Probability & Statistics
Project

Hana Ali Rashid, hr05940
Tasmiya Malik, tm06183
Ifrah Ilyas, Student ID

April 21, 2021

Q1: Random Walk

1.1

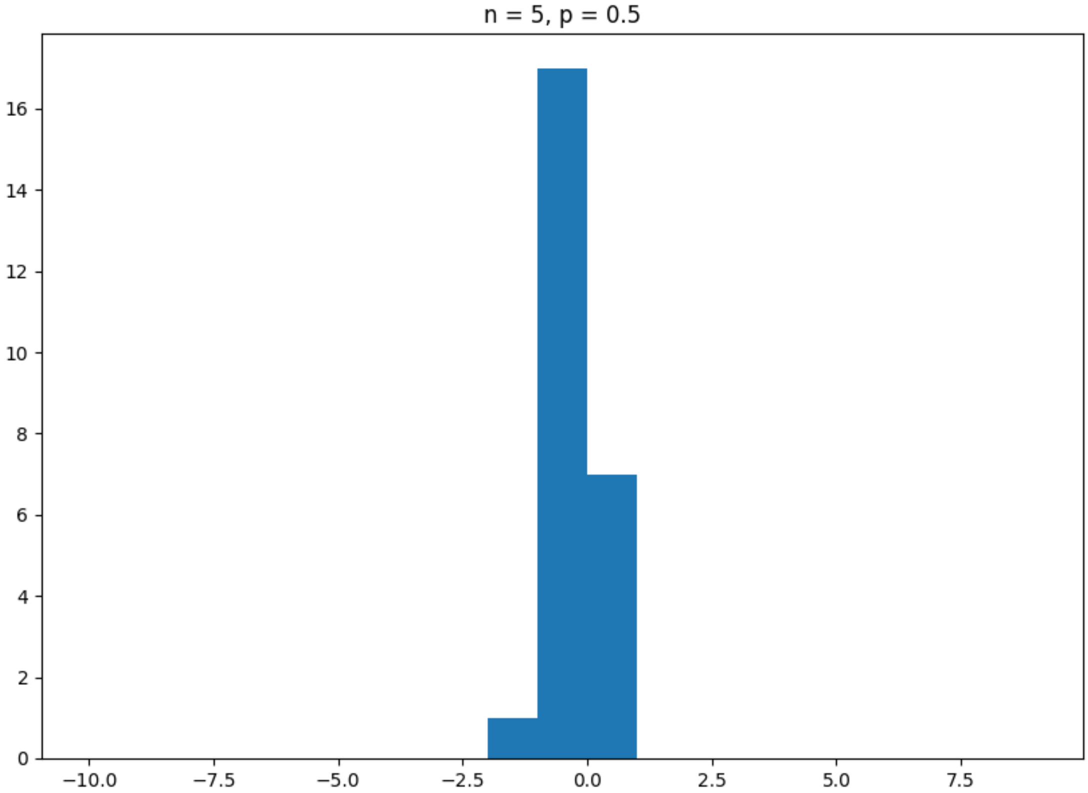
Function implementation in Python:

```
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
```

