Jared Rivera

804603106

CEE/MAE M20

February 21, 2017

Homework 6

1 The Chords-of-a-Circle Paradox

1.1 Introduction

The goal of this problem is to use Monte Carlo simulation to determine the probability of a randomly generated chord inside a unit circle will have a length greater than one. This will be done is two methods; random choice of angle and random choice of radius. For the random angle method, a random angle is generated and the chord is taken to be the length between the radius at 0 and the radius at the angle. For the random radius method, a random radius is generated and the chord is taken to be the length perpendicular to the random radius.

1.2 Model and Methods

The first step in this problem is to shuffle the randomly generated numbers created by MATLAB by using the shuffle command in the random number generator. Next, the radius of the circle in question is set (in this case as 1 because it's a unit circle), and the arrays that will hold the chord lengths are preset using the zeros function. Next the trials are run to perform simulation. Each trial in this case is run 1 million times and they work in a very similar method to each other, so I'll use the random angle as the example.

First, the random number that will correspond to the angle is generated as shown below

```
theta=rand*2*pi;
```

Where the random number (which has a value between 0 and 1) is multiplied by 2π to allow for all possible radian values between 0 and 2π to be generated. Next, the chord length corresponding to that angle is calculated using the function provided in the problem statement, and the array of chord lengths is populated with the chord length of that iteration, as shown below

```
anglechordlengths(k)=ltheta;
```

Where k is the current iteration. Next, the chord length is checked to see if it is larger than 1 by use of an if statement, and if so, a count variable is updated by adding 1. The count is preset to 0 at the start of the program and will serve to account for each chord length greater than 1. A similar procedure is followed for the random radius method.

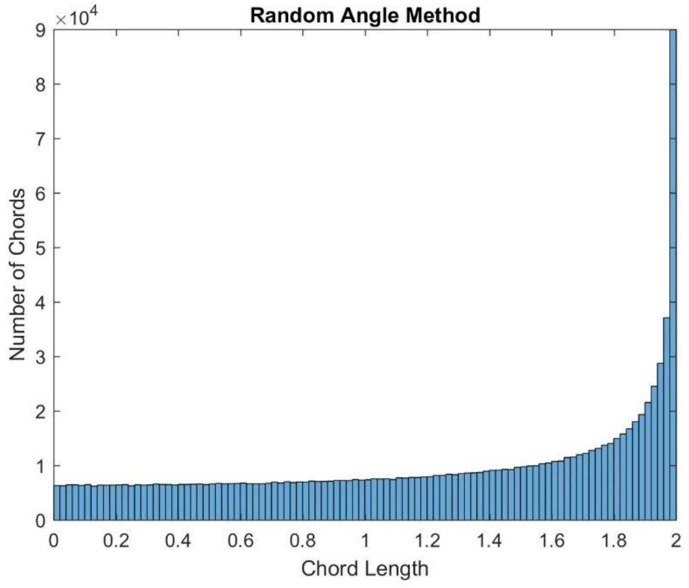
Next, the percentage chance of a chord being longer than 1 is calculated by taking the count variable for each case, dividing it by the total number of trials, and multiplying by 100. Now that the two percentages are calculated, they are printed to the command window in the format specified in the problem statement. Finally, the histograms are created using the built-in histogram function and axis titles and heading are assigned in a similar fashion to a standard plot.

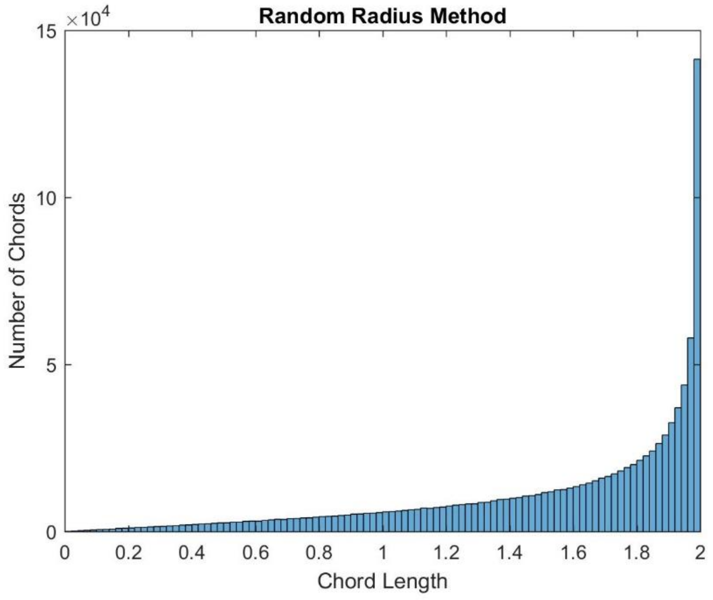
1.3 Calculations and Results

When the program is run, the following is printed to the command window:

```
Random Angle: 66.63 percent
Random Radius: 86.60 percent
```

