Sruti Munagala

UID: 605102910

CEE/MAE M20

November 19, 2019

Homework 7

1 Euler-Bernoulli Beam Bending

1.1 Introduction

The goal of this problem is to examine the deflection of a steel beam when it is subjected to a force of 800 N. We are given constants, such as the length, outer width, inner width, force, and where the force is applied, as well as the equations for the bending moment along the beam. We must build matrices to find the deflection, plot the deflection versus time, calculate the percent error between the calculated deflection and the theoretical deflection, and experiment with varying constants.

1.2 Model and Methods

First, I made a function called bendingMoment1.m that I used in my main function. bendingMoment1 is a function that calculates and outputs a vector named Mx that contains the different values for the bending moment across the beam. This function takes the force exerted on the beam, the length of the beam, the location the force is exerted, and the number of nodes in as input parameters.

bendingMoment1 begins by calculating the step size, delta_x, which could be calculated from the length of the beam and the number of nodes-1. (Why is it number of nodes-1??????) Then, we calculate the total number of steps that we have to take to travel from one end of the beam to the other. We need this value so that the for-loop knows how many iterations to make. Next, we make an vector, called 'x,' that contains linearly spaced x-values that range from 0 to L. We will pass 'x' into our function and use each individual element to calculate the different bending moments:

```
delta_x = L/(n-1);
totalsteps = L/delta_x;
x = linspace(0,L,n);
```

Next, we create a vector Mx, which will contain all of the values for bending moments for the various values of x using the zeros function, as we don't want any particular values in this array. Then, we enter the for-loop, which iterates from 1 to the value of totalsteps, which should be 20. Because there are two different equations for the bending moment provided, I used an if-statement. If x, the distance from the left end of the beam, is between 0 and d, the equation for the moment is Eq. (1): $-\frac{P(L-d)x}{L}$. For the remainder of the beam, the equation for the moment is Eq.(2): $-\frac{Pd(L-x)}{L}$. Because x is either less than or equal to d (in which case we'd use Equation 1), or greater than d (in which case we'd use Equation 2), I used an if-else statement in my for-loop to take both equations into account:

```
for m=1:totalsteps
      if x(m) <= d
            Mx(m) = ((-P*(L-d)*x(m))/(L));
      else
```

```
            Mx(m) = ((-P*d*(L-x(m)))/(L));
        end
    end
```

As you can see, the if-statement contains Equation 1 and is used when x is less than or equal to d. When x is greater than d, we switch over to the 'else' condition and Equation 2 is used.

Now we'll move onto the main function. We begin by declaring the number of nodes, n, we are using, initializing the A matrix using the zeros function, and inputting the boundary conditions for A. 'n' is also equal to the number of rows and columns our A matrix will have. Now, we must fill our A matrix with the correct values using a for-loop. From row 2 to the (n-1)th row, we need to change certain zeros to either 1, -2, or 1. The leading one must form a diagonal line going from the top left to the bottom right of the matrix. The variable c is set to 1 and r is set to 2 so that we start at A(2,1). We change that matrix element from a 0 to a 1. Then, we shift columns by moving to the right by adding one to c. We change the new element we're at to -2, and then move to the next column again. We change that element to a 1, and then move one column to the left and down one row and repeat the process:

```
    for j = 2:n-1 %For rows 2 to the (n-1)th row, we need to change
    certain values
        A(r,c) = 1; %In the current location, change the 0 to a 1
        c = c+1; %Shift columns (move to the right)
        A(r,c) = -2; %Change the new element from a 0 to a -2
        c = c+1; %Shift columns (move to the right)
        A(r,c) = 1; %Change the new element from a 0 to a 1
        c = c-1; %Shift columns back (move to the left)
        r = r+1; %Move down in the matrix (move to the following row)
 %We continue to replace elements with 1,-2,1 until we reach the second
%to last row (the (n-1)th row)
    End
```

Next we have to declare all of the constants we use, such as P (the force exerted onto the beam), L (the length of the beam), d (where the force is exerted), E (the modulus of elasticity), delta_x (the step size), a (the outer width of the beam), b (the inner width of the beam), and I (the moment of inertia, which we calculate from a and b). Then we initialize an n by 1 array for the right-hand size vector b.