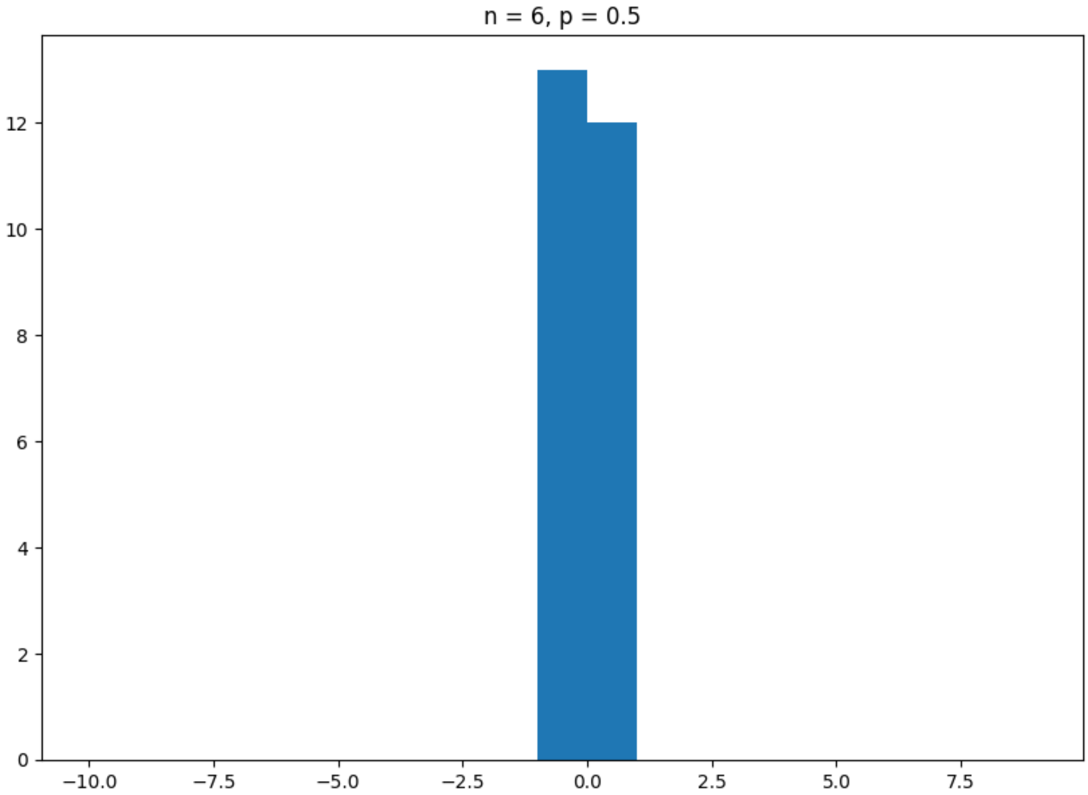
Calling the function for several iterations to get multiple expected values:

```
def main_11():
    '''Calculating expected outcomes for various combinations of n and p'''
    expected = []
    outcomes = []
    p = 0.5
    while p <= 0.9: #for values of p from 0.4 to 0.9
        for n in range(5,11): #for values of n from 5 to 10
            for j in range(25): #expected value for each (n,p) for 25 iterations
                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)
                outcomes = [] #resetting outcomes list
            #plotting and showing a histogram of calculated expected values
            fig, ax = plt.subplots(figsize =(10, 7))
            ax.hist(expected, bins = range(-10,10))
            plt.title('n = '+str(n)+', p = '+str(p))
            plt.savefig("Q1_histograms/q1"+'n = '+str(n)+', p = '+str(p)+'.png')
            plt.show()
            expected = [] #reset expected list
        p = round((p+0.1),1) #incrementing
```

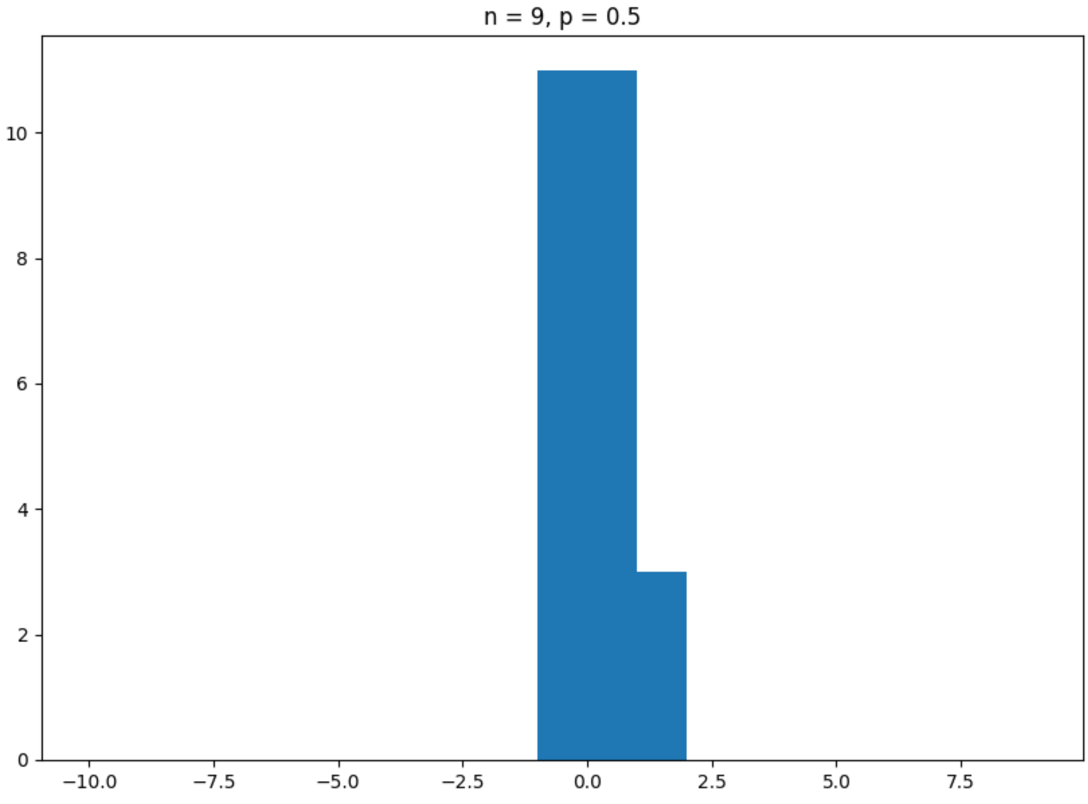
Histograms produced by the above code for various combinations of n and p:



The above histogram shows that...



The above histogram shows that...



The above histogram shows that...

1.2

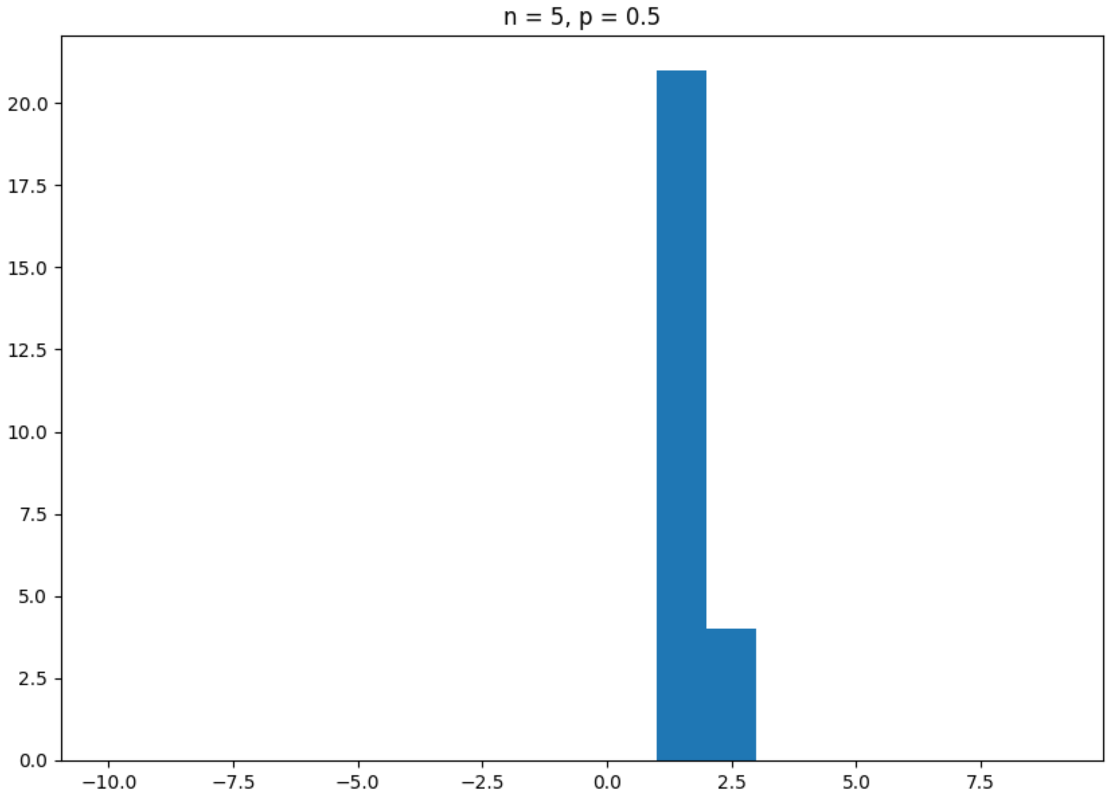
Function implementation in Python:

```python
def get_updated_position_restricted(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 or pos <= 0: #move one step right if pos == 0
            pos += 1
        else:
            pos -= 1 #move one step left
    return pos #return final position
```

