Probability & Statistics Project

Hana Ali Rashid, hr05940
Tasmiya Malik, tm06183
Ifrah Ilyas, ii06178

April 27, 2021

Q1: Random Walk

1.1

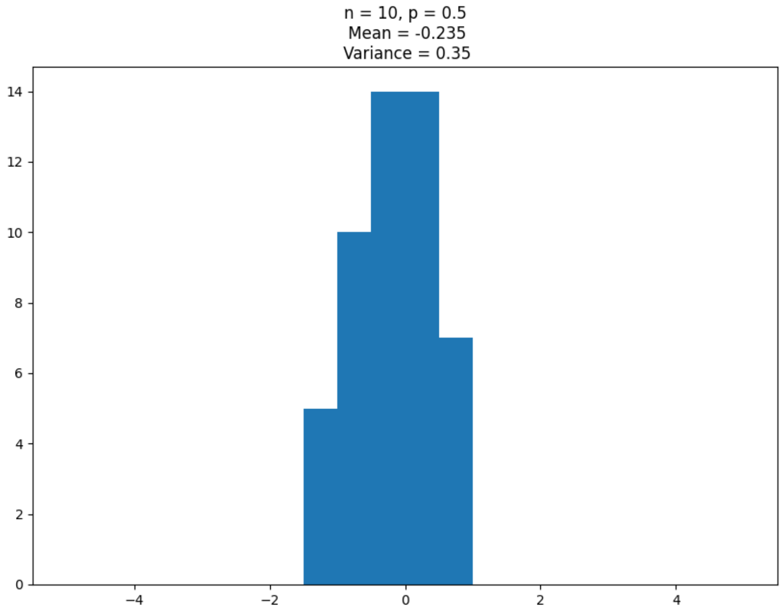
Function implementation in Python:

```python
def get_updated_position(n,p):
    pos = 0 #position
    for _ in range(n):
        rand = random.randint(1,100) #generating a random number in the range 1 to 100
        if rand < p*100:
            pos += 1 #move one step right
        else:
            pos -= 1 #move one step left
    return pos #return final position
```

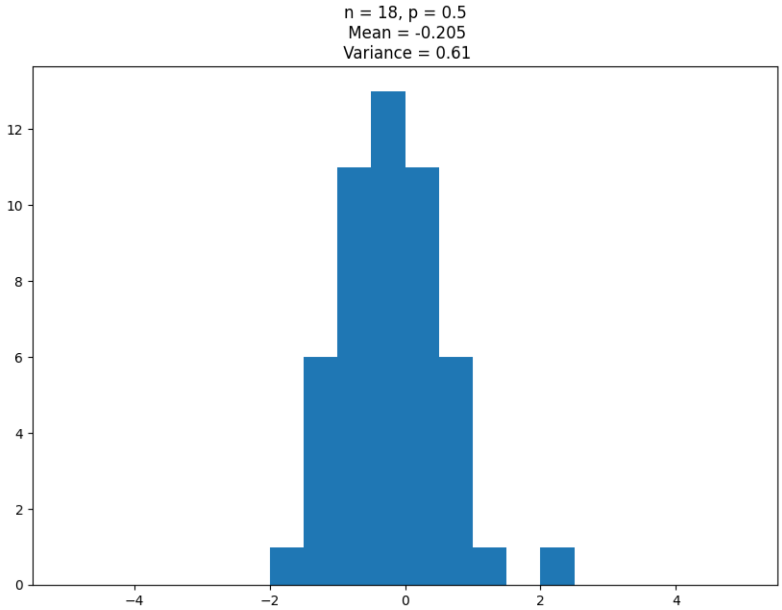
Calling the function for several iterations to get multiple expected values:

```python
def main_11():
    '''Calculating and plotting expected outcomes for various combinations of n and p'''
    p = 0.7
    n = 10
    expected = []
    for j in range(50): #expected values for each (n,p) for 50 iterations
        outcomes = []
        for i in range(25):
            outcomes.append(get_updated_position(n,p))
        expected.append(sum(outcomes)/25) #appending the expected (average) value for each(n,p)
    #plotting and showing a histogram of calculated expected values
    fig, ax = plt.subplots(figsize =(10, 7))
    ax.hist(expected, bins = range(-5,10))
    plt.title('n = '+str(n)+', p = '+str(p)+'\n Mean = '+str(round(statistics.mean(expected),3))+'\n Variance = '+str(round(statistics.variance(expected),3)))
    plt.savefig("Q1_histograms/q1"+'_n = '+str(n)+'_ p = '+str(p)+'.png')
    plt.show()

# 1.2 -----------------------------------------------------------------------------------------------
```

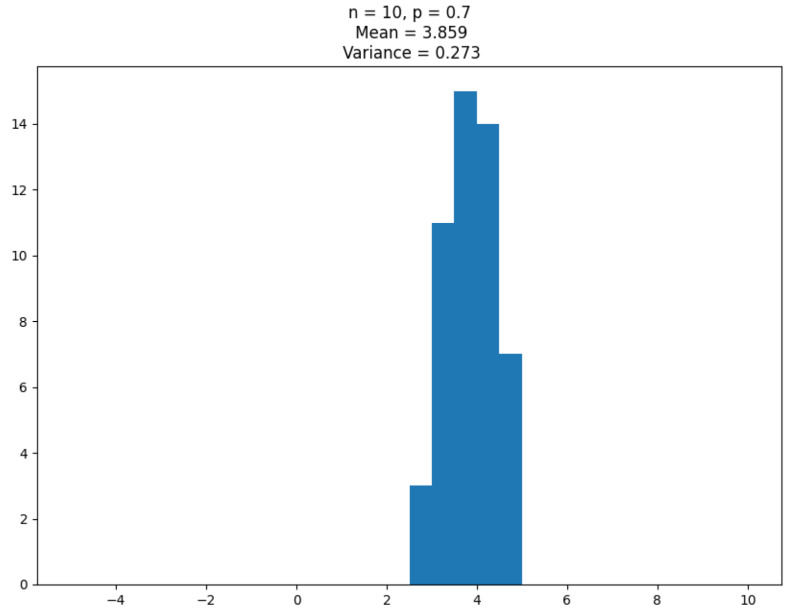
Histograms produced by the above code for various combinations of n and p:



The above histogram appears to follow a normal distribution with a mean of 3.267 and variance of 0.233.



