MATH 310 - Project

Aliza Rafique - ar05986, Aliza Saleem - al05435, Ayesha Syed - aa05444, Hafsa Irfan - hi05946

April 27, 2021

Question1

Solution 1.1

Implemented a function which calculates the expected value of the object's final position.
The object took 10 steps and this experiment was repeated 1000 times, to obtain several expected values and to make the trend in result obvious.

bins = 15
number of experiments = experiments = 1000
number of steps = n = 10

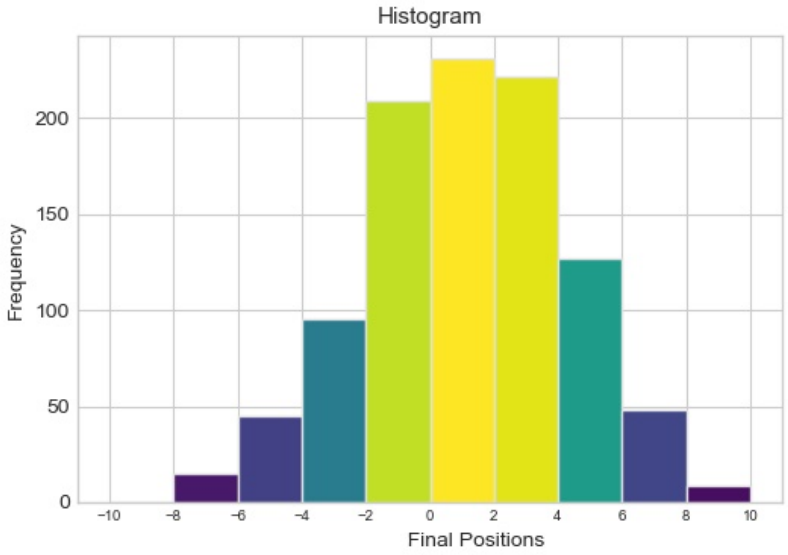

Figure 1:

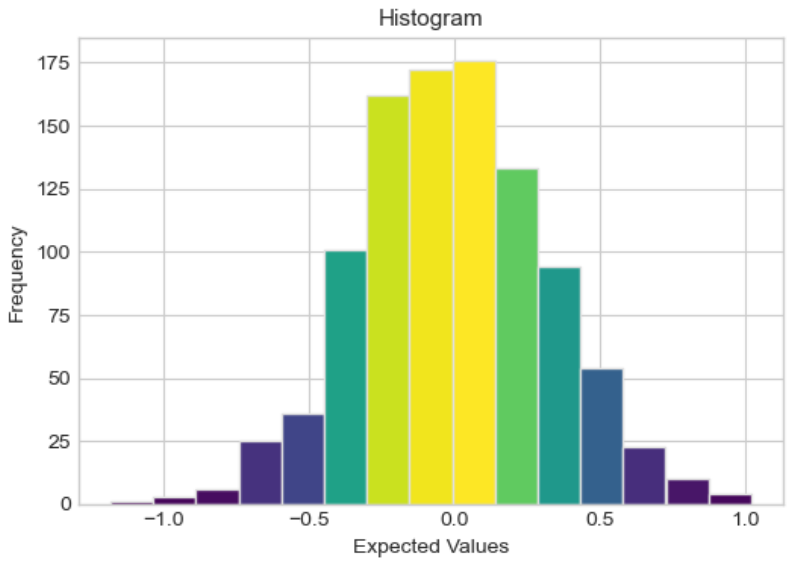

Figure 2:

---

bins = 20
number of experiments = 1000
number of steps = n = 100

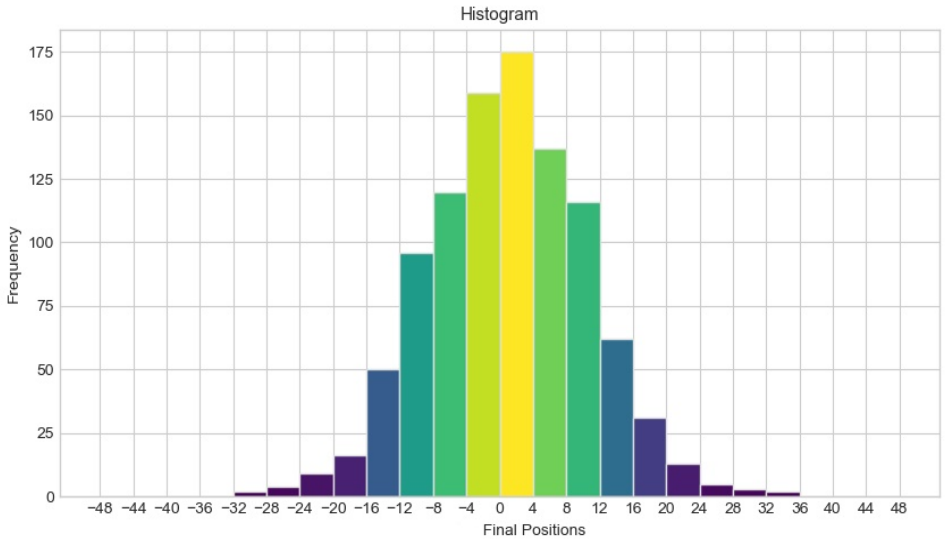

Figure 3:

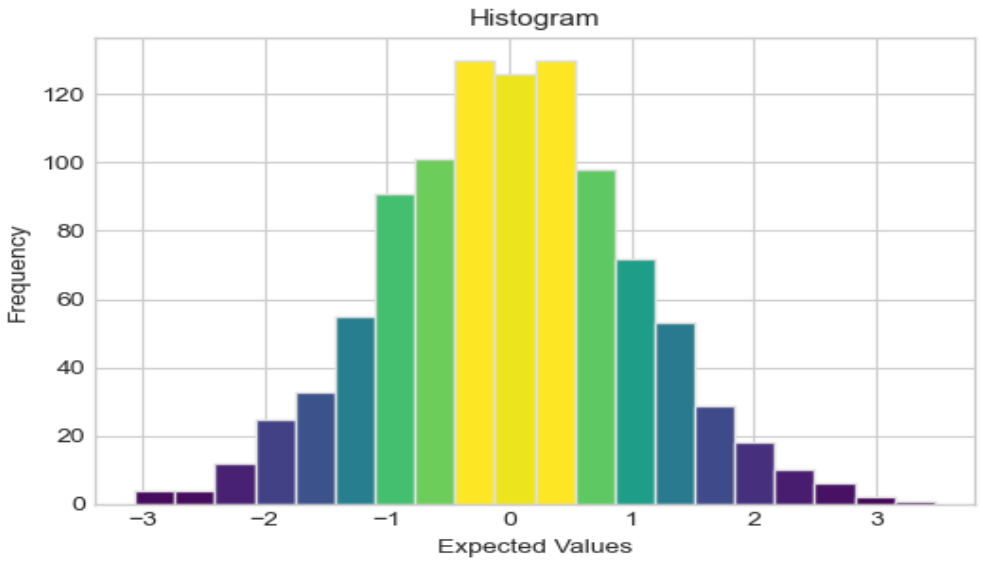

Figure 4:

It was observed from results of walks that on the odd turn, object's position is on odd numbers.On even turns object's position is on even numbers.

1. Histogram Results:

   - The walk's final position hovers around 0, when the walk is shorter. But the longer the walk, greater it spreads out. When we went from steps = 1 to steps = 10 (ie n = 1 to n = 10) in Figure 1, there was sufficient spreading out but much of it was concentrated around 0, but when the walk was made longer for example when steps = 100 (ie n = 100), we can see in the Figure 3 that the results are more spread out and form a more precise bell-shaped graph.
   - In Figure 3, after 100 steps, we can see that the green bars have the most frequency, so most of the time final position is between +10 and - 10 which is $\sqrt{100}$
   - From the Expected Values in Figure 3 (n = 10), it can be observed that the expected final position lies mostly around 0. It can be concluded that object moves about the same times ahead of 0 as less than 0 and hence the average is around 0.
   - From the Expected Values in Figure 4, it can be observed that with greater steps (n = 100), expected position also spreads out compared to Figure 3.

2. Using different values of p:

   - If we use p = 0.7, the graph becomes negatively skewed as it is more likely that the object moves to the right and so final expected position is more towards the right of the starting position.
   - However, p = 0.3 has just the opposite effect. The graph becomes positively skewed, because the object is more likely to move left than right due to the smaller probability of 0.3 of moving right.And so the expected final position is more likely to be on the left of starting position.

Solution 1.2

Implemented a function which calculates the expected value of the object's final position with the added constraint that going into the negative part of the number line is not allowed.

number of experiments = 1000
number of steps = n = 100

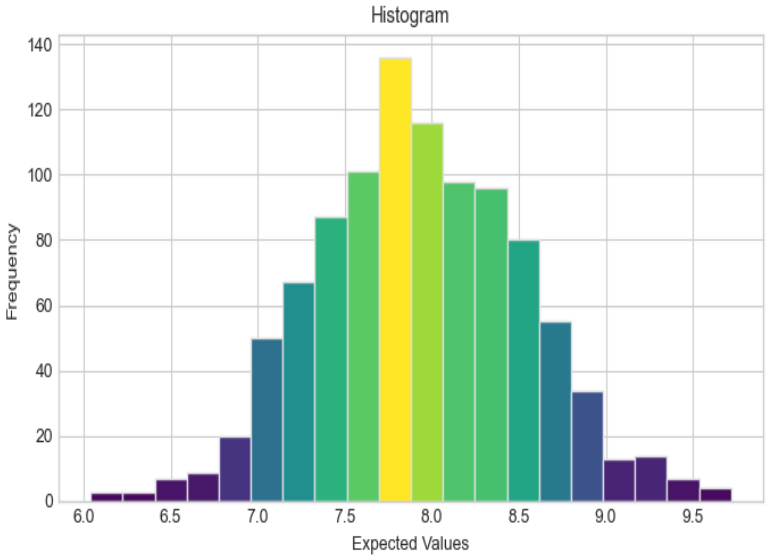

Figure 5:

- Expected Value is in the range 6 to 10
- Expected values are only positive due to the given constraint of object not being allowed to move left of 0.
- It can be observed that that there is much greater variance in expected values of 1.2 in Figure 5 (than in Figure 3 of 1.1) as we can see the graph is spread out in greater range of values.