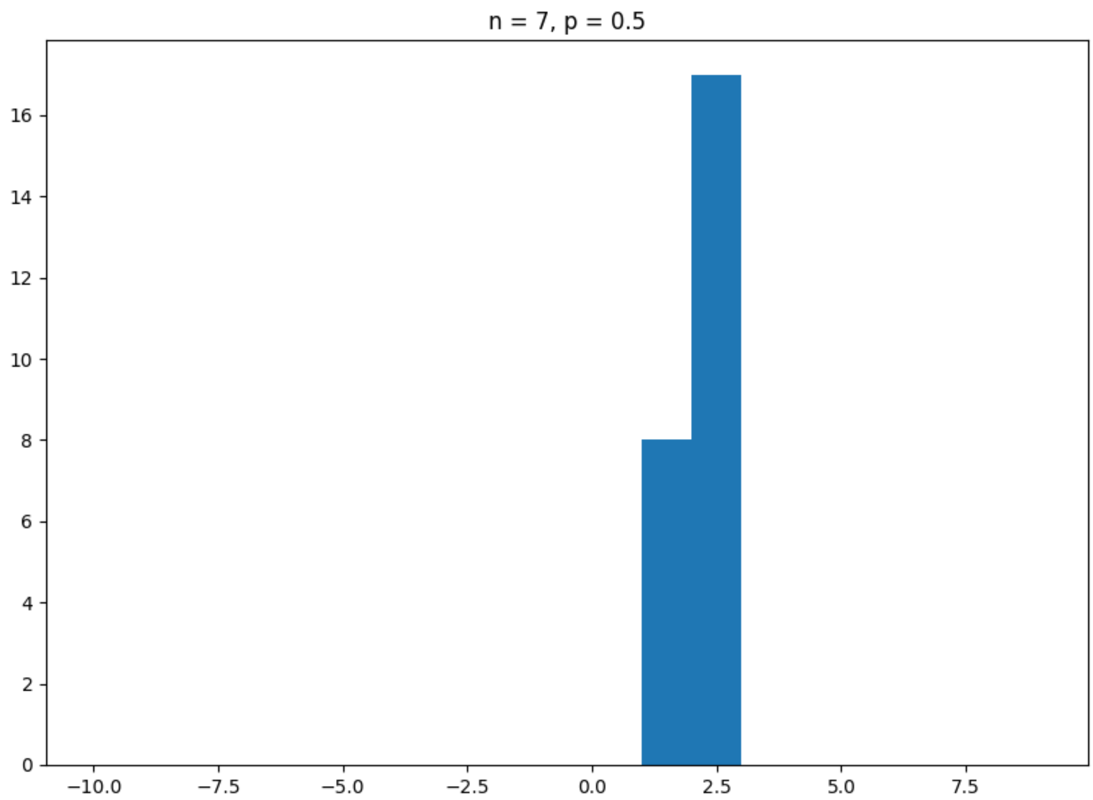
Calling the function for several iterations to get multiple expected values:

```python
def main_12():
    '''Calculating expected outcomes for various combinations of n and p'''
    expected = []
    outcomes = []
    p = 0.5
    while p <= 0.9: #for values of p from 0.4 to 0.9
        for n in range(5,11): #for values of n from 5 to 10
            for j in range(25): #expected value for each (n,p) for 25 iterations
                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)
                outcomes = [] #resetting outcomes list
            #plotting and showing a histogram of calculated expected values
            fig, ax = plt.subplots(figsize =(10, 7))
            ax.hist(expected, bins = range(-10,10))
            plt.title('n = '+str(n)+', p = '+str(p))
            plt.savefig("Q1_histograms/Q1.2 "+'n = '+str(n)+', p = '+str(p)+'.png')
            plt.show()
            expected = [] #reset expected list
        p = round((p+0.1),1) #incrementing
```