And the following histograms are displayed:





1.4 Discussion

The large discrepancy seen between the values created by the two methods can likely be attributed to the nature of the difference between a radius taken in measure of length, and an angle taken by measure of radian. For example, take the generation of two random numbers, 0 and 1. These are the endpoints of the possible range. In the case of the random angle example, the angle in generated by multiplying the random number by 2π. In this case that creates angle values of 0 and 2π. In the case of the random radius example, the radius is taken to be exactly the random number, so 0 and 1. The difference between these methods is easily seen here because the random angle method now creates two of the same exact chord, while the random radius method creates two entirely different chords. In the case of the random angle, the values begin to repeat after π due to the nature of the geometry while the radius continues to generate different values. Because of this, the random radius method may be better for this problem due to it's more inherently random ability to be programmed.

2 Random Walker Collisions

2.1 Introduction

The goal of this problem is to use Monte Carlo simulation to estimate how many steps it takes for two "random walkers" to collide, or find each other, in a domain of specified size. This will be done by programming two random walkers and tracing their position over time in order to determine where and when they collide. Through this, we can estimate over a certain number of trials the median number of movements, or steps, needed to be taken by the walkers until they collide. This program can then be used to determine whether the walkers will find each other more quickly if they both move at random or if one of them stays still.

2.2 Model and Methods

The first step to writing this program, as in the previous, is to shuffle the random numbers generated by MATLAB. After this, the number of trials to be run in the simulation is specified by the NumTrials variable (5000 in this case). Next, the array that will contain the number of steps needed for collision is initialized with the zeros function, preallocating a space for each trial. With this set up, the outer loop starts. The outer loop runs the simulation NumTrials times, and at the beginning of each loop initializes the positions of the random walkers. It also sets the collision criteria to false and sets the number of moves take by the walkers in that trial to zero.

The next step is to move the walkers. This is done by setting up a while loop that will run so long as the walkers aren't in the same position, or will max out at 1000 steps because there is no guarantee they will meet. In the loop, the positions of the walkers are fed into the walkitout function that will update their positions, and will be explained below. After the walkers are moved, the move count is updated by 1, and the positions are checked for collision. If the positions of the walkers are exactly equal, the collision criteria is updated to true, the array that contains the move count to collision is populated, and the next trial is started. The code for this set of operations is shown below

```
if xa==xb && ya==yb
    collision=1;
  movecount(k)=move;
        end
```

Where k is the current simulation iteration and move is the move count of that trial.

The walkitout function moves the walkers in the following fashion. It sets the boundary conditions (i.e. a wall preventing motion) then generates a random number. The direction of motion is then determined by the value of the random number and whether or not the walker has already reached a boundary condition. The case for North movement is shown below.

```
if r<=0.25
%Move north
if y~=wall
  y=y+1;
   end
```

And conditions similar to these are followed for the remaining directions.

Finally, the median number of moves needed to collide is take to be the median of the values in the array containing the move counts per trial. The result is then printed to the command window.

2.3 Calculations and Results

When the program is run in part b, the following is printed to the command window:

```
Median = 203
```

For part c, the following is printed:

```
Median = 267
```

Where these exact values depend on the given trial.

2.4 Discussion

As seen in the results section, it would appear that if two random walkers became separated they would find each other the quickest if they were to both move at random, rather than have one of them stay still. For the 5x5 grid in this problem, this seems like a reasonable outcome because one would picture tow walkers reaching each other in a space that size would be more probable than one seeking out the exact position of the other at random.

One odd thing to note, is that if the size of the grid is doubled, from 5 to 10, the number of moves needed to find each other when both walk at random is less than half the original. This outcome entirely goes against my intuition as there is more open space for the walkers to independently occupy in such a case. If the size of the grid matters, the initial positions of the walkers surely do as well. In the case that each walker is started at the (4,0) and (-4,0), respectively, the walkers reach each other in 193 moves. This is less than the original and makes sense due to their close proximity to each other. This trend continues when the walkers are moved to the (3,0) and (-3,0) positions, where the median number of moves is 180. These changes may seem small, but it is also important to note that when the walkers do not start on a barrier, their degree of freedom of motion increases by 25% as well.

Combining this program with last week's split and average problem, it would likely be possible to use a weight vector to allow the walkers to account for each other's current position before making a move, which would likely create a very different outcome in this simulation.