Solution 1.3

Implemented a function which calculates expected values for the number of steps it takes for the two objects to meet separated by initial even steps.

Object A's Starting Position = 2
Object B's Starting Position = 4
Probability of A of moving right = 0.5
Probability of B of moving right = 0.5

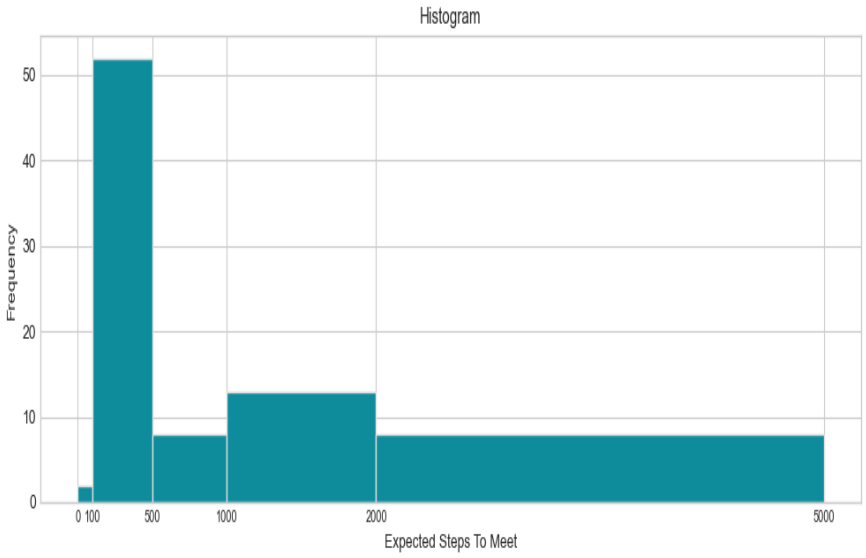

Figure 6:

There is much variation.

Question2

Solution 2.1

The code clearly accomplishes following the given distribution.It can observed that Y follows the exponential distribution with mean 1 as can be seen by the given distribution equation and the plotted graph with a much longer tail to the right-hand end of the distribution than to the left.

The following is the histogram with $bins = 10$

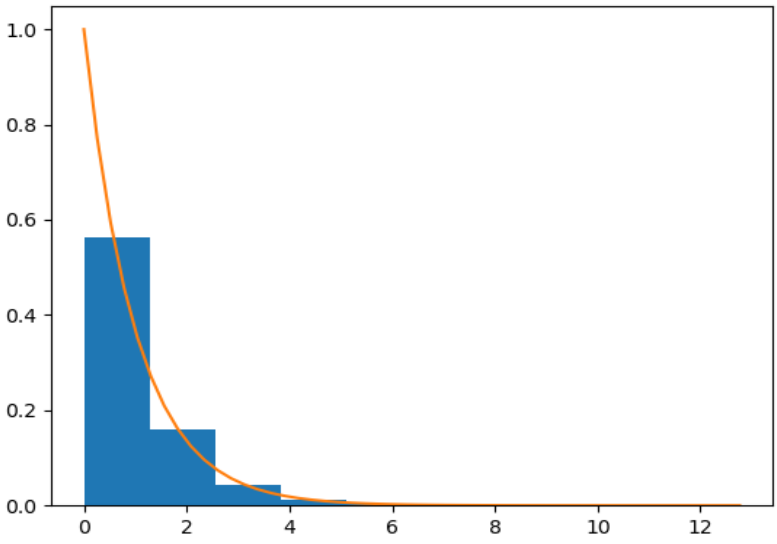

Figure 7:

The following is the histogram with $bins = 20$

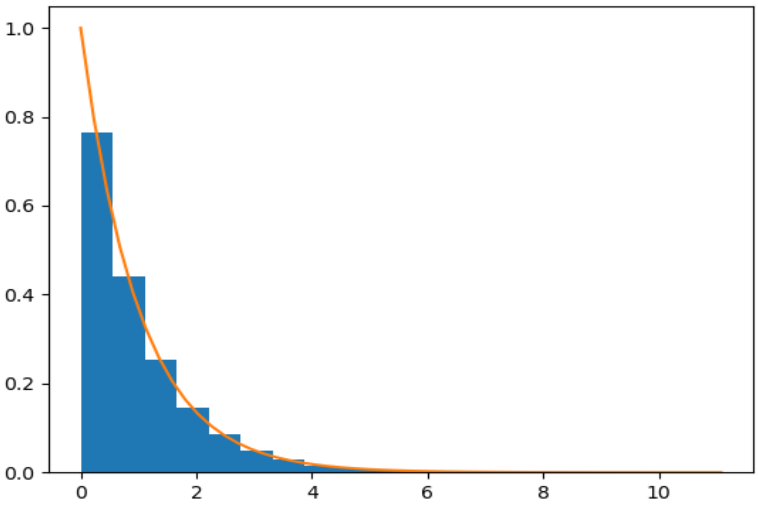

Figure 8:

The following is the histogram with $bins = 50$

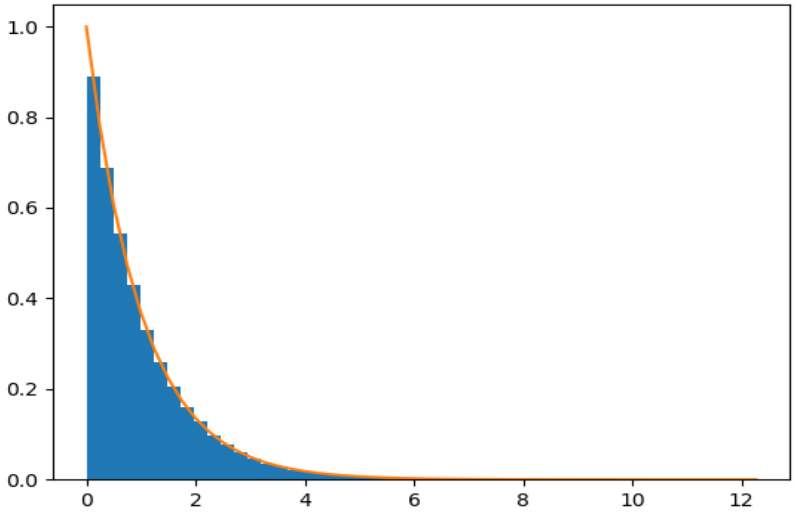

Figure 9:

The following is the histogram with $bins = 100$

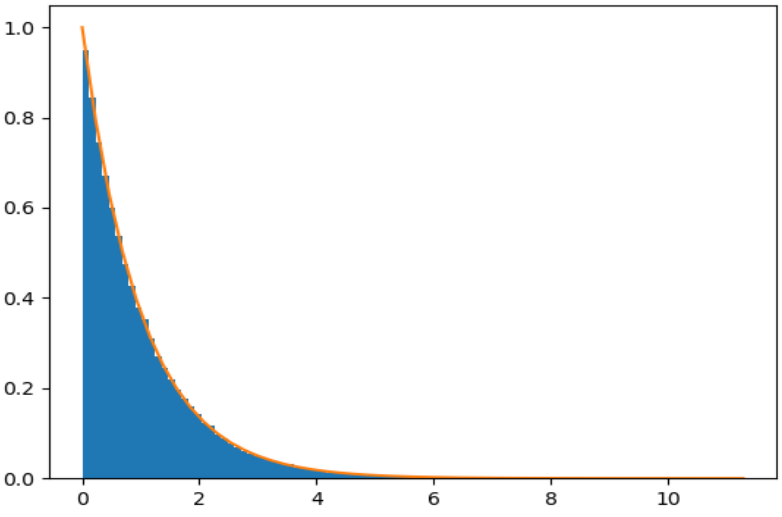

Figure 10:

The following is the histogram with $bins = 1000$

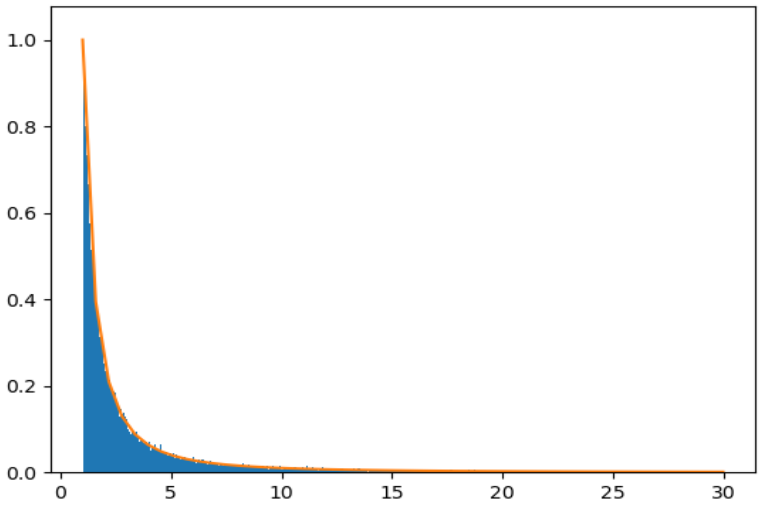

Figure 11:

- It can be observed from the trend in the histogram that as the value of bins increases, number of bars increase with decrease in width of each bar.