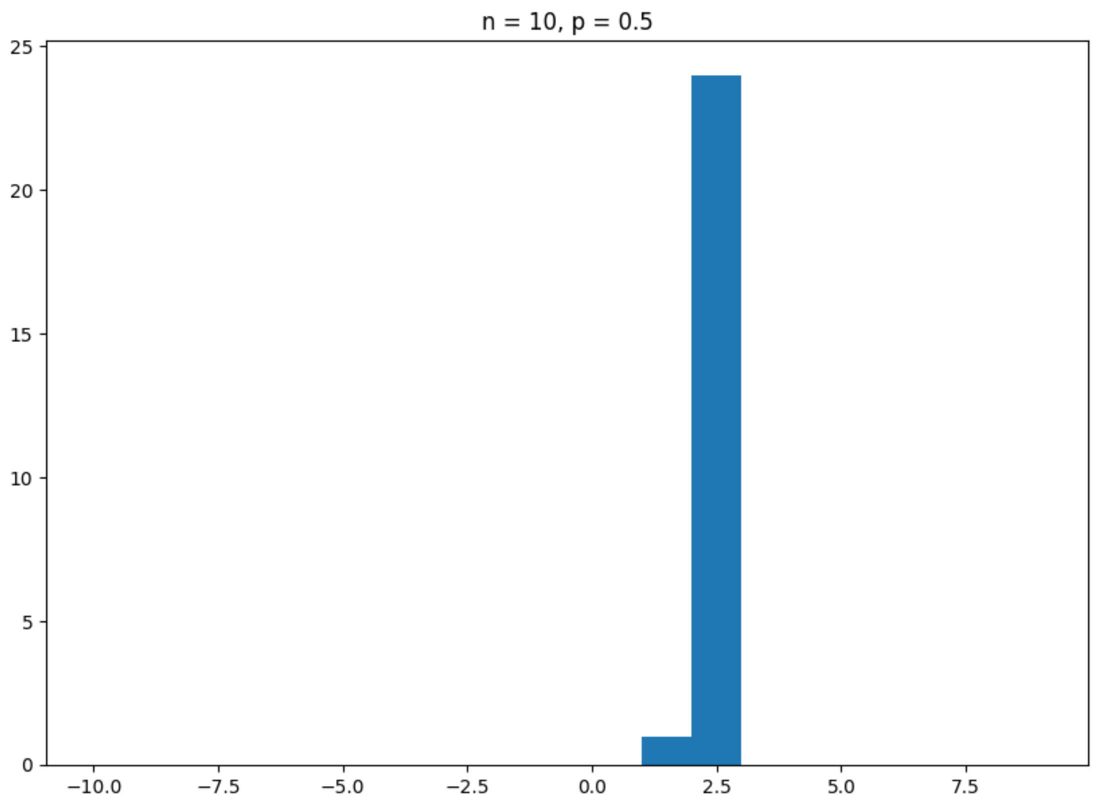
Histograms produced by the above code for various combinations of n and p:



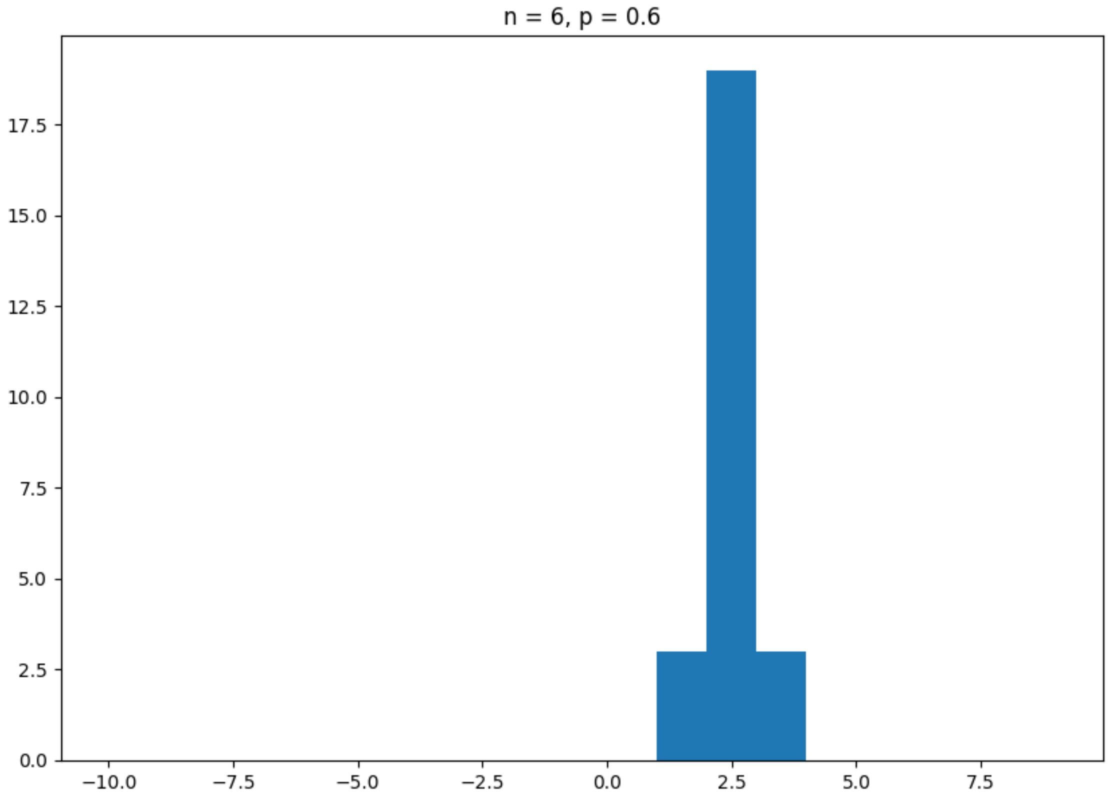
The above histogram shows that...



The above histogram shows that...



The above histogram shows that...



1.3

Function implementation in Python:

```
def stepsToMeet(pos1,pos2,p1,p2):
    count = 0 #keeps count of number of steps taken for objects to meet
    while pos1 != pos2:
        rand = random.randint(1,100) #generating a random number in the range 1 to 100 to determine outcome
        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
```

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

```
def main_13():
    '''Calculating expected outcomes for various combinations of n & p'''
    expected = []
    outcomes = []
    p1 = 0.5
    p2 = 0.5
    pos1 = -5
    pos2 = 6
    while p1 <= 0.9:
        while p2 <= 0.9:
            for i in range(25):  #calculating the expected value for each (n,p) for 25 iterations
                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)
                outcomes = [] #resetting outcomes list
            #plotting a histogram of calculated expected values
            fig, ax = plt.subplots(figsize =(10, 7))
            ax.hist(expected, bins = range(-10,10))
            plt.title('p1 = '+str(p1)+', p2 = '+str(p2)+', pos1 = '+str(pos1)+', pos2 = '+str(pos2))
            plt.show()
            expected = [] #reset expected list
            p2 = round((p2+0.1),1) #incrementing
        p1 = round((p1+0.1),1) #incrementing
```

Q3. Picking a Random Point Correctly

3.1

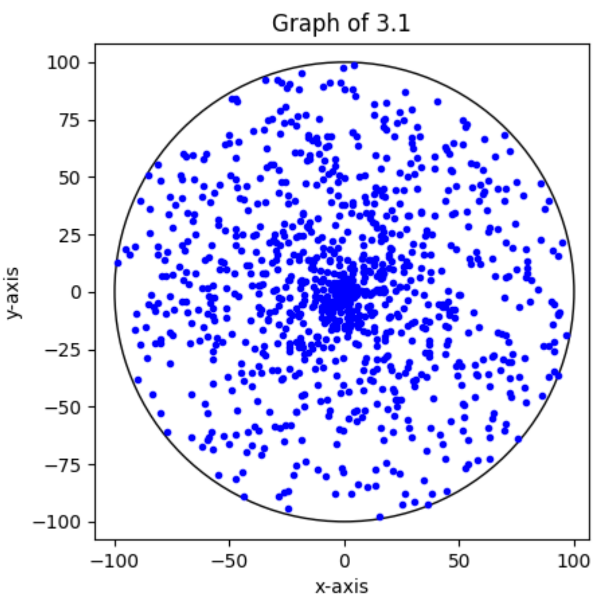
For part 3.1, we followed the method specified in the question. The program takes radius as an input from the user, and passes it to *gen_points(R)* to display the graph.