Because b, Mx, and x are all vectors, we can simply use matrix operations to fill the b array with the correct values:

```
    b = (((delta_x)^2)*bendingMoment1(P,L,d,n))/(E*I);
```

(This is modeled after $y_{k+1} - 2y_k + y_{k-1} = \frac{\Delta x^2 M(x)}{EI}$.)

Next, we calculate the deflection vector, y, by dividing A by b. We must use a backwards slash in order to do this because we are dividing matrices. The next step is to plot the deflection vector, y, and the position along the beam, x. So, we make a vector for x using the linspace function. Here, the first parameter in the parenthesis is the starting point, the second parameter is the ending point, and the third parameter is the number of points we wish to have in between:

```
    x=linspace(0,L,n);
```

Then we plot x and y and add titles and axes labels. We include the 'o-' so that the points on the plot can be seen as dots that are connected:

```
    plot(x, y, 'o-')
```

Then we find the point of maximum deflection by using the min function. We will use this value when calculating our theoretical maximum deflection. We calculate the theoretical maximum deflection using the equation provided in our homework specification:

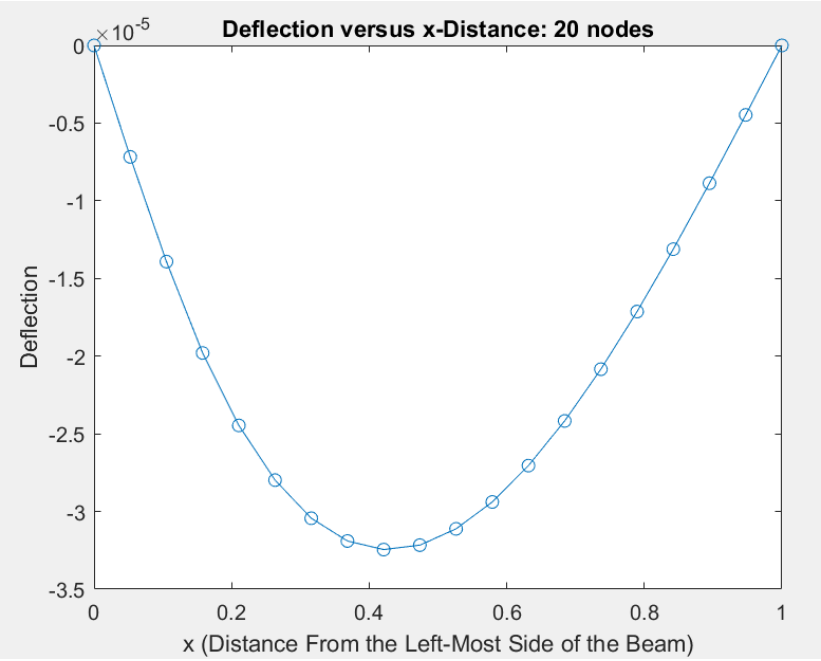
```
    ymax = min(y);
    ytheo = (P*c*((L^2-c^2)^1.5))/(9*sqrt(3)*E*I*L);
```

Then, we calculate the percent error and print it to the command window:

```
percenterror = (abs(ymax - ytheo)/abs(ytheo))*100;
```

1.3 Calculations and Results

When we have 20 nodes, the following plot is generated:



The following is also outputted to the command window:

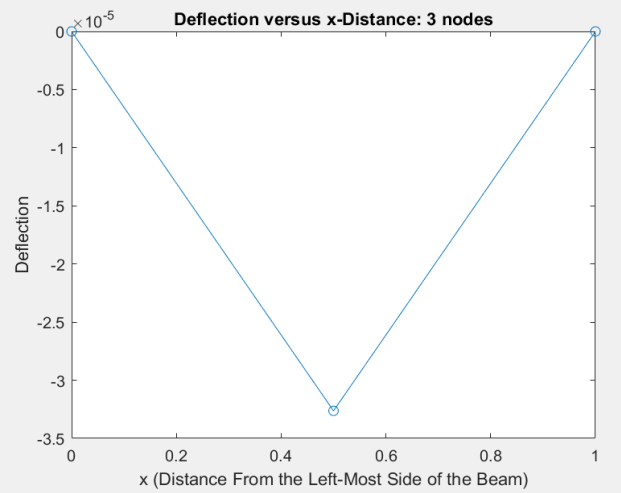
```
This is ymax:   -0.0000324505

Theoretical maximum deflection:   -0.0000323698

Percent Error:    0.2491572058>>
```

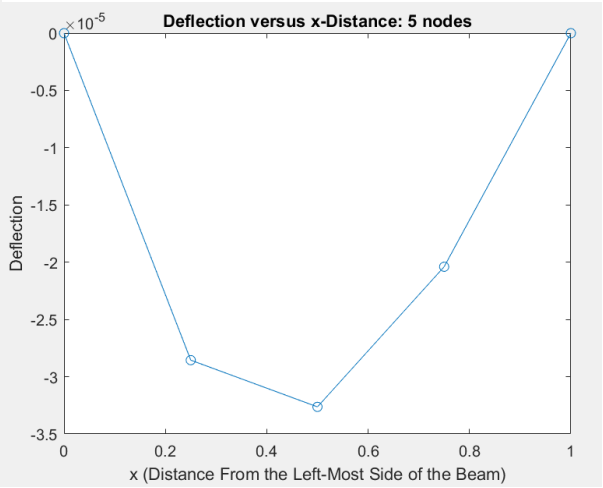
1.4 Discussion

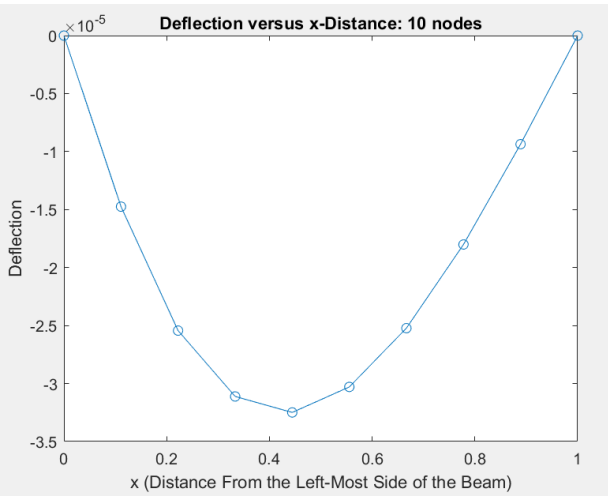
With 20 nodes, the maximum displacement is approximately -3.245*10^-5. This yields a percent error of roughly 2.492*10^-1. A table with the maximum displacements and percent errors with different numbers of nodes is displayed below:

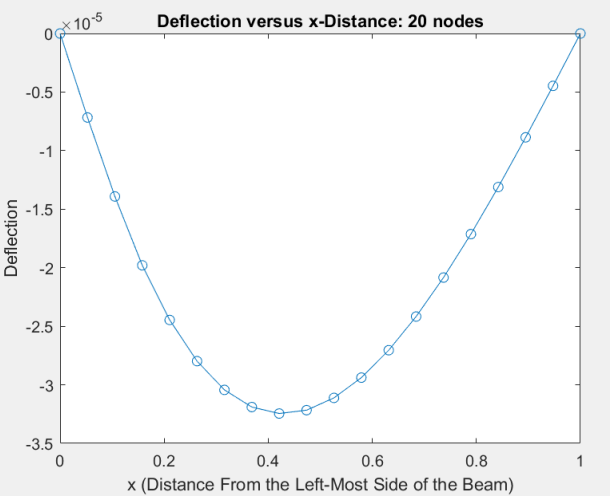
Number of nodes	Maximum Displacement	Theoretical Displacement	Percent Error
n=3	-3.263*10^-5	-3.237*10^-5	8.11*10^-1
n=5	-3.263*10^-5	-3.237*10^-5	8.11*10^-1
n=10	-3.249*10^-5	-3.237*10^-5	3.859*10^-1
n=20	-3.245*10^-5	-3.237*10^-5	2.492*10^-1
n=100	-3.237*10^-5	-3.237*10^-5	2.123*10^-3