- The distribution starts becoming smoother as we increase the bins seen by the bars. From bins = 10 to 20 to 100 to 1000, it can be observed that the histogram becomes more and more similar to the underlying exponential distribution. And with bins = 1000, it gets quite a close approximation to the true distribution.

- Moreover, several observations for each of the bins show that the the histograms are more similar for larger bins.

Solution 2.2

As given in the code:

$$y = \frac{1}{1-x}$$
$$y(1-x) = 1$$
$$y - yx = 1$$
$$y - 1 = yx$$
$$yx = y - 1$$
$$x = \frac{y-1}{y}$$
$$x = 1 - \frac{1}{y}$$

Differentiating the RHS:

$$x = \frac{d}{dy}(1 - \frac{1}{y})$$
$$x = \int_1^y \frac{1}{y^2} dy$$
$$\int_1^y \frac{1}{y^2} dy = x$$

At limit 0 it is undefined, histogram starts at 1

It follows the distribution, $f_Y(y) = \frac{1}{y^2}$ for $y \geq 1$

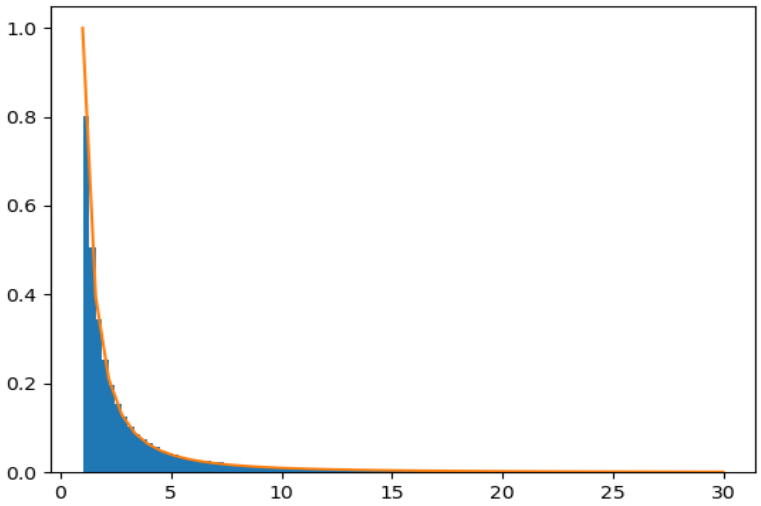

Figure 12:

The x value in the given code is chosen randomly anywhere between 0 and 1. Thus the variable x can take infinite values close to 1 which can make (1 - x) in the denominator very small and the whole fraction very large. And so the range will become infinite.
But lines 5-7 takes care of the infinite range by limiting the maximum value, allowing us to see the plotted bars. Thus lines 5-7 are very important because removing them shows no bar results and thus without them we won't be able to decipher the graph results.

Solution 2.3

The distribution given is

$$f_Y(y) = \frac{1}{y^3}; y \geq \sqrt{\frac{1}{2}}$$

$$P(Y < y) = P(X < x)$$

$$\int_{\sqrt{\frac{1}{2}}}^{y} \frac{1}{y^3} dy = x$$

$$\frac{1}{-2y^2}\Big|_{\sqrt{\frac{1}{2}}}^{y} = x$$

$$\frac{-1}{2y^2} - (-\frac{1}{2(\sqrt{\frac{1}{2}})^2}) = x$$

$$\frac{-1}{2y^2} + 1 = x$$

$$y = \sqrt{\frac{-1}{2(x-1)}}$$

Histogram And Line Plot for random variables from $f_Y(y) = \frac{1}{y^3}; y \geq \sqrt{\frac{1}{2}}$:

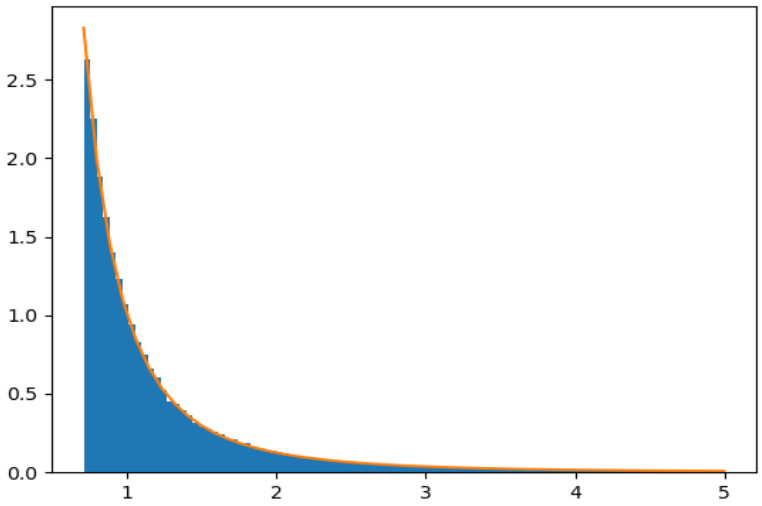

Figure 13:

Histogram for set of Expected Values:

Experiments = 100
Average of 100000 y values

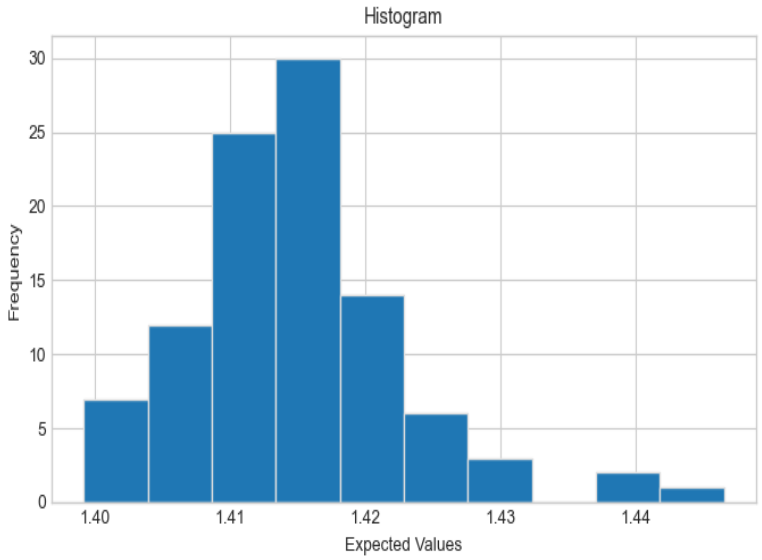

Figure 14:

Question 3

Solution 3.1 The following libraries were used in 3.1:

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

Listing 1: convertToXY

In this question we are considering $R = 1000$.

In this solution, the first function, *generateRandom* takes the value of R as an argument. A loop with a range of 1000 values is used to generate random values of radius for a range of 1 to R. In the same loop, 1000 theta values are generated with range 1 to 360.

```
def generateRandom(R):
    '''
    - Generates random values of radius and theta.
    - Calculates variance.
    args: Radius - integer
    returns: theta and radius list
    '''
    theta = []
    radius = []
    # Generating 1000 random radius and theta values.
    for i in range(1000):
        radius.append(random.randint(1,R))
        theta.append(random.randint(1,360))

    return radius, theta
```

Listing 2: generateRandom

Next, in the function *convertToXY*, these theta and radius values are converted to 1000 values in x and y plane by using the formulas:

$$x = r \cdot cos(theta)$$

$$y = r \cdot sin(theta)$$

```
def convertToXY(radius, theta):
    '''
    Converts radius and theta values to xy values for plotting.
    args: radius list (radius), theta list (theta)
    returns: x and y lists.
    '''

    x = []
    y = []
    for i in range(1000):
        x.append(radius[i] * np.cos(theta[i]))
        y.append(radius[i] * np.sin(theta[i]))
    return x, y
```

Listing 3: convertToXY

In the *plot* function, random points and the circle is being plotted using the matplotlib library.

```
def plot(R, x, y):
    '''
    Plots circle and random points.
    args: Radius (R), x and y value list for random points
    '''

    # plotting circle
    t = np.linspace(0, 2*np.pi, 150)
    a = R * np.cos(t)
    b = R * np.sin(t)
    fig, ax = plt.subplots(1)
    ax.plot(a, b)
    ax.set_aspect(1)
    # plotting points
    plt.scatter(x, y, marker='.')
    plt.show()
```

Listing 4: plot

Then, finally everything is integrated in the *main* function and the variance in theta and radius is calculated:

```
def main():
    R = 1000
    radius, theta = generateRandom(R)

    x , y = convertToXY(radius, theta)

    # calculating variance in x and y
```

```
    print("Variance in x values: ", np.var(x))
    print("Variance in y values: ", np.var(y))

    plot(R, x, y)
```

Listing 5: main

The result is as follows:

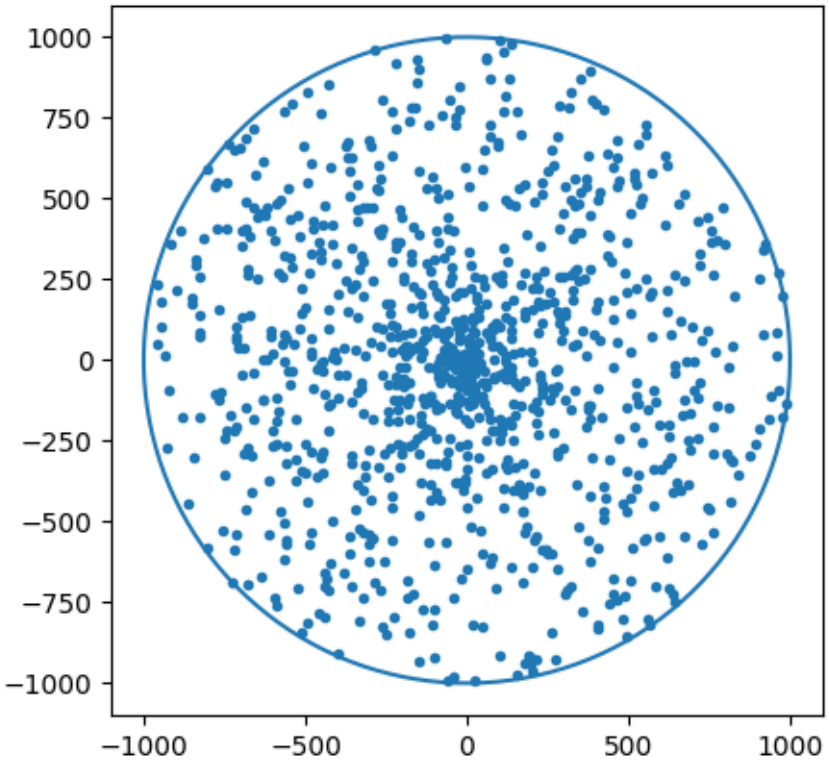

Figure 15: Circle plot for 3.1

The corresponding variance calculated:

Variance in x values: 170961.8866800192

Variance in y values: 165447.8954833384

Solution 3.2

The given method was implemented in the function, *generateRandomXY*. The pythagorean formula was used to check whether the distance of x and y values is less than the given R.

```
def generateRandomXY(R):
    '''
    Generates random x and y values.
    args: R - radius
    returns: x and y list.
    '''
    x, y = [], []
    for i in range(1000):
        ax = (random.randint(-R, R))
        wy = (random.randint(-R, R))
        if np.sqrt(ax**2 + wy**2) < R:
            x.append(ax)
            y.append(wy)
    return x, y
```

Listing 6: generateRandomXY

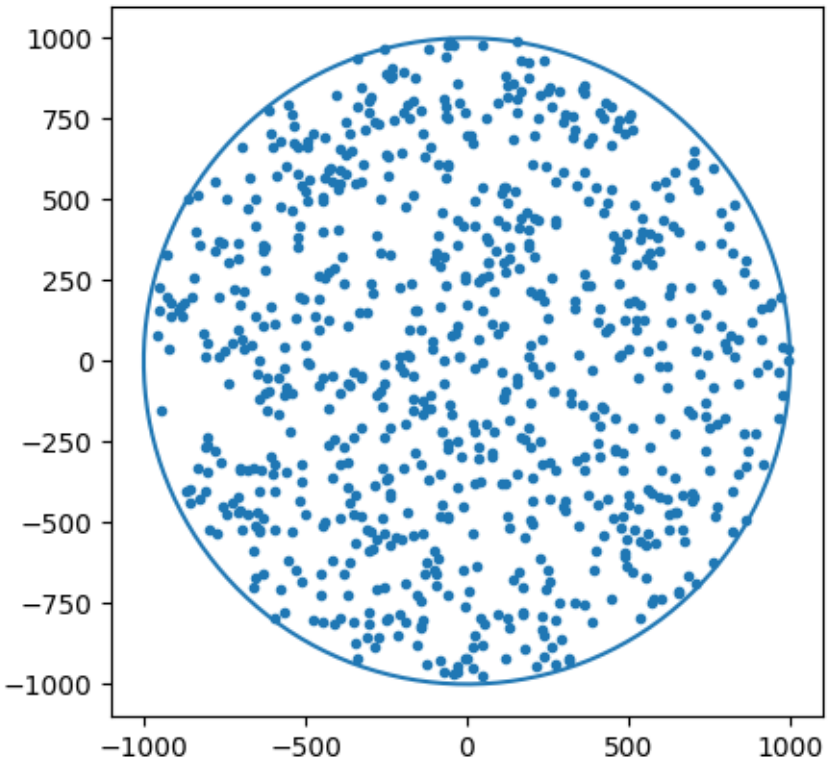

Figure 16: Circle plot for 3.2

The corresponding variance calculated:

Variance in x values: 227149.6579797342

Variance in y values: 255754.4683562264

The variance in both x and y values have increased significantly for the same number of total random points as compared to the previous part. This further indicates that the random points are greatly spread around the circle.

Solution 3.3

To apply the mentioned intuition, R is divided by 2 to create $R2$. The function *generateRandom* is modified in a way that now it keeps the count of points generated in $R1$, the original radius, and $R2$, the smaller radius.

```
def generateRandom(R1, R2):
    '''
    - Generates random values of radius and theta.
    - keeps count of points generated
    args: R1 (large circle), R2 (smaller circle) - integer
    returns: theta and radius list, count of R1 and R2
    '''
    theta = []
    radius = []
    countR1 = 0
    countR2 = 0
    # Generating 1000 random radius and theta values.
    for i in range(1000):
        r = random.randint(1,R1)
        t = random.randint(1,360)
        # if (r not in radius) and (t not in theta):
        radius.append(r)
        theta.append(t)
```

```
        if (r <= R2):
            countR2+=1
        countR1+=1
    return radius, theta, countR1, countR2
```

Listing 7: generateRandom

Following result is obtained:

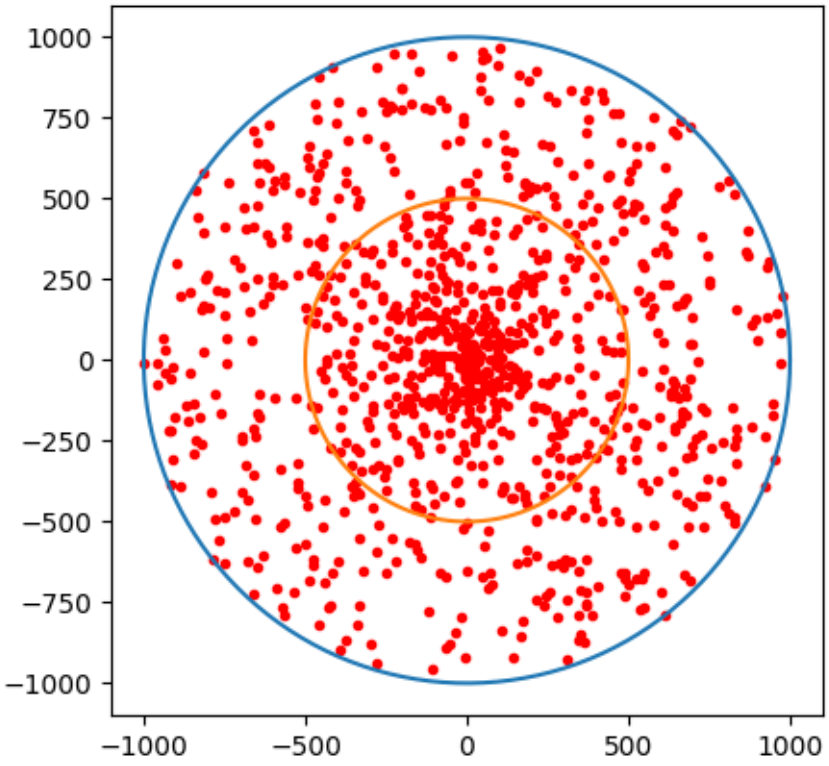

Figure 17: Random points in two circles

Variance in x values: 174494.6537683193

Variance in y values: 163347.95735094158

Number of points in R2: 505

Number of points in R1: 1000

As it can be observed that the variance in x and y values are similar to the variance found in solution 3.1. It indicates that the points are not spread properly and concentrated to one area. Moreover, the random points in R1 are not 4 times the points in R2 but as the values indicate they are 2 times greater.

Finding a better solution

Let us consider the problem we are facing in getting a uniform distribution. As we can see from the previous experiment, the number of points in the inner circle and the outer area are approximately the same. The reason we can see the points concentrated towards the centre is due to the density. Inner circle has a higher density as compared to the outer area. By density we mean,

$$density = \frac{Numberofpoints}{Area}$$

To resolve this, my first approach was to use the same intuition provided in the question but dividing the points based on the ratio of the areas of both regions. Following were the calculations:

Here, A_1 and A_2 are areas of outer and inner circles respectively.

$$A_1 = \pi \cdot R^2$$

$$A_2 = \pi \cdot \frac{R^2}{4}$$

Now, the area of the region X between the circles:

$$X = A_1 - A_2$$

$$X = 3\pi \cdot \frac{R^2}{4}$$

$$X = 3A_2$$

So, for 1000 random points, $\frac{3}{4} \cdot 1000 = 750$ lie in the region X.
The following figure was obtained:

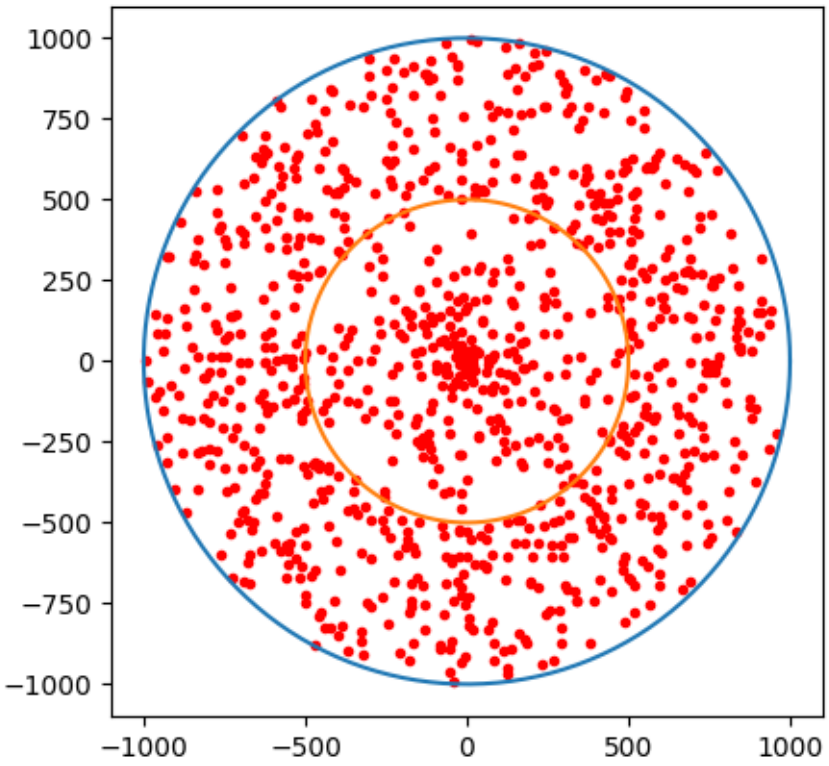

Figure 18: Image for first approach

Variance in x values: 227473.81919074082

Variance in x values: 222225.24834074604

Number of points in R1: 1000

Number of points in R2: 252

As we can see that the variance has greatly increased, although in the inner region we still have a non-uniform distribution. Now, to cater to this issue we can try another approach[1].
Now, instead of having one inner circle, we can assume increasing circles from the centre of the original circle. With each increasing area, we will increase the probability of points in those regions. By doing this, we will be keeping the density of each segment the same.

