Duncan Di Mauro
UID: 805163177
CEE/MAE M20
November 12, 2019

Homework 6

1 Implementing Custom Probability Distributions

1.1 Introduction

The goal in this problem is to implement a method to draw samples from

$$p(x) = \frac{1}{2}x + \frac{1}{2} \text{ if } x \in [-1, 1] \text{ and } p(x) = 0 \text{ otherwise} \quad (1)$$

using randomly generated numbers. The cumulative probability density function

$$P(x) = \int_{-\infty}^{x} p(u)du \quad (2)$$

should be explicitly calculated. A function should be created that samples *p(x)* by drawing a random sample *y* from a uniform distribution on the domain [0, 1] and then get *x* by using $x = P^{-1}(y)$. Finally, 1,000 samples should be drawn using that function, and a histogram of the results should be created and plotted along with *p(x)*.

1.2 Models and Methods

First, equation (2) was explicitly calculated by hand. The results are as follows:

$$P(x) = (0.25x^2 + 0.5x + 0.25) \text{ when } x \text{ is within [-1, 1]} \quad (3)$$

$$P(x) = 0 \text{ if } x < -1 \quad (4)$$

$$P(x) = 1 \text{ if } x > 1 \quad (5)$$

Then, a function `x = myRand()` was created. This function draws a random sample for *y* from a uniform distribution on the domain [0, 1] (this was done using the `rand` function) and then outputs a value for *x* using $x = P^{-1}(y)$, which was found by rearranging equation (3):

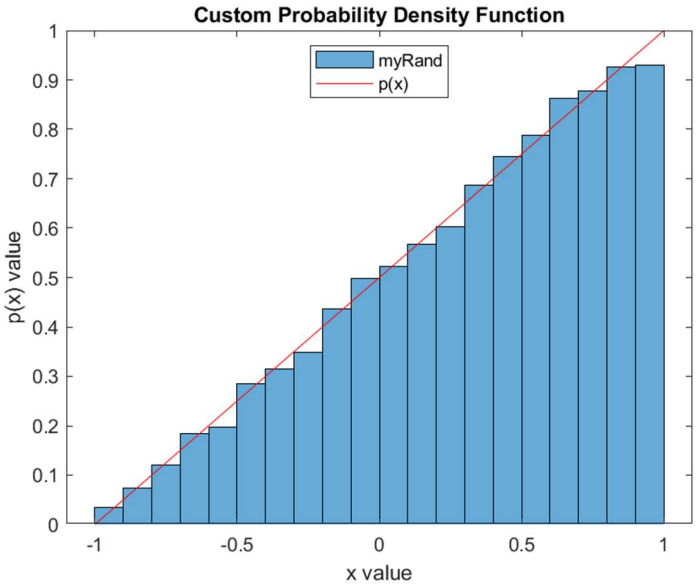
$$x = P^{-1}(y) = 2\left|\sqrt{y}\right| - 1. \quad (6)$$

The positive value of the square root of *y* is taken because *x* goes from -1 to 1. If the negative value was taken, *x* would go from -3 to -1.

Next, a script is written which uses a `for` loop to take 1,000 samples of the `myRand()` function and assign the *x* values to a vector. Finally, a histogram of the results is created by using the `histogram` function in MATLAB and setting the `Normalization` property to `pdf`. Additionally, the function *p(x)* was plotted alongside this histogram. The values for *p(x)* were created using two vectors, one including *x* values on the domain [-1, 1] with a sufficiently small step size (a step size of 1/1000 was chosen here), and the other vector being the *y*-values found using Equation (1).

1.3 Calculations and Results

When the main script is executed, the following figure is created (note that the figure will not look the same every time due to the variation of random numbers in this problem and the finite sample size):



1.4 Discussion

This problem was very simple and not many new concepts were used. The only new concepts included learning how to utilize the `rand` and `histogram` functions. In the plot shown above, we see the histogram created using `myRand`, along with a line representing *p(x)*. We notice that the histogram approximates the shape of *p(x)* very well, with a small and relatively equal amount of deviation in either direction. The code was run several times to produce several plots, and these properties are true of each histogram produced. There is a small amount of error at individual points along the histogram, but this can be improved by increasing the sample size and reducing the width of each bar on the histogram. It can therefor be concluded that our implementation approximates the probability density function *p(x)* decently well and consistently.

2 Monty Hall

2.1 Introduction

In this problem, we will simulate the Monty Hall problem (look online to see what this is). The goal here is to see whether or not it is in one's best interest to switch doors (i.e., to find the probability of winning given that the contestant always switches doors). A script should be written that randomizes the contestant's selection of a door and the location of the prize using the `rand` function. The script should run 1,000 trials and count the number of trials in which switching was beneficial.

2.2 Models and Methods

First, the script creates variables `Win` and `Loose` that represent the probabilities of winning and losing, respectively, assuming that the contestant chooses to switch doors. These variables are set equal to 0 and will be updated later in the script. Then, a `for` loop is created that runs the given amount of trials (1,000). At the beginning of the loop, the contestant's door (`guess1`) and the prize door (`prize`) are

randomly assigned to integers between 1 and 3. These integers are representative of the three doors in the Monty Hall problem (i.e., Door 1, Door 2, and Door 3). This done by using the `rand` function to obtain a number between 0 and 1, multiplying that number by 3, and then rounding that number up to the nearest integer using the `ceil` function. It is necessary to always round up so that a value of 0 is never obtained.

For the next part of the Monty Hall problem, the host opens one of the two remaining doors to reveal a goat. More specifically, the host opens a random door that is not the prize door and is not the contestant's door. This is done by using a `while` loop, as shown below:

```
FLAG = false;
while ~FLAG
    OpenDoor = ceil(rand()*3);
    if OpenDoor ~= guess1 && OpenDoor ~= prize
        FLAG = true;
    end
end
```

Next, the contestant decides to switch to the remaining door (represented by `guess2` in the code). This means that the contestant switches to the door that is not `guess1` and is not the `OpenDoor`. Another `while` loop is used to assign the relevant door value to `guess2`.

Finally, if the value of `guess2` is equal to the value of `prize`, it is concluded that the contestant wins, and the value for the probability of winning is increased by the current value of `Win` + 1/(The number of samples). At the end of all the trials, this will result in the variable `Win` being equal to the total number of wins divided by the total number of trials, which we can assume to be equal to the overall probability of winning (assuming a large sample size). This same process is done for the variable `Loose` if `guess2` is not equal to `prize` (calculating the value of `Loose` is not necessary but is helpful in making sure that the code works properly). Then, the value of `Win` is printed to the screen.

2.3 Calculations and Results

When the script is executed, the following text is printed to the command window (note that the printed value will not be the exact same every time due to the variation of random numbers in this problem and the finite sample size):

```
The probability of winning is:  0.667500
```

2.4 Discussion

This problem was very straightforward and involved new techniques surrounding the `rand` function, such as utilizing algebraic techniques to generate a select few integers randomly. Additionally, this problem introduced the Monte Carlo method as a way of solving certain problems. The goal of this problem was to see whether or not it was beneficial to switch doors in the Monte Hall problem. When the contestant first chooses a door, the probability of choosing the door with the prize is 1/3. Even if the host reveals an empty door later, this probability stays the same, and is the probability of winning assuming that the contestant never switches doors. However, if the contestant does choose to switch doors, results of our code show that the probability of winning is approximately 2/3 (the expected

answer of the Monte Hall problem). After running the script multiple times, the resulting probability stays close to this number. Therefor, it can be concluded that switching is indeed beneficial, and our script is able to simulate the Monte Hall problem very well.

3 Numerical Integration with Monte Carlo

3.1 Introduction

The goal of this problem is to numerically integrate the following function using a Monte Carlo method:

$$\int_0^5 \frac{1}{x^2+1}\,dx \tag{7}$$

Random samples should be picked in a rectangle containing the region of the function to integrate. Then, the number of samples under the curve should be counted and utilized to produce an estimated value of the area under the curve (i.e., the value of the integral). Finally, the results should be plotted as a function of the numbers of samples picked. The constant arctan(5) should also be plotted on the same graph to show how the numerical results converge to this expected answer.

3.2 Models and Methods

The script first assigns variables to all initial values, including the rectangle's area (this is equal to 5 since the rectangle containing the function has a height of 1 unit and a width of 5 units), the number of samples under the curve (this is originally set to 0 and will be updated later), the maximum number of samples to use (this was chosen to be 100,000), and the step-size for the number of samples (this was chosen to be 100 in order to reduce run-time of the code, but can be smaller for a greater number of data-points). Space is also pre-allocated for the vector containing all calculated area values, which has a number of columns equal to (The maximum number of samples)/(step size for number of samples).

Next, a `for` loop is created for the number of samples, starting at the step-size (this allows for easier code manipulation) and ending at the maximum number of samples. Afterwards, an additional `for` loop is created that chooses a number of random points on the rectangle equal to the current number of samples set by the first `for` loop. The *x* and *y* values of the points are randomly assigned to values within the domain and range of the rectangle, respectively. This is shown below:

```
randomx = rand()*5; %Range of x-values is between 0 and 5
randomy = rand(); %Range of y-values is between 0 and 1
```

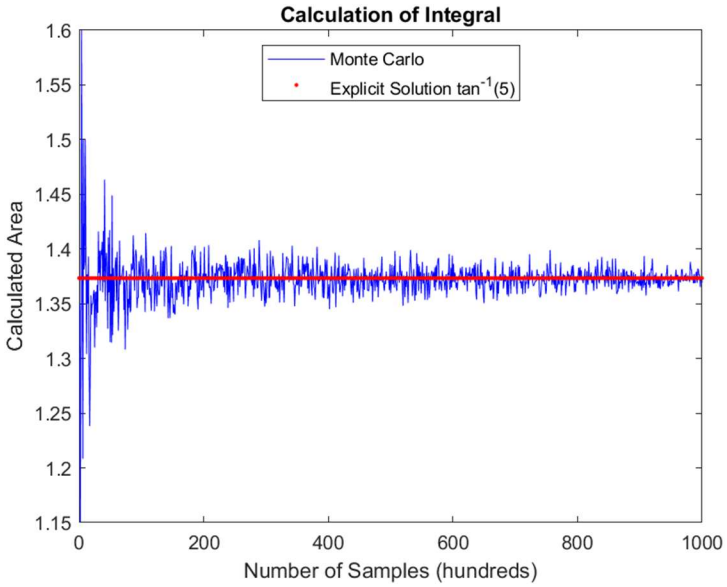
Next, the code checks if the random point lies under the curve. If the random *y* value obtained is less than the value of the given function (The inner-part of equation (7)) at the random *x* value, then the point lies under the curve. If this is the case, then the value for the number of samples lying under the curve is updated by `SamplesUnder = SamplesUnder + 1`. After all the random sample points are taken, the inner loop ends.

Afterwards, the area (which equals the integral value) is calculated by multiplying the probability that a sample will be under the curve (number of samples under the curve divided by the number of samples taken) and multiplying this by the area of the rectangle. Then, the calculated value is placed into our area vector, the variable for the number of samples under the curve is set back to 0, and the first `for` loop moves on to the next sample size.

Finally, the vector containing the calculated area values is plotted against sample size. Additionally, the expect value of the integral (arctan(5)) is plotted on the same graph using the same methods as in problem 1, and results are discussed.

3.3 Calculations and Results

When the script is executed, the following figure is created (note that the figure will not look the same every time due to the variation of random numbers in this problem and the finite sample size):



3.4 Discussion

This problem was very straightforward and did not involve any new concepts. The graph above shows that the numerical results for the integral converge to the expected value of arctan(5). The script was run several times to produce several plots, and this convergence is true for each plot produced. However, it is interesting to note that the convergence takes a while, as shown by the graph above, where there is still a visible amount of error, even after a sample size of 100,000.

4 Dropping a Needle on a Grid

4.1 Introduction

The goal of this problem is to simulate dropping a given number of needles with length *L* onto a grid with unit-length square tiles. A function should be created that takes three arguments: the needle length *L*, the width of the entire grid, and the height of the grid. This function should return three values: x-position of the needle, y-position of the needle, and the angle θ that the dropped needle makes with the x-axis. Additionally, the function should drop the needle again if the needle crosses the

grid boundaries. Finally, the code should drop 10,000 needles and count the number of needles that cross grid-lines. The simulation should be repeated for *L* = 0.1, 0.2, ... ,1.6, and the probability versus needle length should be plotted.

4.2 Models and Methods

First, a function `[x, y, theta] = dropNeedle(L, gridWidth, gridHeight)` is created. The function starts by obtaining the radius of the needle, which is equal to half the length. Next, a `while` loop is used in the same fashion as problem 2 in order to obtain random values for x, y, and theta, whilst also ensuring that the needle does not cross any gird boundary. For the x and y-values, respectively, a random number between 0 and `gridWidth` is obtained, and a random number between 0 and `gridHeight` is obtained. Note that for this problem, the x and y values generated were considered to be at the center of the needle. For the value of theta, a random value between 0 and pi is generated (since the needle cannot go below the x-axis, it cannot make angles greater than pi with the x-axis). Next, values for the length of the radius in the x direction and length of the radius in the y direction are obtained using trigonometry. Finally, at the end of the `while` loop, the following condition is presented to ensure that the needle does not cross the boundary:

```
if x - radiusX >= 0 && x + radiusX <= gridWidth ...
            && y - radiusY >= 0 && y + radiusY <= gridHeight
        FLAG = true;
end
```

Afterwards, the function ends and the main script for this problem is created. The script first assigns all given values (grid length, grid width, and sample size) to relevant variables. Next, a variable representing the probability that a needle will cross a grid-line is initialized to zero. Additionally, space is pre-allocated for the vector holding all calculated probabilities for different *L*s, and for the vector holding all values of *L*.

Next, a `for` loop is created that simulates the dropping of needles for *L* = 0.1, 0.2, ... , 1.6. A vector containing the iteration number is then created and updated accordingly. Then, the vector containing all *L* values is updated for each iteration. Afterwards, the radius of the needle is assigned to a variable.

Another `for` loop is then created, which simulates the dropping of 10,000 needles (the given sample size). The script uses a call to the previously created `dropNeedle(L, gridWidth, gridHeight)` function in order to obtain random values for `x`, `y`, and `theta`. Afterwards, values for the length of the radius in the x direction and length of the radius in the y direction are obtained using trigonometry.

An `if` loop is then created, which updates the probability of a needle crossing a gridline in the same fashion as problem 2. The `if` loop states that the probability is increased by 1/(the sample size) if any of the following are true:

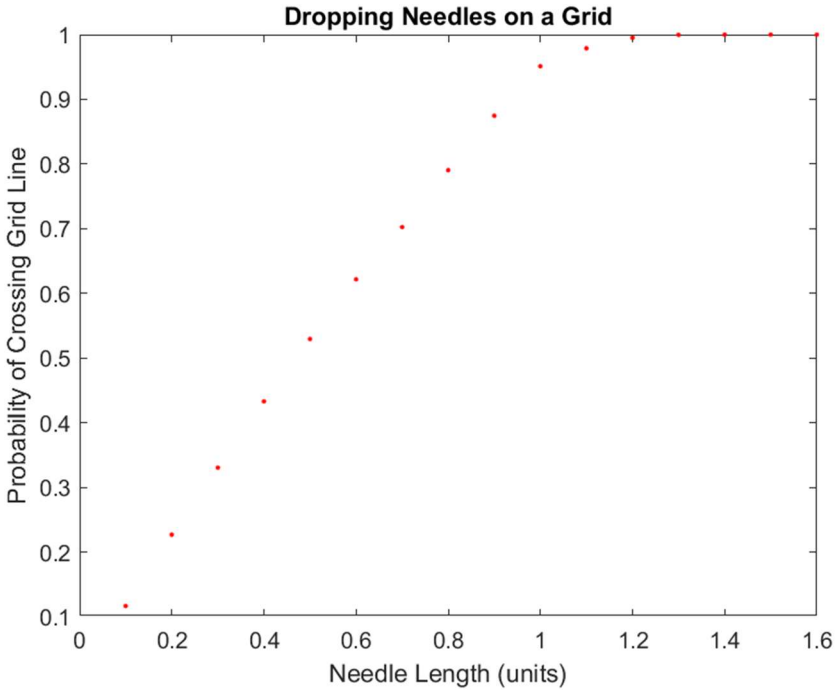
1. The x-values at the needle's ends round up to different integers
2. The y-values at the needle's ends round up to different integers
3. The x-value at the needle's rightmost tip is an integer
4. The y-value at the needle's upper tip is an integer

If any one of these is true, it means that the needle crosses (or touches) a grid-line. The `if` loop then ends, along with the inner `for` loop.

Next, the vector containing the probabilities for different lengths of *L* is updated using the value of the probability variable. Afterwards, this value is set back to 0 so that the first `for` loop can move on to the next value of *L* and calculate the new probability. The original `for` loop then ends and probability is plotted against needle length *L*.

4.3 Calculations and Results

When the script is executed, the following figure is created (note that the figure will not look the same every time due to the variation of random numbers in this problem and the finite sample size):



4.4 Discussion

This problem was very similar to the previous problems and did not involve any new concepts. If we look at the graph above, we notice that the probability increases with needle length, and converges to 1 at a certain length. This agrees with my intuition that a longer needle will be more likely to cross a grid-line, since the space between the grid-lines does not increase in size. Additionally, this agrees with my intuition that at some point, the needle will become so long that it will be impossible for it to *not* cross a grid line. Mathematically, a needle will always cross a grid-line once it is equal to $\sqrt{2}$, which is the diagonal of a unit-square, which is what our grid consists of. This value is a little larger than 1.4, but on the graph, it appears as though the probability becomes 1 starting at a needle length of 1.3. This is understandable since we use a finite sample size of 10,000 needles and it is still very unlikely that a needle of length 1.3 will be in the exact spot so as to not touch a grid-line. I do not expect a different probability for L = 1.5 and L = 1.6, since all probabilities starting at L = $\sqrt{2}$ will be equal to 1.