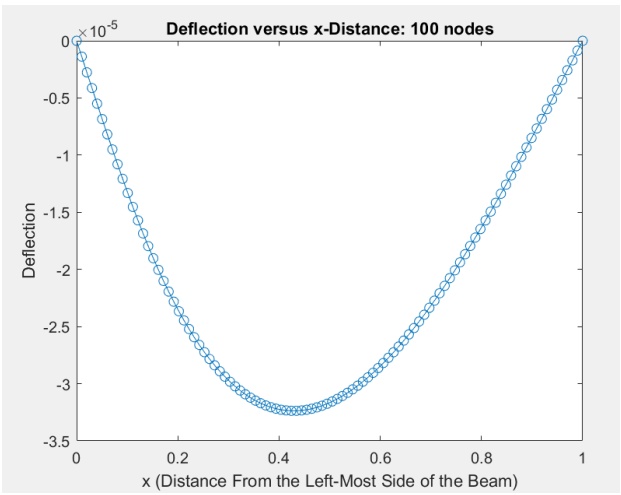
As we can see, there is a general trend: as the number of nodes increases, the percent error generally decreases. We can see why if we examine the plots for the various number of nodes:











As we can see, as we increase the number of discretization points, the distance between the dots on the plots become smaller, simply because there are more points within the same distance. When the distance between the points becomes smaller, our second derivative, or calculations for the right-hand side vector b, become more accurate. When our b vector becomes more accurate, a more accurate deflection vector y is yielded, as we divide A by b to get y.

Moreover, it should be noted that as we increase the number of points, the time it takes for the program to run generally increases:

Number of nodes	Time elapsed (seconds)
n=3	0.374471
n=5	0.379577
n=10	0.376805
n=20	0.373202
n=100	0.396805

This may be because we have for-loops in the main program and the bendingMoment1 function, and more discretization points means that there are more iterations that the for-loops must undergo.

If we change the location of the applied force using the 100-node simulation, we can identify the range of x-positions that always contain the location of maximum displacement:

d (Distance From the Left-End of the Beam)	Location of Maximum Displacement
0.05	0.4242
0.10	0.4242
0.15	0.4242
0.20	0.4343
0.25	0.4444
0.30	0.4545
0.35	0.4647
0.40	0.4747
0.45	0.4848
0.50	0.5051
0.55	0.5152
0.60	0.5253
0.65	0.5455
0.70	0.5556
0.75	0.5556
0.80	0.5657
0.85	0.5758
0.90	0.5758
0.95	0.5758

As we can see, the range of x-values that contain the maximum displacement is 0.4242<= x <= 0.5758. Therefore, maximum deflection will occur between 0.4242 and 0.5758 regardless of where the bending force is applied.

There were many takeaways from this problem. For one, I learned how to better fill arrays with certain elements. For example, in the main function, I had to fill the 'A' array with the values 1,-2, and 1. I wasn't sure how to do this at first, but I quickly realized there was a pattern in which we had to replace the values in the matrix: We start from A(2,1) and replace that element, then move onto the next two columns and replace those before moving down one row and shifting one column to the left. As I continued writing the code, it became easier and easier to recognize patterns like this because of that experience.

Moreover, I realized that if-else statements are very useful in for-loops. In the bendingMoment1 function, I was trying to differentiate between the two moment equations and tried using two separate for-loops, but the indexing was unnecessarily confusing. I didn't really think of putting an if-else statement in the loop

until later on, which greatly simplified the program. Lastly, I also learned how to divide matrices, as I initially tried to do A/b before reading the homework specification and realizing that A\b was required when dividing matrices.