To find out the relationship between A_1 and all the other increasing regions, we can continue finding out the areas for those region. Let's rename region X as X_1.

$$X_2 = \pi \cdot \frac{R^2}{4} - \pi \cdot \frac{R^2}{16}$$

$$X_2 = 5\pi \cdot \frac{R^2}{4}$$

$$X_2 = 5A_2$$

We can continue it to as many regions. This indicates that with an increase in the radius of the smaller circle till it reaches the outer circle, the area of the regions follow an arithmetic progressing of odd numbers. The probability distribution function (PDF) would be: $k \cdot r$, where k need to be found. To find k:

$$\int_0^R k \cdot r \cdot dr = 1$$

By solving the integral, $k = \frac{2}{R^2}$.

$$pdf(r) = \frac{2 \cdot r}{R^2}$$

By integrating the pdf, we get

$$cdf(r) = \frac{r^2}{R^2}$$

Now, this CDF function is of our interest. The graph of the function is below:

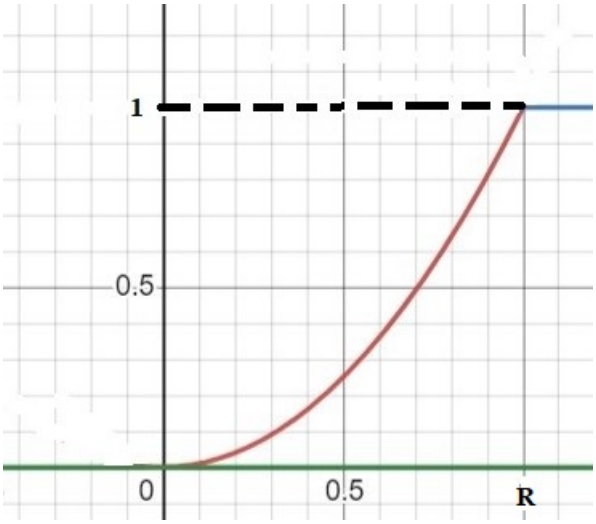

Figure 19: CDF(r)

This graph represents the range for CDF. Now, let's consider the inverse of the CDF function:

$$cdf(r) = y$$

$$y = \frac{r^2}{R^2}$$

$$\sqrt{y} = \frac{r}{R}$$

$$r = R \cdot \sqrt{cdf(r)}$$

As $cdf(r)$ can only take values from 0 to 1, our random function will have the same range, r is the different radius random values. The *generateRandom* function is modified in the following way:

```
def generateRandom(R):
    '''
    - Generates random values of radius and theta.
    - Calculates variance.
    args: Radius - integer
    returns: theta and radius list
```

```
    '''
    theta = []
    radius = []
    # Generating 1000 random radius and theta values.
    for i in range(1000):
        radius.append(R*np.sqrt(random.random()))
        theta.append(random.randint(1,360))

    return radius, theta
```

Listing 8: generateRandom

The code above produces the following output:

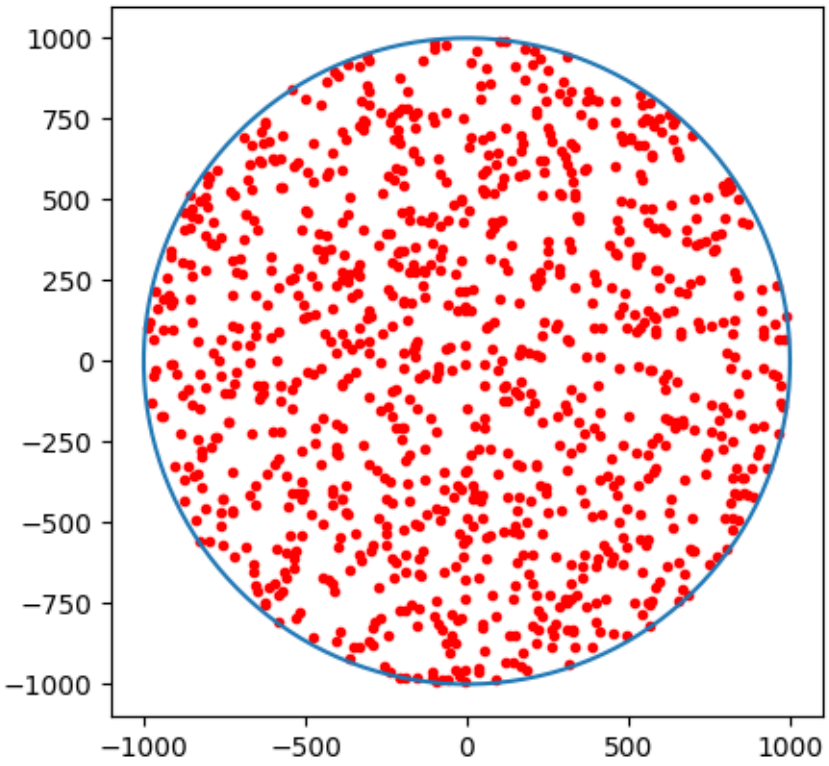

Figure 20: Uniformly Distributed Circle

Variance in x values: 251001.45048260008

Variance in y values: 262692.5017265683

These variance values also indicate that the points are uniformly spread around the circle.

Question 4

The following libraries were used in question 4:

- math
- matplotlib.pyplot
- numpy

[1] This approach is adapted from the blog, Generate Uniform Random Points within a Circle by meyavuz

For the entirety of question 4, we will assume radius r to be 10 cm.
We will construct a histogram by finding chord lenghts of 100 chords which will be our data set. The lengths of chords obtained are stored in $chordLengthList$ to be used for $x - axis$ values in the histogram while probabilities are displayed on the $y - axis$.

Solution 4.1

Selecting angles:

Two angles, $theta1$ and $theta2$ are uniformly selected using $np.random.uniform$ in the range 0 to 2π using the function $selectAngles$ given in listing 1. 100 values of $theta1$ and $theta2$ are selected and stored in $thetaListA$ and $thetaListB$ respectively. $theta2$ is taken to be greater than $theta1$ as the subtraction of the two will be used in calculation of central angle, $centralAngle = theta2 - theta1$ and thus avoid negative values. The two lists are returned.

```
# Function to uniformly select 2 angles theta1 and theta2, between 0, and 2pi.
def selectAngles():
    thetaListA = []
    thetaListB = []
    i = 0
    while len(thetaListA) < 100:
        theta1 = np.random.uniform(0, 360) # thetas are uniformly selected
        thetaListA.append(theta1)
    while len(thetaListB) < 100:
        theta2 = np.random.uniform(0, 360)
        while theta2 < thetaListA[i]: # Our function is such that theta2 will always be
    greater than theta1 to avoid negative values
            theta2 = np.random.uniform(0, 360)
        thetaListB.append(theta2)
        i += 1
    return thetaListA, thetaListB
```

Listing 9: selectAngles

Calculations:

$centralAngle$ is calculated using the values of $theta1$ and $theta2$ stored in $thetaListA$ and $thetaListB$ respectively:

$$centralAngle = theta2 - theta1$$

This $centralAngle$ is used in the calculation of chord length:

$$chordLength = 2 \times r \times \sin(\frac{centralAngle}{2})$$

The lengths of chords obtained are stored in $chordLengthList$. Each index in the list corresponds to the angles with same index in the list of angles $thetaListA$ and $thetaListB$.

```
# Function to calculate length of chord formed by theta1 and theta2
# Formula = 2*r*sin*(c/2)
# Where r is radius and centralAngle is angle between theta1 and theta2
def calculateChordLength():
    global chordLengthList # global variable so that it can be accessed outside function to
    plot histogram
    i = 0
    while len(chordLengthList) < 100:
        thetas = selectAngles() # thetas contains two lists, one containing theta1 and other
    containing theta2 respectively
        centralAngle = thetas[1][i] - thetas[0][i]
        chordLength = 2*r*(math.sin(math.radians(centralAngle/2))) # formula to calculate
    chord length is implemented here
        chordLengthList.append(chordLength) # a list of chord lengths is collected to plot
    histogram
        if i < 99:
            i += 1
    return chordLengthList
calculateChordLength()
```

Listing 10: calculateChordLength

Plotting Histogram:

Function to plot the histogram takes the *chordLengthList* achieved in the last part and thus obtains 100 values of lengths of chords. It calculates their probability using the parameter weights[2]. As each chord has a weight of 1 and the height of a single bin represents the weights of the chords in it, we can make the weight of each chord to be $1/chord$ for the height of the bin to represent probability.

```
# Function to plot histogram
def plotHistogram():
    n, bins, patches = plt.hist(chordLengthList, weights=np.ones_like(chordLengthList) / len(
    chordLengthList), color='#0A4614', alpha=0.5, edgecolor = 'black')
    plt.xlabel('Chord Length/cm', fontsize = 13)
    plt.ylabel('Probability', fontsize = 13)
    plt.title('Probability Of Chord Lengths', fontsize = 15)
    plt.grid()
    plt.show()
plotHistogram()
```

Listing 11: plotHistogram

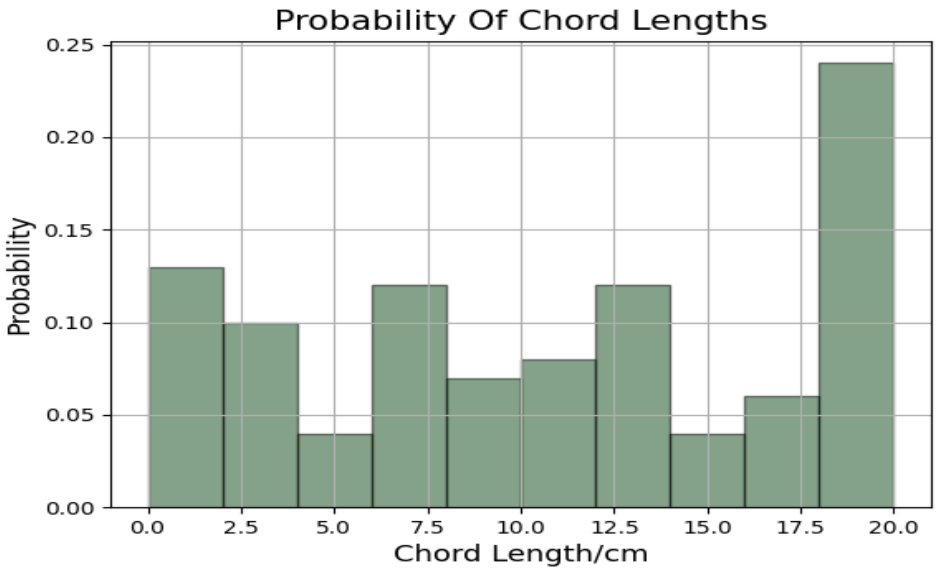

Figure 21: Simulation for first approach

Number Of experiments = 100

Number Of bins = 10

Note: The $y-axis$ limit has been automatically set as 0.25 as no probability exceeds that number, and to retain the overall aesthetic of the graph, the same applies to $x-axis$ limits.

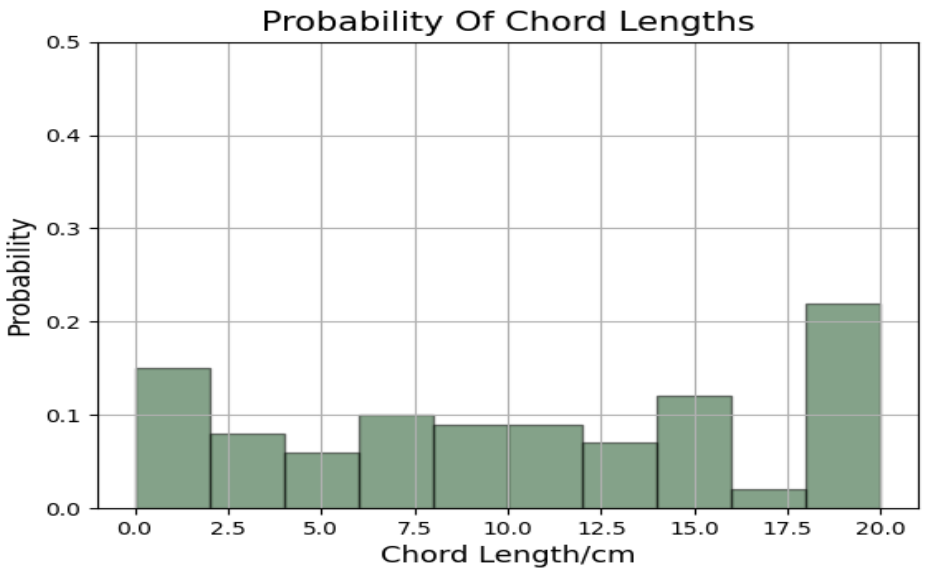

Figure 22: Simulation for first approach with $y - limit$ as 0.5

Histogram Observation:

- Probability distribution is slightly skewed towards the right
- Skewness of this histogram indicates that this method of selecting a random chord slightly results more in greater lengths of chords

Solution 4.2

Selecting angle:

The function and method of selecting a random angle is similiar to that in the previous part. The angles are stored in a list.

```
# Function to randomly select an angle theta between 0 and 2pi.
def selectAngle():
    thetaList = []
    i = 0
    while len(thetaList) < 100:
        theta = np.random.uniform(0, 360) # thetas are uniformly selected
        thetaList.append(theta)
    return thetaList
```

Listing 12: selectAngle

Find Point Of Bisection:

For each angle that has been randomly selected and stored in our list, a random bisection point $xRand, yRand$ in the range $[0, x]$ and $[0, y]$, with x and y being coordinates of the line formed by angle from center of circle to boundary. It is stored in a list of bisection points *bisectionList*. Each index in the list corresponds to the angle with same index in the list of angles *thetaList*.

```
def pointOfBisection():
    angles = selectAngle()
    bisectionList = []
    for i in angles:
        x = r*(math.sin(math.radians(i))) # x and y coordinates found of the line formed by
    angle from center of circle to boundary
        y = r*(math.sin(math.radians(i)))
        xRand = np.random.uniform(0, x) # xRand and yRand are random bisection point
    coordinates
        yRand = np.random.uniform(0, y)
        tup = (xRand, yRand) # cooridantes stored in tuple
        bisectionList.append(tup) # list of bisection points corresponding to different
    randomly selected angles
```

```
    return bisectionList
```

Listing 13: pointOfBisection

Calculations:

Distance formula between two coordinates, bisection point and origin, is used to find length of line highlighted in red in Figure 23 below:

$$\text{Length of bisection point from origin } b = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

where (x_2, y_2) are coordinates of bisection point $xRand$ and $yRand$ and (x_1, y_1) is the point of origin

Hypotenuse formula is used to find half of chord length marked as c in the figure while radius r is highlighted in blue:

$$r^2 = b^2 + c^2$$

$$c^2 = r^2 + b^2$$

$$\text{Chord Length} = 2 \times c^2$$

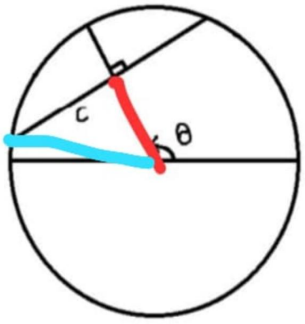

Figure 23:

```
def lengthOfChord():
    global chordLengthList
    bisection = pointOfBisection()
    for i in bisection:
        triangleSideLength = math.sqrt(i[0]**2 + i[1]**2) # length of one side of triangle:
    distance between two coordinates formula
        chordLength = 2*(math.sqrt(abs(r**2 - triangleSideLength**2))) # r^2 = a^2 + c^2 = c
    ^2 = |r^2 - b^2|, 2*sqrt(c)
        chordLengthList.append(chordLength) # chord lengths stored in list
    return chordLengthList
print(lengthOfChord())
```

Listing 14: lengthOfChord

Plotting Histogram:

The function to plot the histogram is same as that in the previous part.

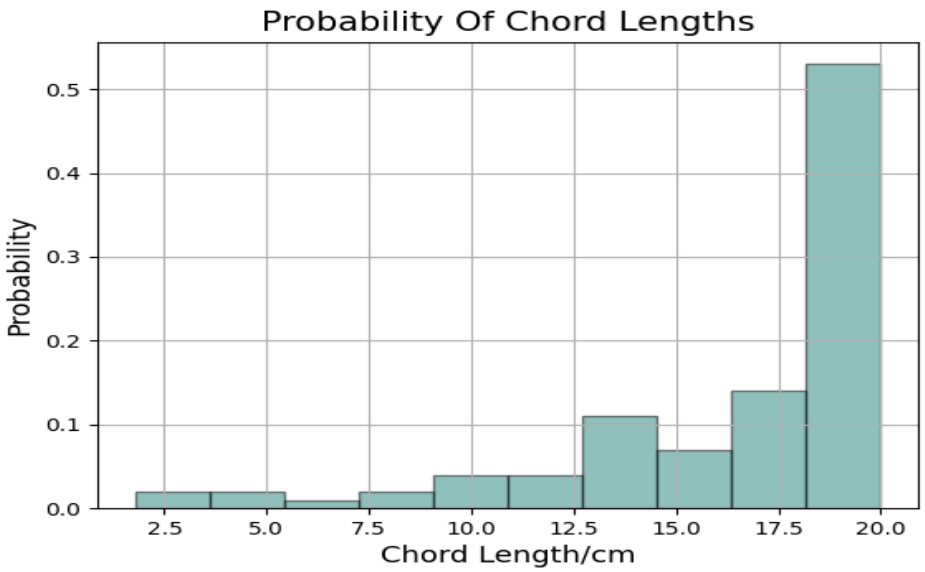

Figure 24: Simulation for second approach

Number Of experiments = 100

Number Of bins = 10

Histogram Observation:

- Probability distribution is extremely skewed towards the right
- Skewness of this histogram indicates that this method of selecting a random chord largely results more in greater lengths of chords

Solution 4.3

Selecting angles:

This function and method is the same as in part 4.2.