- *polar_to_cart(r,t)*: This function takes the polar cordinates ($r and \theta$) and converts them to their cartesian equivalent. It is called from *polar_point(R)*, to return the cartesian cordinates.
- *polar_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)*, 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.

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

```python

def random_point(radius):

    #R=radius, theta = angle
    R = np.random.uniform(0, radius)    #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(l_radius):

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

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

    print("The variance on the x-axis is " + str(np.var(X)))    #variance of x

    #Drawing the circle
    fig, axes = plt.subplots()
    circle = plt.Circle((0,0),l_radius,Fill=False)
    axes.set_aspect(1)
    axes.add_artist(circle)
    plt.title("circle")

    #Plotting X and Y
    plt.plot(X,Y,'.',color="blue")
    plt.xlabel("x-axis")
    plt.ylabel("y-axis")
    plt.title("Graph of 3.1")
    plt.show()

#Testing
R = input("Enter the radius: ")
gen_points(int(R))
```

Figure 1: radius = 100, variance of x = 1626.2176657086331



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.
- *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.

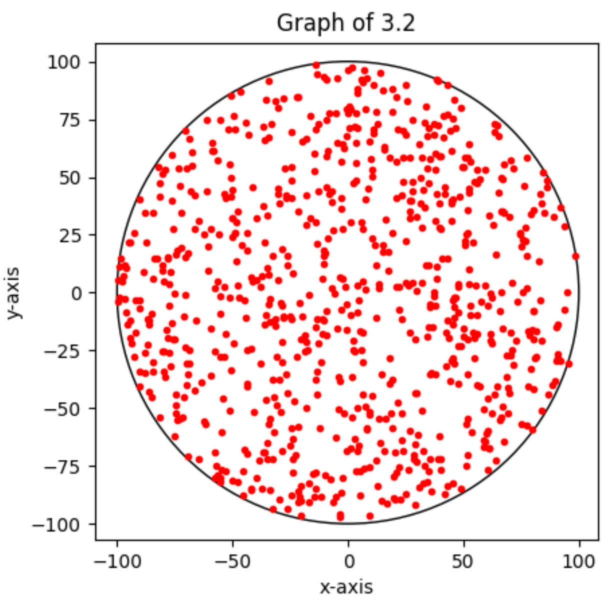
```
import matplotlib.pyplot as plt
import numpy as np
import math as m

def random_point(lmt_rad):

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

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

def gen_points(l_r):

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

    for x in range(I):
        point = random_point(l_r)
        if dist_from_origin(point[0],point[1]) <= l_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
            x = x-1

    print("The variance on the x-axis is " + str(np.var(X)))    #variance of X

    #Drawing the circle
    fig, axes = plt.subplots()
    circle = plt.Circle((0,0),l_r,Fill=False)
    axes.set_aspect(1)
    axes.add_artist(circle)
    plt.title("circle")

    #Plotting the points
    plt.plot(X,Y,'.',color="red")
    plt.xlabel("x-axis")
    plt.ylabel("y-axis")
    plt.title("Graph of 3.2")
    plt.show()

#Testing
R = input("Enter the radius: ")
gen_points(int(R))
```

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 part 3.2.

- *random_point(R)*:
- *polar_to_cart(cord)*:

Figure 2: radius = 100, variance of x = 2533.130556271337



- *gen_points(R)*:

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

def random_point(radius):

    #R=radius, theta=angle
    R = radius*m.sqrt(np.random.uniform(0, 1))  #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(l_radius):

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

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

    print("The variance on the x-axis is " + str(np.var(X)))    #variance of x

    #Drawing the circle
    #Outer Circle
    fig, axes = plt.subplots()
    circle1 = plt.Circle((0,0),2,Fill=False,color="red")
    axes.set_aspect(1)
    axes.add_artist(circle1)

    #inner circle
    circle2 = plt.Circle((0,0),1,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")
    plt.xlabel("x-axis")
    plt.ylabel("y-axis")
    plt.title("Graph of 3.3")
    plt.show()

#Testing
x = gen_points(2)
```

Figure 3: radius = 2, variance of x = 1.017564829759483