2 The Game of Life

2.1 Introduction

The goal in this problem is to create a model of John Conway's "Game of Life." In the "Game of Life," there is a matrix of cells that are either alive or dead. Each cells survival is dependent on whether its neighbors are alive or dead. We initially start with a matrix that determines whether each cell is alive or dead randomly. From there, we must write a program that calculates which cells are alive or dead and outputs a diagram that displays the matrix. We must iterate through 300 "generations." Then, we must determine how many cells are living as a function of time and implement the chance of a cell prematurely dying. Lastly, we must create a video that shows the simulation iterating through all 300 "generations."

2.2 Model and Methods

This program begins with code that is used to create the video:

```
vidHandle = VideoWriter('game_of_life_video','MPEG-4');
vidHandle.Quality = 100;
vidHandle.FrameRate = 30;
open(vidHandle);
```

Then, we declare the number of rows and the number of columns we are considering with the variables num_rows and num_cols. Then, we create a matrix A that has the dimensions (num_rows, num_cols) using the rand function. I used the rand function because this way, A will contain an even random distribution of numbers in between 0 and 1. Because the number 0 corresponds to being dead and the number 1 corresponds to being alive, we only want 0s and 1s in the matrix, we specifically want 20% of the array to initially be alive and 80% to initially be dead. To do this, I checked if the current cell was less than or equal to 0.20. If so, the cell is alive and contains a 1. Else, if the number is greater than 0.20, the cell is dead and contains a 0. This was done using a for-loop and an if-else statement:

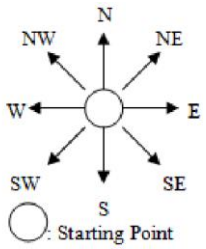
```
for i=1:num_rows
      for j=1:num_cols
            if (A(i,j)<0.20)
                  A(i,j) = 1;
            else
                  A(i,j) = 0;
```

Then, we create a figure that shows the initial A matrix using figure() and imagesc. We call this figure Figure(1) so that it will be displayed separate from other figures we might generate.
Then, we enter our main for-loop that creates and updates the North, South, East, and West indices. The for-loop runs 300 times. Consider the following figure:

j = columns

k = rows

A(1,1)			A(1,j)
	Northwest A(k-1,j-1)	*North* A(k-1,j)	*Northeastt* A(k-1,j+1)
	West A(k,j-1)	*Current Location* A(k,j)	*East* A(k,j+1)
	Southwest A(k+1,j-1)	*South* A(k+1,j)	*Southeast* A(k+1,j+1)
A(k,1)			A(k,j)



To create the indices for the North, East, South, and West directions, I simply followed the model above. If we're moving North you subtract one from the rows, moving South then add one to the rows, if you're moving eest then add one to the columns, and if you're moving west then you subtract one from the columns. A bit of this code is shown below:

```
N = k-1;
E = j+1;
```

Then we check if any of the neighbors are edges or corners. For example, if one of the neighbors is a "North edge," that means that your row-index is k=1. If we pass that value of k into the expression N=k-1, we get N=0, which means that our neighbor is nonexistent and that we must loop around to the south end of the matrix. To do this, we set N equal to num_rows, as num_rows references to the south-most row. Similarly, if we are at the east-most edge, our column-index is num_cols. If we pass this value into E=j+1, then we get E=num_cols+1, but this value does not exist in the matrix and need to loop around to the west-end of the matrix. To do this, we set E equal to 1, which references the first row in the matrix. The exact same logic is applied if we are at a south-edge or a west-edge. A snippet of this code is pasted below:

```
if (N<=0)
        N = num_rows;
    end
    if (E > num_cols)
        E = 1;
    end
```

If we are at a corner, we need to loop around twice. However, we can use the same code we have now and don't need to write special loops for these specific instances. For example, if we are at a Northeast corner, our neighbors on the north and east sides don't exist so we need to loop around. Using the same code, we reset North to num_rows and East to 1 so we are now at the Southwest corner. This Southwest corner will act as the non-existent Northeast corner.