Find Midpoint Of Chord:

Random x and y coordinates, $xRand$ and $yRand$ are found in the range $[r, -r]$.

```
xRand = np.random.uniform(r, -r) # xRand and yRand are midpoint of random chord
yRand = np.random.uniform(r, -r)
```

Listing 15: chordMidpoint

Calculations:

Calculation for length from origin till midpoint of random chord:

$m = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ where $(x_2, y_2) = (xRand, yRand)$ and $(x_1, y1) = (0, 0)$

Hypotenuse formula is used to find half of chord length marked as c in the Figure 25 below while radius r is highlighted in blue and m is in red:

$$r^2 = m^2 + c^2$$

$$c^2 = r^2 + m^2$$

Chord Length $= 2 \times c^2$

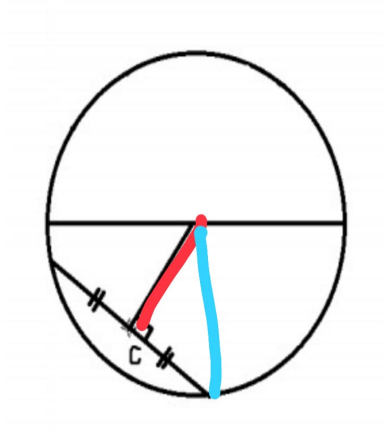

Figure 25:

Plotting Histogram:

Same method as previous part is implemented.

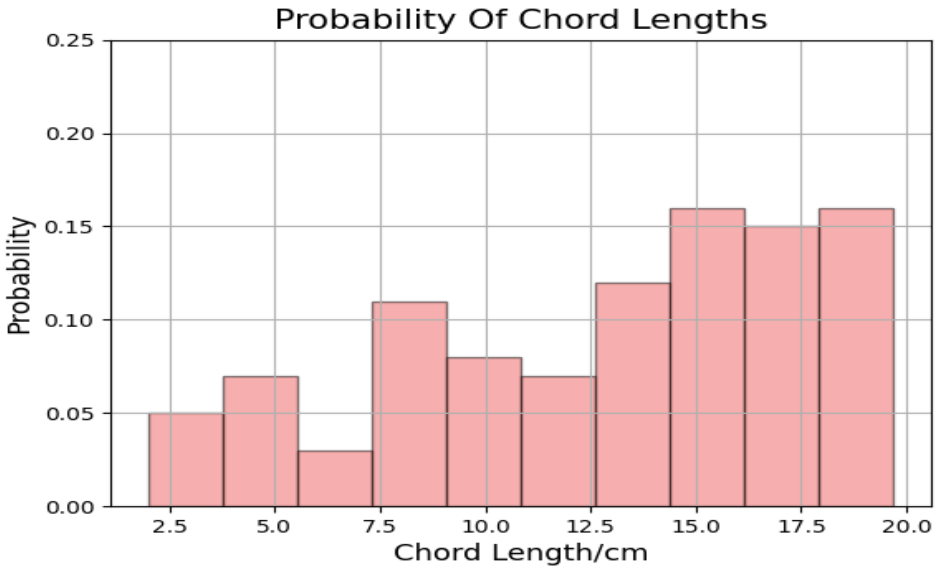

Figure 26: Simulation for third approach

Number Of experiments = 100

Number Of bins = 10

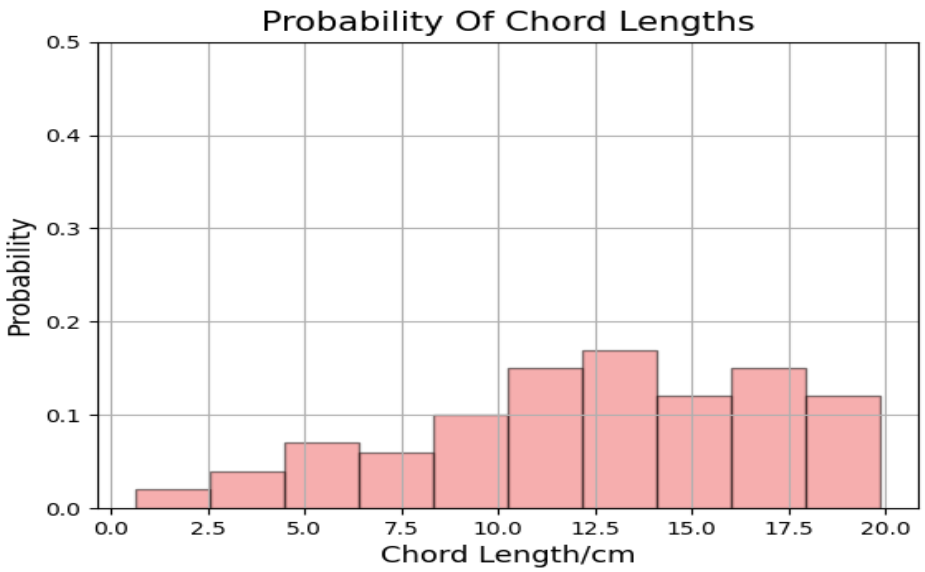

Figure 27: Simulation for third approach with $y - limit$ as 0.5

Histogram Observation:

- Probability distribution is more even than previous parts
- Skewness of this histogram indicates that this method of selecting a random chord results in all lengths of chords more equally than previous parts

Solution 4.4

Histogram 4.1: This method of selecting a random chord slightly resulted more in greater lengths of chords as seen by the probability distribution
Histogram 4.2: This method of selecting a random chord largely resulted more in greater lengths of chords as indicated by the probability distribution
Histogram 4.3: This method of selecting a random chord resulted in all lengths of chords more equally than previous parts
Thus, the most appropriate method from these 3 is the last method, 4.3, as it resulted in a more even distribution of probability and gave output of random chord lengths of more variety than other parts.

Question 5

Solution 5.1

Implemented a function heads_tails(no_of_flips): that simulates the behavior of a fair coin.
Implemented simulate_multiple(x) using above function to simulate 10 coin tosses multiple times and find the expected number of times the null hypothesis is rejected even though it is true.

Histogram of Expected Values:

Number Of Experiments = 100
Bins = 15

[2]https://stackoverflow.com/questions/38650550/cant-get-y-axis-on-matplotlib-histogram-to-display-probabilities

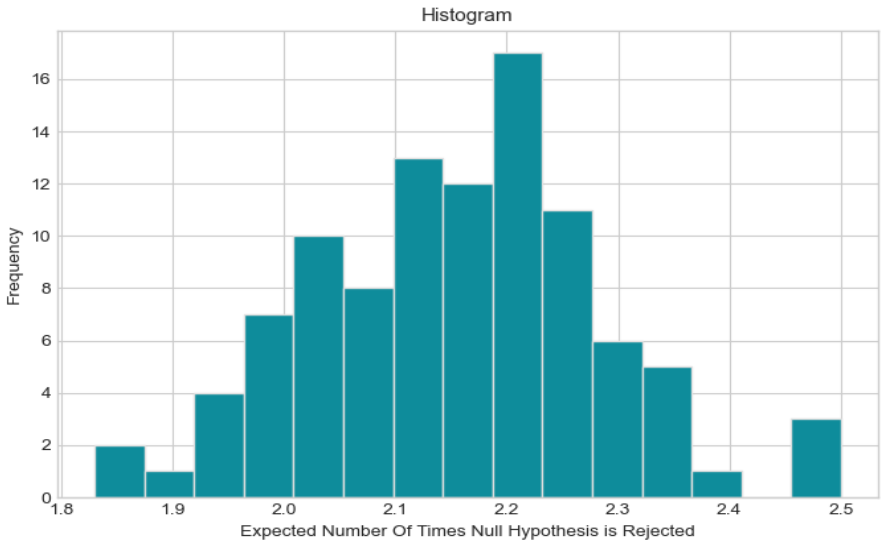

Figure 28:

What is the probability we will reject the null hypothesis even though it is true?

1. Mathematically:

Critical value associated with a $= 0.05$:

$$P(X \geq b \text{ or } X \leq 10 - b) = 0.05$$

$$P(X \geq b) = 0.025$$

$$P(X \leq b - 1) = 0.975$$

```
from scipy.stats import binom
b = 1 + binom.ppf(0.975,10,0.5)
b = 9
```

Probability of rejecting the null hypothesis is when we get heads ≤ 1 or ≥ 9 times in 10 coin tosses

$$P(X \leq 1) + P(X \geq 9)$$

$$P(X \leq 1) + (1 - P(X \leq 9))$$

From the distribution table:

$$0.0107 + (1 - 0.999)$$

$$0.0177$$

2. Simulation-wise:

```
def simulate_multiple(noOfExps):
    averagelst = []
    for _ in range(noOfExps):
        lst = []
        for _ in range(noOfExps):
            expected = 0
            for _ in range(noOfExps):
                heads_count = heads_tails(10)
                if heads_count >= 9:
                    expected += 1
                elif heads_count <= 1:
                    expected += 1
            lst.append(expected)
        averagelst.append(sum(lst)/noOfExps)
```

```
    return averagelst


noOfExps = 100
expected = simulate_multiple(noOfExps)
print("Expected probability that we will reject the null hypothesis even though it is
    true:",
      round((sum(expected)/noOfExps)/noOfExps, 2))
```

In the function simulate _ multiple() :

- Counted the number of times null hypothesis was rejected ie when heads were ≥ 9 or ≤ 1, for 10 coin tosses in 100 experiments
- Repeated this 100 times to get 100 test results
- Then found the average of these 100 test results
- Repeated all of the above steps 100 times to get 100 expected values when null hypothesis was rejected
- Probability = expected value/no of times experiment occurred = 0.022

The probabilities of the null hypothesis being rejected despite being true can be observed to be close to the threshold of 0.05.

