods discussed. First was rejection sampling, which has to run on average 1.273 times to get each point, and utilized the cartesian system. However, only a uniform distribution was possible using this method, so this distribution will be used to compare for all the tests. The second is the polar method that uses inverse transform sampling to produce a transformation for the desired distribution. First, each method was used ten times to plot 3141 points and the times, obtained by using the time library, are as shown in the table below:

Table 2

Test#	Rejection Sampling (seconds)	Polar (seconds)
1	0.01400136947631836	0.03400826454162598
2	0.01400303840637207	0.034006357192993164
3	0.01400303840637207	0.03400707244873047
4	0.013002634048461914	0.03400778770446777
5	0.014003276824951172	0.033007144927978516
6	0.014003753662109375	0.03300738334655762
7	0.014003276824951172	0.03400778770446777
8	0.015001773834228516	0.03400754928588867
9	0.01300501823425293	0.03400731086730957
10	0.014002323150634766	0.03300642967224121

Using this data (Table 2), the mean time and standard deviation can be calculated:

Table 3

	Mean	Standard deviation
Rejection Sampling	0.013902950286865 Seconds	0.00053797334540567 Seconds
Polar	0.033707308769226 Seconds	0.0004584691830947 Seconds

As can be concluded from the data above (Table 3), rejection sampling was the more efficient method of plotting a uniform distribution. Looking at the mean times for each method, the polar method that used the inverse transform sampling took more than two times more time on average to achieve the same result. This is a surprise because, as discussed earlier, the rejection method has to run 1.27 times per point while the polar method only has to run once. A theory as to the reason for this, is the processing required to take the square root of the radius and then convert each set of polar coordinates to cartesian coordinates. These processes for each point outweigh the computing power needed to run more times to get a valid point as with rejection sampling.

Another important note is that the polar method had a lower standard deviation. This makes sense because while it may take more processing, this method has the advantage of being consistent. The rejection method had to cycle through points until it reached one that worked, this was a random process that had to take place for every point. On the other hand with polar sampling, the only random process is the selection of the distance from the origin, and the angle. Besides these two factors, all the computation involved is concrete mathematical evaluation. This is why there is a lower deviation on average.

Limitations

A big issue with doing any programming-related evaluation is the reliance on libraries created by third parties. If one were to make everything from scratch, the process could take years, and it's hard to evaluate every function being used in any given program. This is a significant limitation in that it's challenging to know whether or not the ideal/most efficient functions were used to compute what was required. One such example was the library used to plot the points, 'matplotlib,' which uses the cartesian system to plot points; this is the reason the conversion from polar coordinates to cartesian coordinates was required. This limitation, that the library caused, made it necessary to convert these coordinates, which took more processing per point. Also, the data collected for the evaluation was done on a high-power computer. Every computer is different, and the time required to complete various operations is thus also different. This being said, the results should be relatively similar across devices, and the conclusions would be similar.

Taking this Further

There was one main question brought out by this IA, this being why the polar method ended up being less time efficient than the rejection method. In the future I plan to explore how time complexity is calculated, thus the real reason any method would be more efficient than another. Also, in this exploration there was a method that I was unable to include. This method included imagining a circle as the cumulation of an infinite number of infinitely small triangles rotated around a point. By choosing a random triangle, and a random location on this triangle it would be possible to plot a random point on the circle. This just goes to show that there are countless methods to solve the same problem, each with their own drawbacks and benefits. In this exploration there were only two methods discussed, however in the future I hope to be able to expand on this exploration. Particularly by looking deeper into the other methods of solving this problem that are both easy to manipulate, and efficient to run on computers. Overall this exploration was a success, two methods were found along with their effectiveness, as well as their particular use cases.

This math done in this exploration is very versatile. One such example is it's application is the study of epidemiology where it can be used to model circular spaces and how disease can spread through these spaces with people set up in random places with given distributions. Also, as was the rationale for this exploration, it also has many uses for game development far beyond just dart games. One such example is the globally popular first person shooter "Valorant" in which many different guns have unique, circular, and random bullet spreads. Using these methods of randomizing points on a circle, Valorant could optimize their game to make it run better on more computers making it more accessible to millions of people around the globe. To conclude, this was an example of using mathematics to compensate for a shortcoming of computer science, and together their ability to make something beautiful.

Works Cited

Hunter, John. “Pyplot Tutorial.” *Pyplot Tutorial - Matplotlib 3.5.0 Documentation*, 2012, https://matplotlib.org/stable/tutorials/introductory/pyplot.html.

Sigman, Karl. “1 Inverse Transform Method - Columbia University.” *Inverse Transform Method*, Columbia University, 2010, http://www.columbia.edu/~ks20/4404-Sigman/4404-Notes-ITM.pdf.

Wathall, Jennifer Chang. *Mathematics: Analysis and Approaches: Higher Level*. Oxford University Press, 2019.

Appendix A

The python code used to calculate and plot the points for both methods.

```
import random
import matplotlib.pyplot as plt
import math
import numpy as np
import time

#Graphing the circle
theta = np.linspace(0, 2*np.pi, 100)
radius = 1
a = radius*np.cos(theta)+radius
b = radius*np.sin(theta)+radius
figure, axes = plt.subplots(1)
axes.plot(a, b)
axes.set_aspect(1)
xPlot = []
yPlot = []


"""
Rejection Sampling: This method uses a simple algorithm to place the
points within the circle using a traditional cartesian plane.
for each point two random points between 0 and 1 are chosen for the x and
y values. Regularly this would lead to getting a square
distribution. In this case however, points that are over 0.5 units away
(using the distance between 2 2-D points formula) are
rejected and it looks for another point until this condition is met. The
result is an equally distributed plot in the shape of the circle.
"""
def rejectionSampling():
    while True: # This is used to make sure the point is whithin the
circle
        x = (random.random()) * 2
        y = (random.random()) * 2
        if (math.sqrt((radius-x)**2+(radius-y)**2)) < radius:
            return (x,y)
```

```
def basicPolar():

    r = random.uniform(0, radius) # Picks a random number for the distance
of the point from the origin
    angle = random.uniform(0,2*np.pi) # Picks a random number for the
angle of that point around the origin

    return(((r * np.cos(angle))+radius), (r * np.sin(angle)+radius))

def manipulatedPolar():
    r = math.sqrt(random.uniform(0, radius)) # Picks a random number for
the distance of the point from the origin but squares this value
    angle = random.uniform(0,2*np.pi) # Picks a random number for the
angle of that point around the origin

    return(((r * np.cos(angle))+radius), (r * np.sin(angle)+radius))
def manipulatablePolar():

    r =  (random.uniform(0, radius) ** (1.0/n)) # Picks a random number
for the distance of the point from the origin but squares this value
    angle = random.uniform(0,2*np.pi) # Picks a random number for the
angle of that point around the origin

    return(((r * np.cos(angle))+radius), (r * np.sin(angle)+radius))

# Graph selection

print('Which method would you like to use?\n1: Rejection sampling\n2:
Basic Polar method\n3: Uniform Polar\n4: Manipulatable Polar')
option = int(input())
points = int(input("How many points would you like? Preferably over 3141:
"))

startTime = time.time()
if (option == 1):
    for i in range(points):
        x,y = rejectionSampling()
        xPlot.append(x)
        yPlot.append(y)
elif (option == 2):
    for i in range(points):
        x,y = basicPolar()
        xPlot.append(x)
        yPlot.append(y)
elif (option == 3):
    for i in range(points):
        x,y = manipulatedPolar()
        xPlot.append(x)
        yPlot.append(y)
elif (option == 4):
    n = float(input("Which distribution would you like to use? 1 being
concentrated to the center and higher numbers being concentrated farther
from the center: "))
    for i in range(points):
        x,y = manipulatablePolar()
        xPlot.append(x)
        yPlot.append(y)
else:
    print("How did you manage that...")
endTime = time.time()
print("Time Elapsed: ", (endTime - startTime))


# Plotting points as a scatter plot
plt.scatter(xPlot, yPlot, color= "red", marker= ".", s=5)
# x-axis label
plt.xlabel('x axis')
# frequency label
plt.ylabel('y axis')
# plot title
plt.title('Circle')
# showing legend
plt.legend()
# function to show the plot
plt.show()
```