The above histogram appears to follow a normal distribution with a mean of 3.267 and variance of 0.233.



The above histogram appears to follow a normal distribution with a mean of 3.267 and variance of 0.233.

1.2

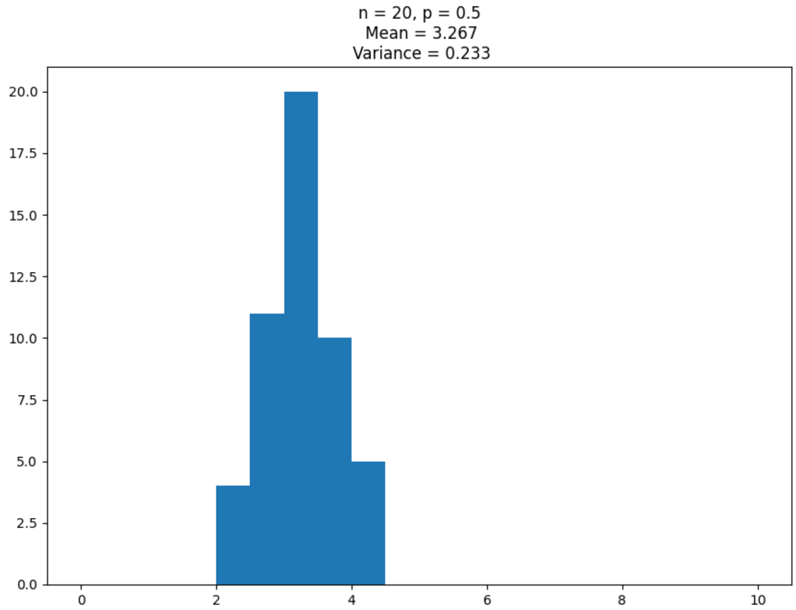
Function implementation in Python:

```
    for _ in range(n):
        rand = random.randint(1,100) #generating a random number in the range 1 to 100
        if rand < p*100 or pos <= 0: #move one step right if pos == 0
            pos += 1
        else:
            pos -= 1 #move one step left
    return pos #return final position
# ---------------------------------------------------------------------------------------------------
def main_12():
```

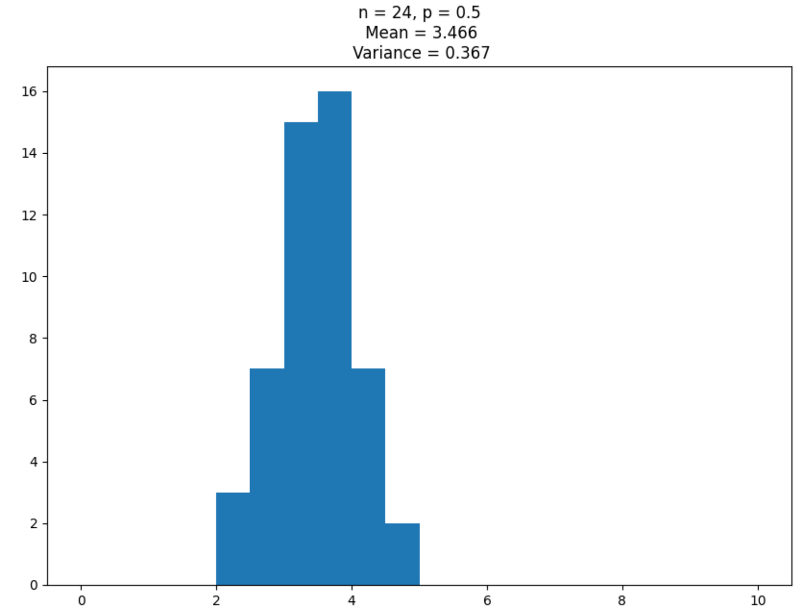
Calling the function for several iterations to get multiple expected values:

```
    p = 0.7
    n = 24
    expected = []
    for j in range(50): #expected value for each (n,p) for 25 iterations
        outcomes = []
        for i in range(25):
            outcomes.append(get_updated_position_restricted(n,p))
        expected.append(sum(outcomes)/25) #appending the expected (average) value for each(n,p)
    #plotting and showing a histogram of calculated expected values
    fig, ax = plt.subplots(figsize =(10, 7))
    ax.hist(expected, bins = [4,4.5,5,5.5,6,6.5,7,7.5,8,8.5,9,9.5,10,10.5,11,11.5,12,12.5,13,13.5,14])
    plt.title('n = '+str(n)+', p = '+str(p)+'\n Mean = '+str(round(statistics.mean(expected),3))+'\n Variance = '+str(round(statistics.variance(expected),3)))
    plt.savefig("Q1_histograms/Q1.22 "+'_n = '+str(n)+'_p = '+str(p)+'.png')
    plt.show()
# 1.3 ---------------------------------------------------------------------------------------------
def stepsToMeet(pos1,pos2,p1,p2):
    count = 0 #keeps count of number of steps taken for objects to meet
    while pos1 != pos2:
```

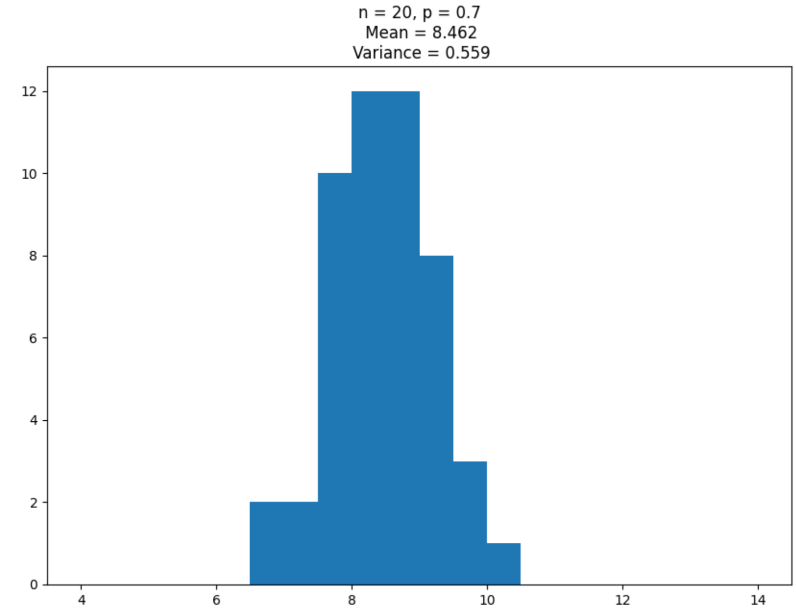
Histograms produced by the above code for various combinations of n and p:



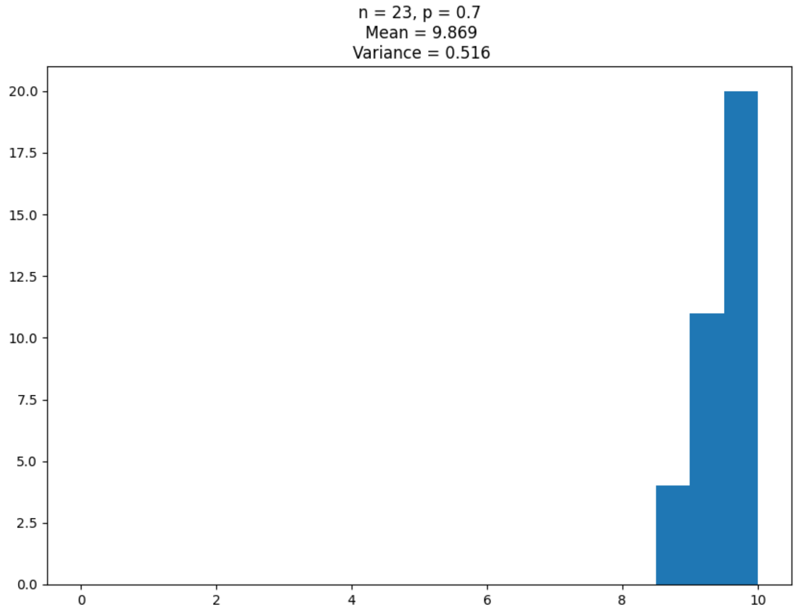
The above histogram appears to follow a normal distribution with a mean of 3.267 and variance of 0.233.



The above histogram shows that...



The above histogram shows that...



1.3

Function implementation in Python:

```
        if rand < p1*100:
            pos1 += 1 #move one step right
        else:
            pos1 -= 1 #move one step left
        rand = random.randint(1,100) #generating a random number in the range 1 to 100
        if rand < p2*100:
            pos2 += 1 #move one step right
        else:
            pos2 -= 1 #move one step left
        count += 1
    return count

def main_13():
    '''Calculating and plotting expected outcomes for various combinations of n & p'''
    expected = []
```

Calling the function for several iterations to get multiple expected values:

```
    p2 = 0.3
    pos1 = -4
    pos2 = 4
    for i in range(25):  #calculating the expected value for each (n,p) for 25 iterations
        outcomes = []
        for j in range(25):
            outcomes.append(stepsToMeet(pos1,pos2,p1,p2))
        #calculating the average expected value for each(n,p)
        expected.append(sum(outcomes)/25) #appending the expected (average) value for each(n,p)
    #plotting a histogram of calculated expected values
    fig, ax = plt.subplots(figsize =(10, 7))
    ax.set_xlabel('Expected no. of steps taken to meet')
    ax.set_ylabel('Frequency')
    ax.hist(expected, bins = range(5,18))
    plt.title('p1 = '+str(p1)+', p2 = '+str(p2)+'\npos1 = '+str(pos1)+', pos2 = '+str(pos2)+'\n Mean = '+str(round(statistics.mean(expected),3))+', Variance = '+str(round(statistics.variance(expected),3)))
    plt.savefig("Q1_histograms/Q1.3 "+'_p1 = '+str(p1)+'_p2 = '+str(p2)+'_pos1 = '+str(pos1)+'_pos2 = '+str(pos2)+'(6).png')
    plt.show()
# ----------------------------------------------------------------------------------------------------------------
main_13()
```

Q2: Simulating Distributions

2.1

It does accomplish as it plots for a value of x (y is the variable expressed in terms of x) against the pdf e^{-y} such that each bar in the histogram shows the probability for an average value of x. The distrubution plotted is a continuous exponential random variable. The number of bins used are significant for our simulation which will become evident with the following images:

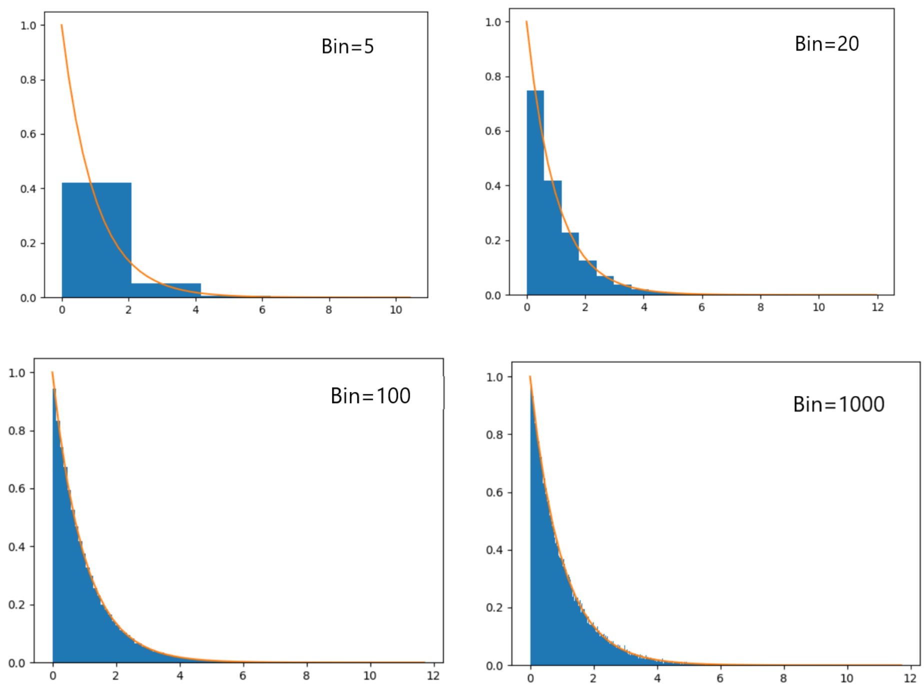


Figure 1: Simulations done with different number of bins

It is noticeable that, with more number of bins, the accuracy of the results increases. However, it is also interesting to note that using 1000 bins almost gives the same result as 100 bins. Therefore, while simulating we need to take an appropriate values for bins for better approximation and less running time.

2.2

In the code we see that :

$$y = \frac{1}{1-x}$$

With the trick shown, we can back track to find the original distribution.

$$1 - \frac{1}{y} = x$$

We know that the upper bound will be y. For the lower bound we see that:

$$\frac{1}{1} - \frac{1}{y} = x$$

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

$$\therefore -\frac{1}{y}\bigg|_1^y = x$$

Performing derivative on the left hand side:

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

$$\therefore \int_1^y \frac{1}{y^2} dy = x$$

We can affirm that we have reached the right deduction by looking at line 12 of the code. So, the distribution of y is :

$$f_Y(y) = \frac{1}{y^2} for y \geq 1$$

```
1 # y . sort ()
2 # ind = ( np . array ( y ) > 30) . tolist () . index (1)
3 # y = y [: ind ]
```

Lines 5-7 is simply trying to implement a cutoff for values of y greater 30 as values starting to get more smaller. Given that we taken small amount of bins, the average value of each bar becomes negligible and is not visible in the graph. Therefore we cut it down to 30 so that the approximation is more accurate.

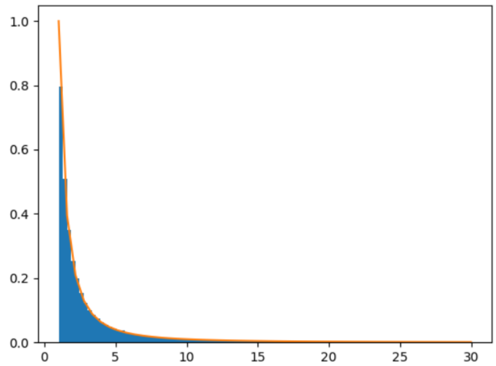


Figure 2: Before Removing

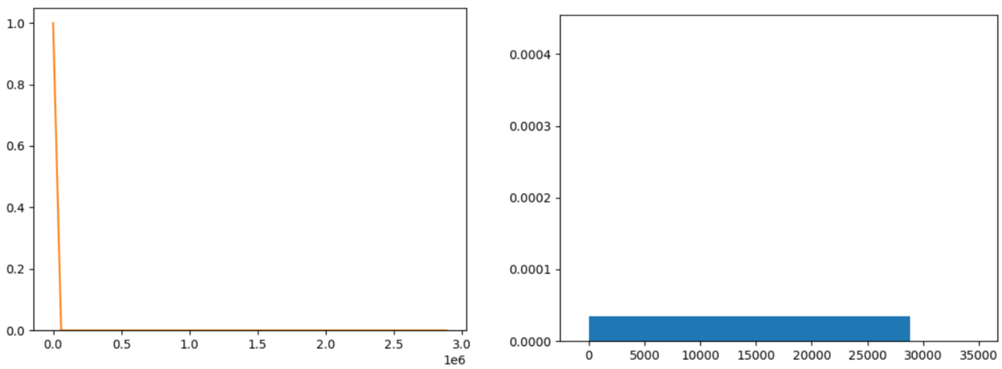


Figure 3: After Removing

2.3

For,

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

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

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

$$\left. \frac{-1}{2y^2} \right|_{\sqrt{\frac{1}{2}}}^y = x$$

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

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

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

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

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

Code:

```
y = []
for i in range (100000) :
    x = np . random . random ()
    y . append ((1 / (2*(1 - x )))**(1/2) )

y . sort ()
ind = ( np . array ( y ) > 10) . tolist () . index (1)
y = y [: ind ]
bins = 100
binWidth = (max( y ) - min( y ) ) / bins
plt . hist (y , bins = bins , weights = np . ones (len ( y ) ) /( len ( y ) * binWidth ) )
values = np . linspace ( min( y ) , max( y ) , 50)
plt . plot ( values , (1 / values ** 3) )
plt . show ()
```

The output of simulation for this y:

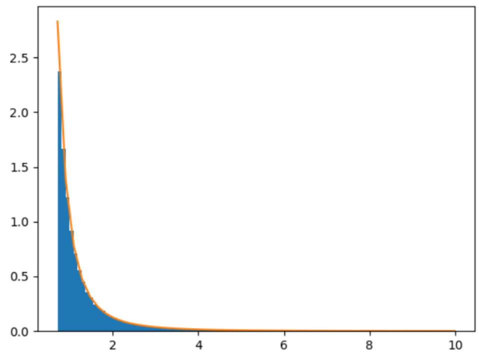


Figure 4: Simulation of distribution $f_Y(y) = \frac{1}{y^3} for y \geq \sqrt{\frac{1}{2}}$

For the simulation of expected values, the same code used for the above function ran for 10000 times and each run would take the mean of y :

```
y = []
for j in range (10000) :
    x = np . random . random (10000)
    x=(1 / (2*(1 - x )))**(1/2)
    y.append(np.mean(x))
print(y)
bins = 100
binWidth = (max( y ) - min( y ) ) / bins
plt . hist (y , bins = bins , weights = np . ones (len ( y ) ) /( len ( y ) * binWidth ) )
values = np . linspace ( min( y ) , max( y ) , 50)
plt . plot ( values , (1 / values ** 3) )
plt . show ()
```

The output of this simulation:

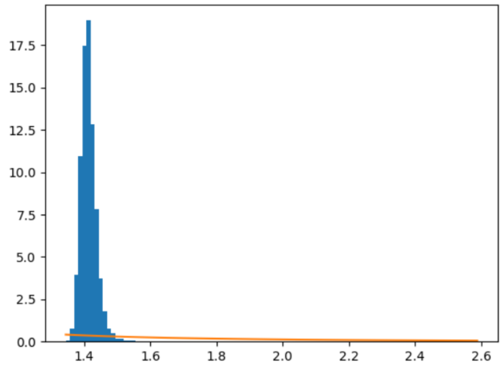


Figure 5: Simulation of expected values of $fY(y) = \frac{1}{y^3} for y \geq \sqrt{\frac{1}{2}}$

Q3. Picking a Random Point Correctly

3.1

For part 3.1, we followed the method specified in the question. The program takes radius as an input from the user, and passes it to *gen_points(R)* to display the graph. It first finds random radius R and angle θ, then converts it into cartesian cordinates. It stores the values of x and y till the iterations are running, then plot the points. To make sure the points lie in the given radius, we also draw the circle.

- *polar_to_cart(r,t)*: This function takes the polar cordinates (r and θ) and converts them to their cartesian equivalent. It is called from *random_point(R)*, to return the cartesian cordinates. It uses the following formulae,

$$x = r\cos(\theta)$$
$$y = r\sin(\theta)$$

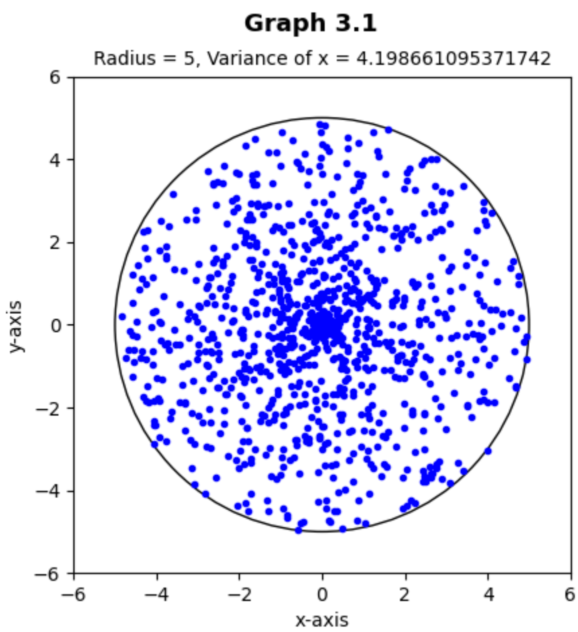
- *random_point(R)*: The function is responsible for generating a random point in polar cordinates, and returns the converted cartesian cordinates.

- *gen_points(R)*: This is the main function which runs 1000 iterations, and calls *random_point(R)*, and stores the result in the separate lists for X and Y cordinates. Then the lists are passed to the plot functions, to graph the points.

The resulting graph has very dense distribution towards the center.

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

def random_point(R):

    #R=R, theta = angle
    r = np.random.uniform(0, R)     #random R in range 0-R
    theta = np.random.uniform(0, 360)*(m.pi/180)    #random theta between 0-360 degrees
    return(polar_to_cart(r,theta))

def polar_to_cart(r,t):

    x = r * np.cos(t)   #x = rcos(theta)
    y = r * np.sin(t)   #y = rsin(theta)
    return((x,y))

def gen_points(R):

    I = 1000      #iterations
    X = []    #list for x axis
    Y = []    #list for y axis
```

```python
    #generating the lists for X and Y
    for _ in range(I):
        point = random_point(R)
        X.append(point[0])
        Y.append(point[1])

    """Drawing the circle and plotting the points"""

    #Drawing the circle
    fig, ax = plt.subplots()
    ax.axis([-R-1,R+1,-R-1,R+1])
    circle = plt.Circle((0,0),R,Fill=False)
    ax.set_aspect(1)
    ax.add_artist(circle)
    plt.title("circle")

    #Plotting X and Y
    plt.plot(X,Y,'.',color="blue")
    fig.suptitle("Graph 3.1",fontsize=13,fontweight='bold')
    plt.xlabel("x-axis")
    plt.ylabel("y-axis")
    plt.title("Radius = {}, Variance of x = {}".format(R,np.var(X)),fontsize=10)
    plt.savefig("Q3/Q3(1).png")
    plt.show()
```



3.2

In part 3.2, we calculated the cartesian cordinates directly from the specified radius R. If the random point lied farther from the distance from the origin, it will be discarded and the program will find a new point.

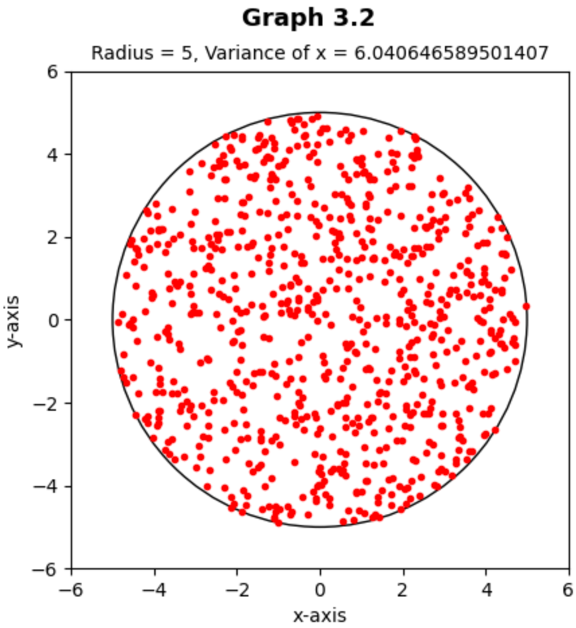
- *random_point(R)*: This function finds random points for x and y cordinates between the range $-R$ and R. It returns the tuple with cartesian cordinates.

- *dist_from_origin(x,y)*: The function find and return the distance of the point from the origin.

$$dist = \sqrt{x^2 + y^2}$$

- *gen_points(R)*: The main function iterates 1000 times and calls *random_point(R)*. Then it checks the condition that if the distance of the point (x, y) is within the radius R, it would append the points to their respective lists, else the counter will be decremented by 1 and the point will not be added to the list. Then the lists X and Y are passed to the plot functions for display.

```python
import matplotlib.pyplot as plt
import numpy as np
import math as m

def random_point(R):

    x = np.random.uniform(-R,R) #random point x
    y = np.random.uniform(-R,R) #random point y
    return((x,y))

def dist_from_origin(x,y):
    return(m.sqrt(x**2+y**2))

def gen_points(R):

    X = []        #list for x-axis
    Y = []        #list for y-axis
    I = 1000      #iterations

    for a in range(I):
        point = random_point(R)
        if dist_from_origin(point[0],point[1]) <= R:  #if distance is smaller than or equal to R
            X.append(point[0])
            Y.append(point[1])
        else:                  #if distance is greater than R
            a = a-1

    """Drawing the circle and plotting the points"""

    #Drawing the circle
    fig, ax = plt.subplots()
    ax.axis([-R-1,R+1,-R-1,R+1])
    circle = plt.Circle((0,0),R,Fill=False)
    ax.set_aspect(1)
    ax.add_artist(circle)
    plt.title("circle")

    #Plotting the points
    plt.plot(X,Y,'.',color="red")
    fig.suptitle("Graph 3.2",fontsize=13,fontweight='bold')
    plt.xlabel("x-axis")
    plt.ylabel("y-axis")
    plt.title("Radius = {}, Variance of x = {}".format(R,np.var(X)),fontsize=10)
    plt.savefig("Q3/Q3(2).png")
    plt.show()
```



3.3

In part 3.3, we had to modify the function for polar cordinates, such that the graph it made was uniformly distributed, similar to the histogram of part 3.2.

The reason behind the uneven distribution when we used polar cordinates was because of the way polar and cartesian planes interact. If we plot points in a smaller radius with varying angles, the points will be closer, as compare to a larger radius. For larger radius the points will be far apart. So when we were converting them into cartesian cordinates, many of them lied near the origin. To solve this, we simply take the CDF, i.e. the area of the circle, and make radius r its subject. Mathematically,

$$P(r \leq R) = k.\pi.r^2$$

To find k, we use P(r=R), which is equals to 1.

$$k.\pi.R^2 = 1$$
$$k = \frac{1}{R^2\pi}$$

Now that we know the value of k, we can find the whole CDF by plugging its value.

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

Now we make radius r, the subject of the equation.

$$r = \sqrt{R^2.CDF} = R\sqrt{CDF}$$

We can find our random radius r using this formula. This will produce the points in proportionality with the area of the circle, thus keeping the point uniform.

From here, we will describe the functionality of the coded program.

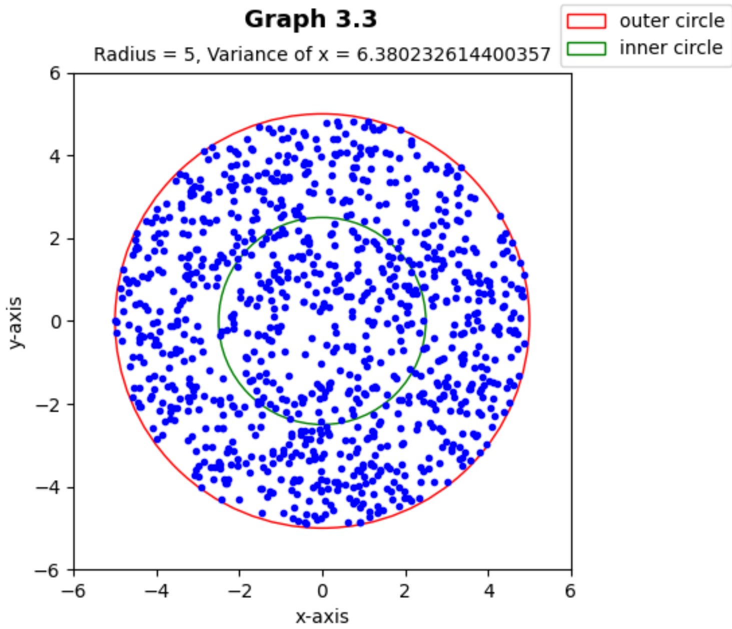
- *random_point(R)*: The function finds random radius R and theta θ. It uses the function for R derived above.

- *polar_to_cart(cord)*: This is the same function that we used in part 3.1. The function takes radius r and angle t as the input, and returns the cartesian cordinates. We use the following formulae to derive the cordinates,

$$x = r\cos(t)$$
$$y = r\sin(t)$$

- *gen_points(R)*: Like previous parts, this function makes 1000 iterations, and call *random_point(R)* for each of them, then stores the returned values in their respective lists X and Y.
  Then it draws outer and inner circles of the given radius R and $\frac{R}{2}$ respectively, and then plot the points stored in the lists X and Y.

```python
import numpy as np
import matplotlib.pyplot as plt
import math as m

def random_point(R):

    #r=radius, theta=angle
    r = R*m.sqrt(np.random.uniform(0,1))      #random r using R(CDF)**1/2
    theta = np.random.uniform(0, 360)*(m.pi/180)    #random theta between 0-360 degrees
    return(polar_to_cart(r,theta))

def polar_to_cart(r,t):

    x = r * np.cos(t)  #x = rcos(theta)
    y = r * np.sin(t)  #y = rsin(theta)
    return((x,y))

def gen_points(R):

    I = 1000      #iterations
    X = []     #list for x axis
    Y = []     #list for y axis

    #generating the lists for X and Y
    for _ in range(I):
        point = random_point(R)
        X.append(point[0])
        Y.append(point[1])

    """Drawing the circles and plotting the points"""

    #Outer Circle of radius R
    fig, axes = plt.subplots()
    axes.axis([-R-1,R+1,-R-1,R+1])
    circle1 = plt.Circle((0,0),R,Fill=False,color="red")
    axes.set_aspect(1)
    axes.add_artist(circle1)

    #inner circle of radius R/2
    circle2 = plt.Circle((0,0),R/2,Fill=False,color="green")
    axes.set_aspect(1)
    axes.add_artist(circle2)
    fig.legend([circle1,circle2],["outer circle", "inner circle"])