Solution 5.2

Implemented a function that picks a sample of 30 fish lengths and returns mean and variance of that sample.
mean_variance():

- Ran a loop 30 times to get 30 lengths using fish() function
- Found mean and variance of that sample using the numpy library

.

Solution 5.2.1

SUB TASK 1:
Implemented a function that calculates the expected number of times the null hypothesis is rejected by comparing p value to threshold of 0.05. The sample is of 30 fishlengths where mean and variance of that sample is calculated by the function implemented in **5.2**

No of experiments = 100
Sample size(n) = 30
Population Mean = 23
Bins = 15

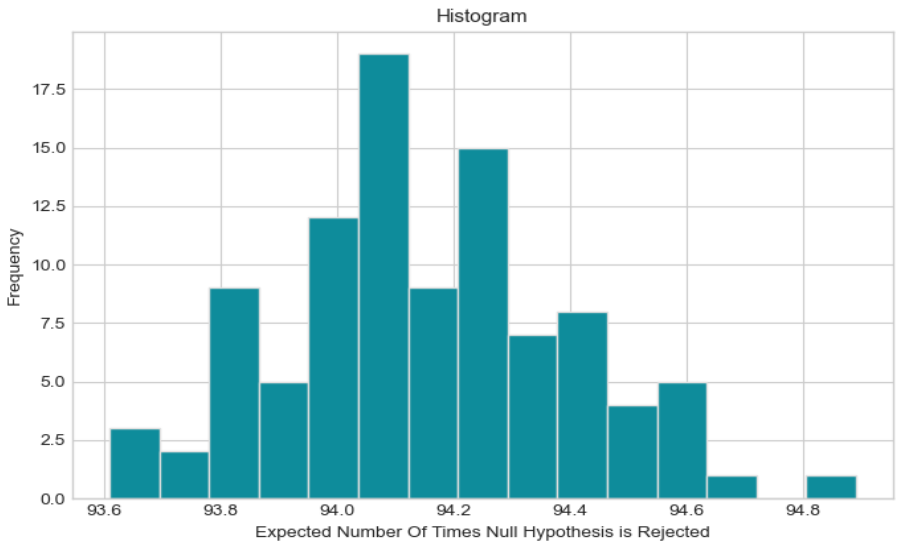

Figure 29:

SUB TASK 2:
Implemented a function that takes in u_o(population mean) and n(sample size), and conducts a single hypothesis test and returns probability $P(|S - u_o| >= a)$ where a = $|u - u_0|$

$$P(|S - u_o| \geq a)$$
$$P(S \leq a - u_o) + P(S \geq a + u_o)$$
$$P(S \leq a - u_o) + (1 - P(S \leq a + u_o))$$

```
#P( S >= u0 + a)
x = 1 - norm(loc= popmean, scale= math.sqrt(variance) /
                  math.sqrt(n)).cdf(popmean + a)
```

```
#P( S <= u0 - a)
y =  norm(loc=popmean, scale=(math.sqrt(variance)) /
                 math.sqrt(n)).cdf(popmean - a)
```

- The function returns the added probabilities $(x + y)$

SUB TASK 3:
Modified the above function to performs several experiments, each with several hypothesis tests and plotted a histogram with:

Hypothesis tests = 100
Experiments = 100

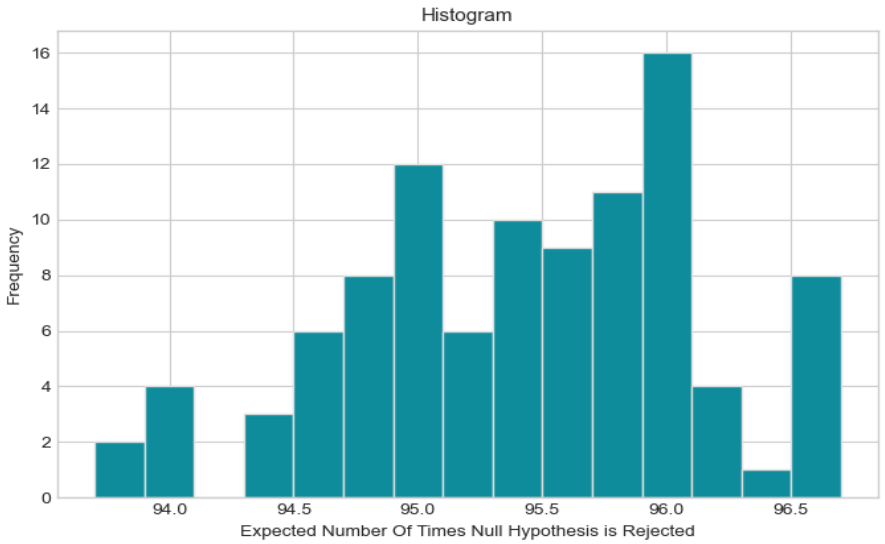

Figure 30:

- It can observed from the figure that about 90% of the time null hypothesis is rejected
- Rejecting a null hypothesis is always accompanied with a chance that the null hypothesis was actually true. And with smaller sample size there is a greater margin of error,due to sample mean being less close to the true value, so it will not yield valid results
- It would not have been sufficient to accept or reject the null hypothesis with a single experiment because we are only looking at a small sample of 30 which is inadequate for the threshold we have chosen, our study will have reduced statistical power

Solution 5.2.2

Conducted several hypothesis tests with:

$u_o = 23$
n = 70
Number of experiments = 100

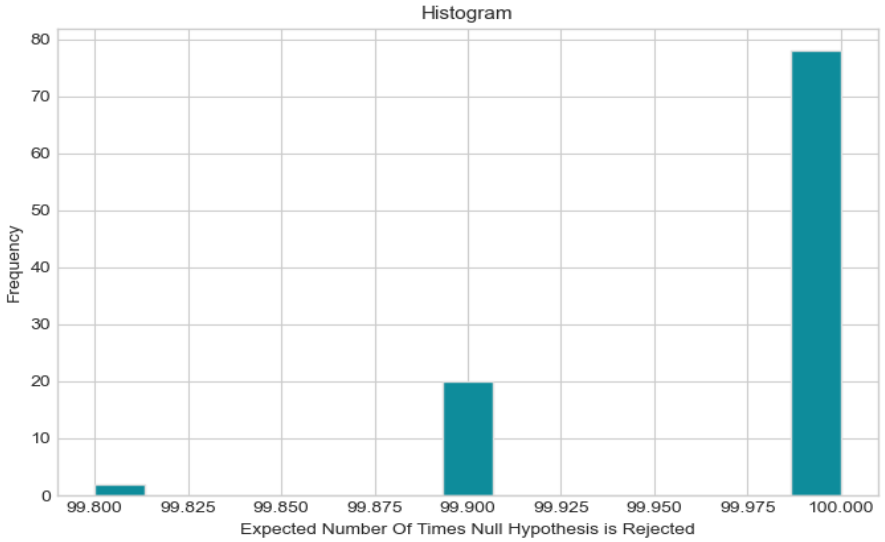

Figure 31:

- It can observed that about 100% of the time null hypothesis is rejected
- Increasing sample size makes the hypothesis test more sensitive - more likely to reject the null hypothesis when it is, in fact, false. And so it would have been sufficient to reject the null hypothesis with a single experiment because that single experiment gives reliable results due to mean being close to true value
- Greater the sample size, greater the power of the test
- When simulating various times, we get the similar results, which proves how conducting experiment once can be enough

Solution 5.2.3

- Implemented own fish() function that would give sample lengths with mean = 23 and standard deviation = 3
- Implemented a function Hypothesis_Testing(n) that checks if probability of null hypothesis being rejected despite being true in sample is <= 0.1
- The probability was tested 100 times in 100 experiments
- Plotted proportions in histogram for different values of n that would ensure null hypothesis is not wrongly rejected more than 10 percent of the time.
- With each value of n the centre of normal distribution was observed, whether it was around 0.1
- And so it was concluded from the plotted histogram that the least value of n is 46.

Number Of Experiments = 100
Number of Hypothesis tests = 100
Bins = 30
n = 46

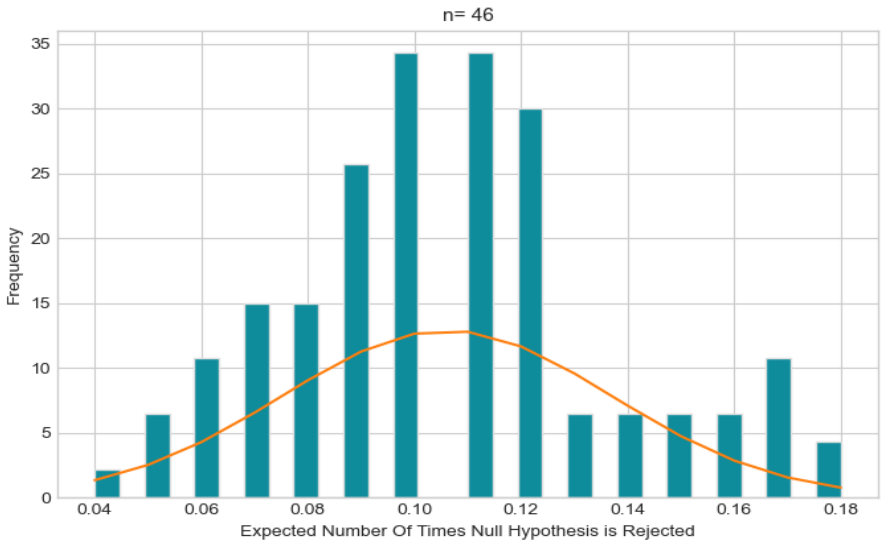
n= 46
Frequency
0
5
10
15
20
25
30
35
0.04
0.06
0.08
0.10
0.12
0.14
0.16
0.18
Expected Number Of Times Null Hypothesis is Rejected

Figure 32: