Q4. Saying Random is not enough

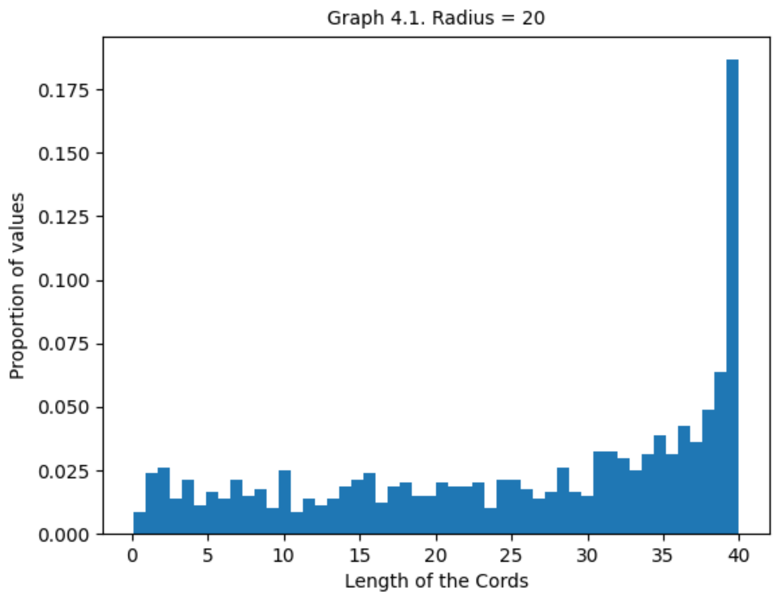
4.1

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.

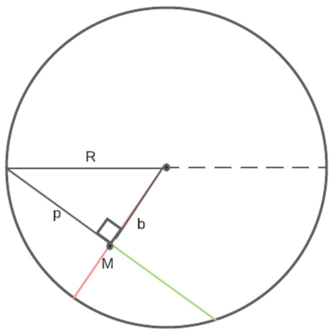
```python

# 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 1: 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,

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

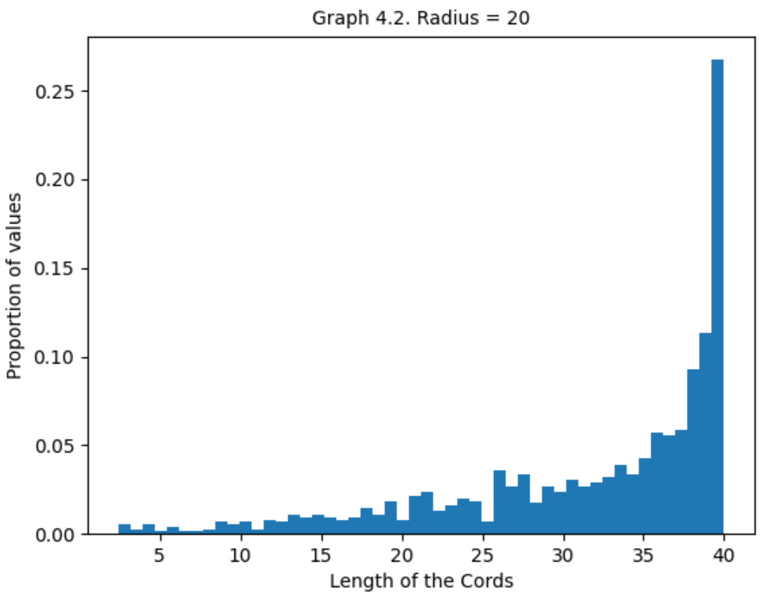
Then we simply multiply the value of perpendicular to find the length of the cord and return it.

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.

```python

# 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 2: 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.

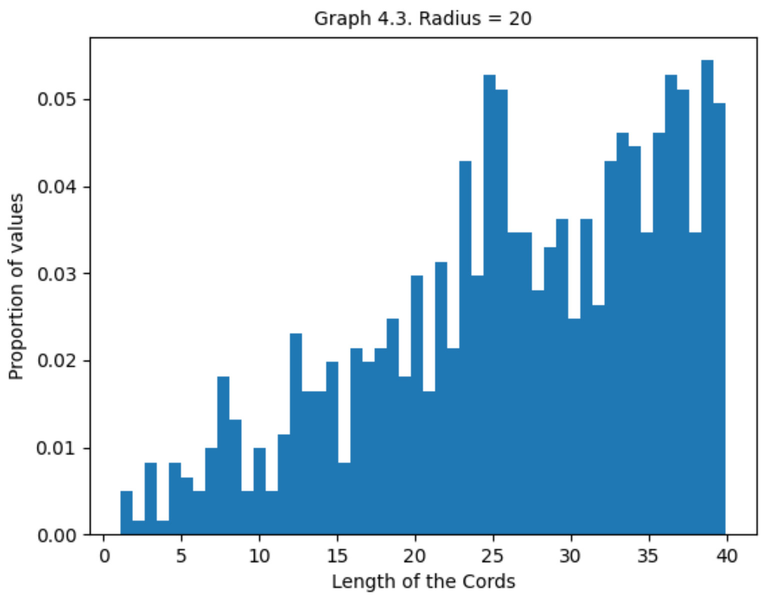
4. *find_cords3(R)*: This function runs 1000 times while calling the *random_point(R)* function, then it checks if the distance/base lies within the circle or not. If it does, then it proceeds to find *cal_cord(R,pnt)* and store it, else it decrements the counter by 1, to discard that value and find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

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

```python

# 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 3: 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.