    #Plotting X and Y
    plt.plot(X,Y,'.',color="blue")
    fig.suptitle("Graph 3.3", fontsize=13,fontweight='bold')
    plt.xlabel("x-axis")
    plt.ylabel("y-axis")
    plt.title("Radius = {}, Variance of x = {}".format(R, np.var(X)),fontsize=10)
    plt.savefig("Q3/Q3(3).png")
    plt.show()
```



Q4. Saying Random is not enough

4.1

For 4.1, we took the approach specified in the question. We first pick two random angles θ_1 and θ_2. Then find their difference to find the angle lying in between θ_1 and θ_2. Then we use this new angle in *cord length* formula. The program does this process 1000 times and stores the values in the specified list, then graphs a histogram for the length of the cords and the probability of them occuring. Following describes each function and their purpose/method for the program.

1. *random_theta()*: The function finds random θ_1 and θ_2. θ is chosen randomly from degrees(ϕ) and then converted to radians.

$$\theta_1 = \phi_1 . \frac{\pi}{180}$$
$$\theta_2 = \phi_2 . \frac{\pi}{180}$$

2. *cord(R)*: The function takes the radius as the input and returns length of the cord between θ_1 and θ_2. It first calls *random_theta()*, and find the absolute value of their difference ($\theta = |\theta_1 - \theta_2|$). Then the length of the cord in calculated using the following formula for cord length,

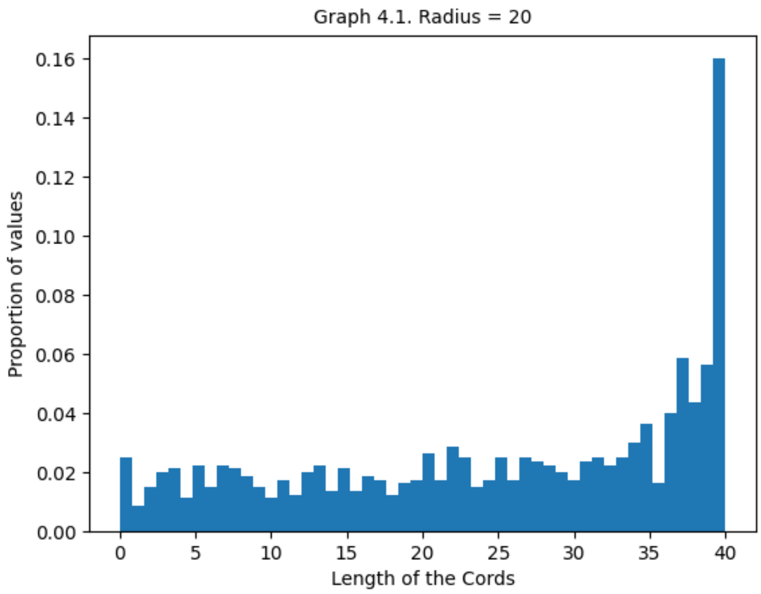
$$\text{cord length} = 2R. \sin(\theta/2)$$

3. *find_cords1(R)*: The function takes radius R as input, then draw the histogram as the output. It iterates 1000 times, calls *cord(R)* for each iteration and append the result in *cord_len*. The bin value is set to be 50 as it provides a good threshold for the 1000 values. Then it plots the graph using matplotlib.

```

# 4.1
    --------------------------------------------------------------------------------
def random_theta():
    theta1 = np.random.uniform(0,360)*(m.pi/180)     #random theta 1
    theta2 = np.random.uniform(0,360)*(m.pi/180)     #random theta 2
    return (theta1,theta2)

def cord(R):
    angle = random_theta()
    theta = abs(angle[0] - angle[1])     #theta between the two radii(theta1 and theta2)
    l = 2*R*m.sin(theta/2)  #length of cord = 2rsin(theta/2)
    return(l)
```

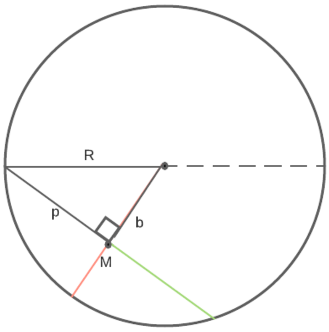
```
def find_cords1(R):

    cord_len = []
    I = 1000     #iterations

    for _ in range(I):
        cord_len.append(cord(R))

    #Finding bins
    bin_val = 50

    plt.hist(cord_len, bins = bin_val,density=True)
    plt.title("Graph 4.1. Radius = {}".format(R),fontsize=10)
    plt.ylabel("Proportion of values")
    plt.xlabel("Length of the Cords")
    plt.savefig("Q4/Q4(1).png")
    plt.show()
```

Figure 6: Histogram of 4.1



4.2

For 4.2, we follow the instructions given in the question. We first pick a direction/angle and assume an imaginary radius there. Then we pick a random point on that radius and find its distance from the origin $(0, 0)$. This serves as our base, while to radius R serves as our hypotaneous. Then we find the perpendicular using the pythagorean theorem, and returns the twice of it. As the perpendicular shows the half part of that cord, with randomly chosen point as its center. This point is also the midpoint for our circle, so we double the perpendicular to find the whole cord length.



1. *random_ cord(R)*: This function does multiple tasks. It first find a random direction/angle in the circle, assumes a radius on that angle, and pick a random point on the assumed radius. This point serves as the Midpoint M as well. Then it converts the polar cordinates into cartesian using the following formulae,

$$x = rcos(\theta)$$
$$y = rsin(\theta)$$

   Then we find our base b and perpendicular p, while assuming it is a right-angle triangle because p and b makes a right-angle triangle. In our case we assume the distance between the origin and the random point (x, y) to be our base, and radius R as our hypotaneous. We use the *distance formula* to find the value of base,

$$base = \sqrt{x^2 + y^2}$$

   Then we find the value of perpendicular line by using the pythagorean theroem,

$$perpendicular = \sqrt{R^2 - base^2}$$

   Then we simply multiply the value of perpendicular to find the length of the cord and return it.

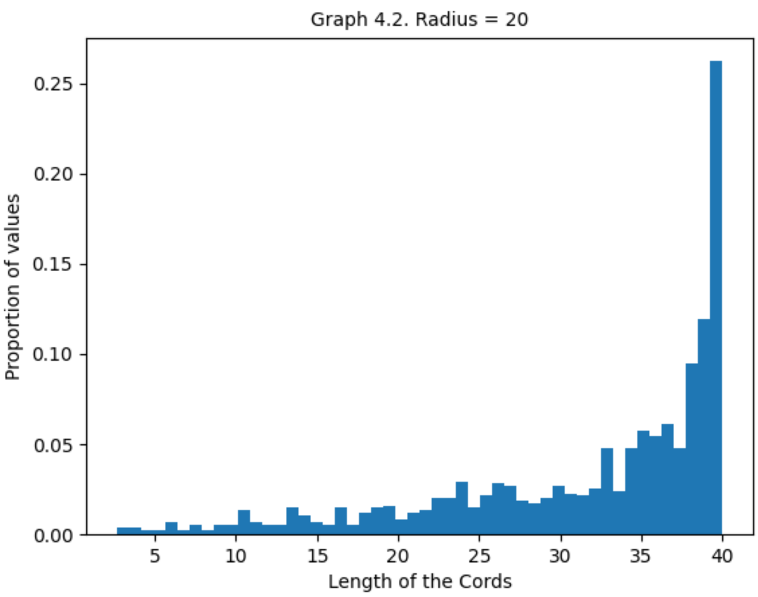
2. *find_ cords2(R)*: This function runs 1000 iterations of the *random_ cord(R)* function, and stores their result in the list *cord_ len*, sets the bin width as 50, and then graphs the histogram as the result.

We run this program with radius 30. The resulting histogram seems to be an exponentially rising graph, with a flat start.

```

# 4.2 ---------------------------------------------------------------------------------
def random_cord(R):

    theta = np.random.uniform(0,360)*(m.pi/180)
    point_at_radius = np.random.uniform(0,R)    #point at R

    #cartesian cordinates at the picked point
    x = point_at_radius*m.cos(theta)
    y = point_at_radius*m.sin(theta)

    #finding base line from center to point_at_radius using distance formula
    base = m.sqrt(x**2+y**2)
```

```
    #perpendicular using pythagorean theorem, p = sqrt(h^2-b^2)
    perp = m.sqrt(R**2-base**2)

    #length of the cord
    l = 2*perp

    return l

def find_cords2(R):

    cord_len = []
    I = 1000     #iterations

    for _ in range(I):
        cord_len.append(random_cord(R))

    #finding the bin size
    bin_val = 50

    plt.hist(cord_len, bins = bin_val,density=True)
    plt.title("Graph 4.2. Radius = {}".format(R),fontsize=10)
    plt.ylabel("Proportion of values")
    plt.xlabel("Length of the Cords")
    plt.savefig("Q4/Q4(2).png")
    plt.show()
```

Figure 7: Histogram of 4.2



4.3

For 4.3, it is similar to 4.2 with a slight difference. Instead of determining a direction, we pick a random point in the circle, and then calculates its distance from the origin $(0, 0)$. If the distance is within the circle as well, we regard it as our base/adjacent of the right triangle. Then we move towards calculating the opposite/perpendicular by using pythagorean theorem, multiply it by 2 to find the total length, and then store it in the list. Then we pass that list plotting functions to plot a histogram.

1. *p_to_o(cord)*: This function simply returns the distance between the randomly chosen point and the origin $(0, 0)$. It takes a tuple of cartesian cordinates as the input and returns the distance. It uses the distance formula,

$$dist = \sqrt{x^2 + y^2}$$

2. *random_point(R)*: It chooses random points for cartesian cordinates (x, y) between the R and -R.

3. *cal_cord(R,pnt)*: It takes the randomly chosen point, and finds its distance from the origin, then uses it as the adjacent/base of our right-angle triangle within the circle. Then it finds the value of the opposite/perpendicular via pythagorean theorem, and returns the twice of the opposite as the cord length.

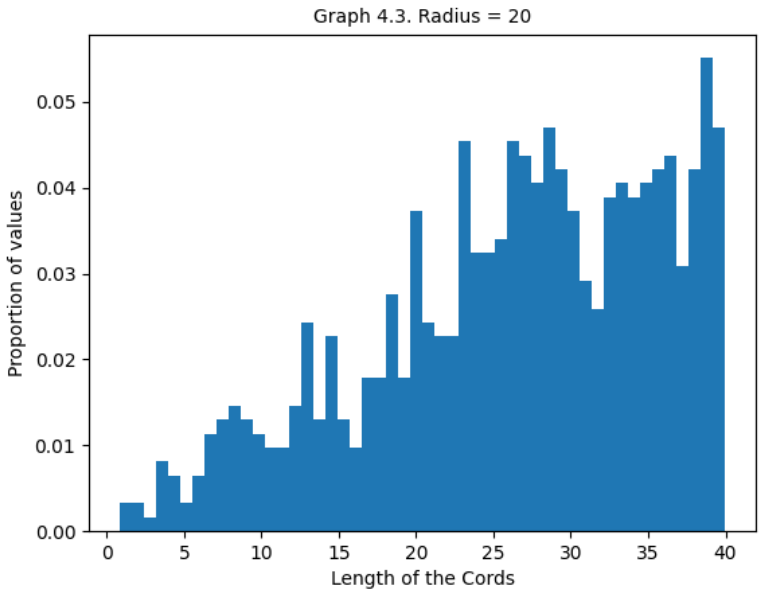
4. *find_cords3(R)*: This function runs 1000 times while calling the *random_point(R)* function, then it checks if the distance/base lies within the circle or not. If it does, then it proceeds to find *cal_cord(R,pnt)* and store it, else it decrements the counter by 1, to discard that value and find another.

We run this program for radius 20. The histogram looks quite linear.

```

# 4.3 ------------------------------------------------------------------------------------

def p_to_o(cord):
    return m.sqrt(cord[0]**2+cord[1]**2)

def random_point(R):

    x = np.random.uniform(-R,R)  #random point x
    y = np.random.uniform(-R,R)  #random point y

    return (x,y)

def cal_cord(R,pnt):
    #finding adjacent from the random point.
    adj = p_to_o(pnt)

    #length of opposite
    opp = m.sqrt(R**2-adj**2)

    #length of the cord
    l = 2*opp

    return l

def find_cords3(R):

    I = 1000
    cord_len = []

    for a in range(I):
        point = random_point(R)
        if p_to_o(point) <= R:
            cord_len.append(cal_cord(R,point))
        else:
            a = a - 1

    #finding the bin sizes
    bin_val = 50

    plt.hist(cord_len, bins = bin_val,density=True)
    plt.title("Graph 4.3. Radius = {}".format(R),fontsize=10)
```

```
    plt.ylabel("Proportion of values")
    plt.xlabel("Length of the Cords")
    plt.savefig("Q4/Q4(3).png")
    plt.show()
```

Figure 8: Histogram of 4.3



4.4

Out of all the distributions, I think the third approach (part 4.3) is the best to take. Its graph seems somewhat linear, which will ensure a little uniformity. As the previous graphs showed dense distribution towards the center of the circle, i.e. near the diameter, however in 4.3, the graph seems more distributed.