Next, we sum the contents of all of the neighbors:

```
sum1 = A(N,j) + A(k,E) + A(S,j) + A(k,W) + A(N,W) + A(N,E) +
A(S,E) + A(S,W);
```

Then, we use if-statements to see if the cell should live or die based on the rules of the game:

```
if (A(k,j) == 1)
    if (sum1 == 2 || sum1==3)
        A_new(k,j) = 1;
        number1 = rand(1);
```

```
            if (number1 <= 0.01)
                A_new(k,j) = 0;
            end
        else
            A_new(k,j) = 0;
```

The snippet of code above reflects the fact that a living cell only continues to live if it has exactly 2 or three living neighbors. Else it, dies. There is a third if-statement to take into account the 1% chance that a living cell has of prematurely dying. We call a random number using the rand function. If this number is less than or equal to 0.01, then the cell has prematurely died.
Similarly, a dead cell can only be revived if it has exactly 3 living neighbors. Else it stays dead. This code was not pasted below because it was very similar to the code above.
Then, we set the A matrix to the new A-matrix so that we continue our iterations instead of having a static simulation that has identical frames. Next, we calculate the number of living cells in each generation by using the sum function:

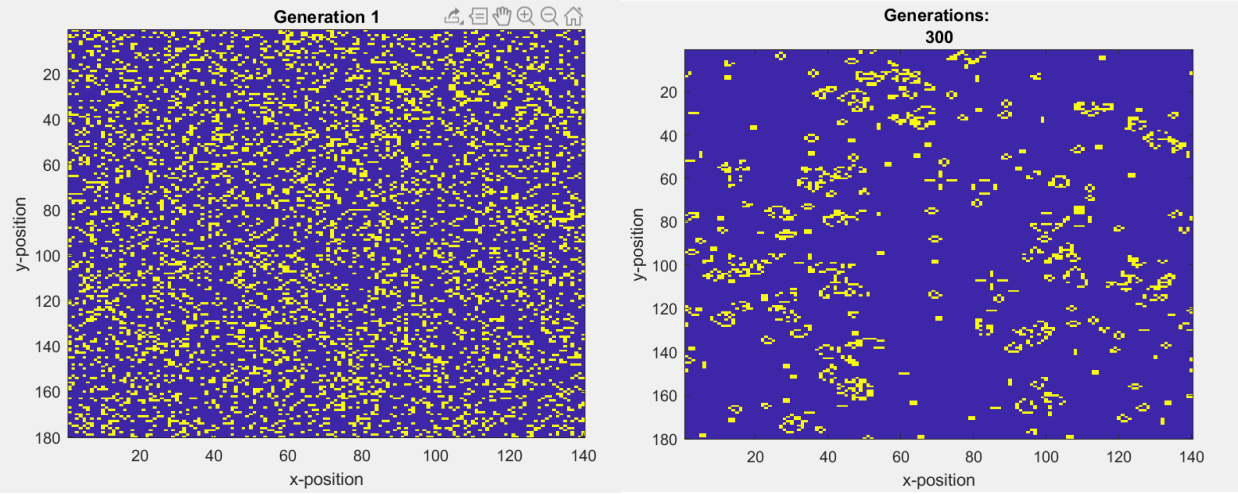
```
living(m) = sum(A(),'all');
```

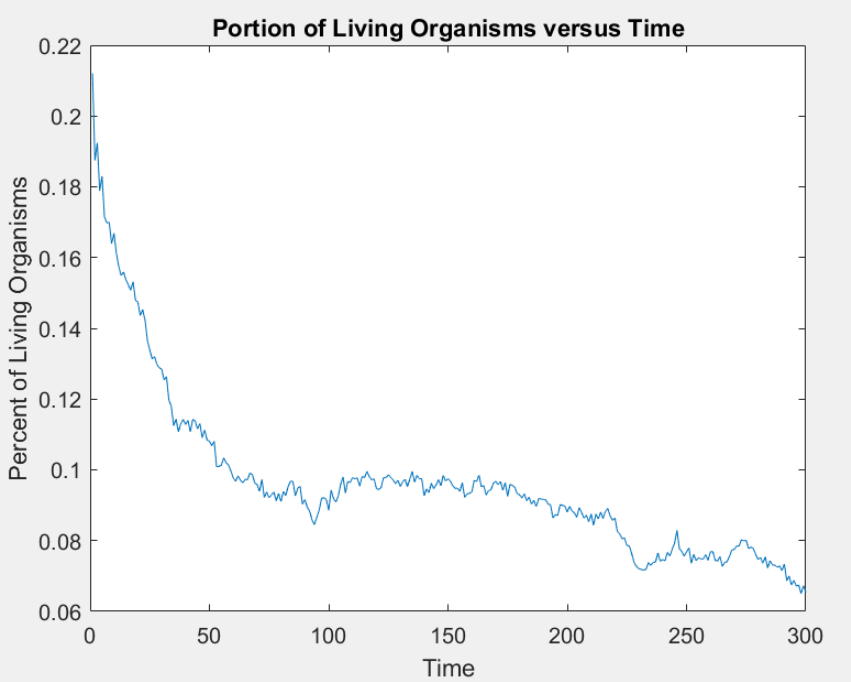
As we iterate from m=1:300, the living() matrix fills up with the number of living cells in each generation. To find the portion or percentage of living cells in each generation, we simply divide this by the number of total elements in the array. We then display the matrix using imagesc and plot the percentage of living cells in each generation versus time using the plot function. We add the appropriate titles and axes labels where applicable.

2.3 Calculations and Results

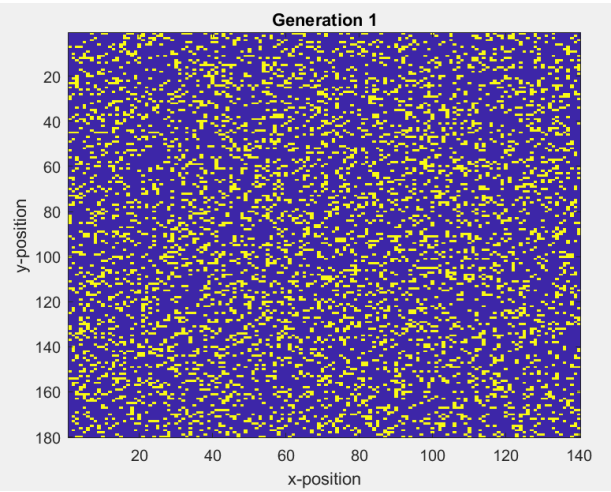
When the program is run, the following plots are generated:

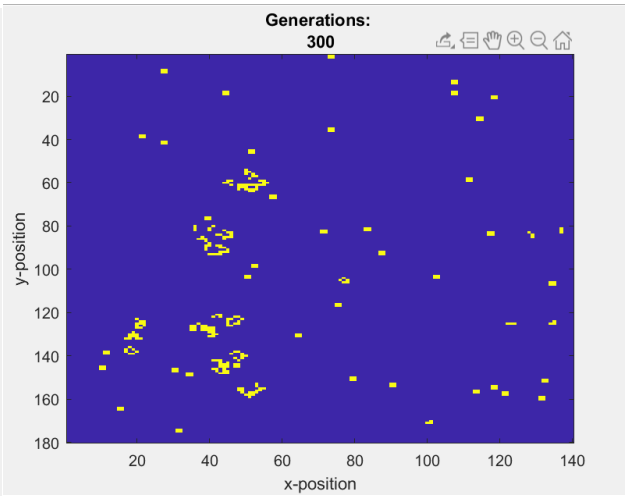
Simulations Without the Chance of Premature Death:

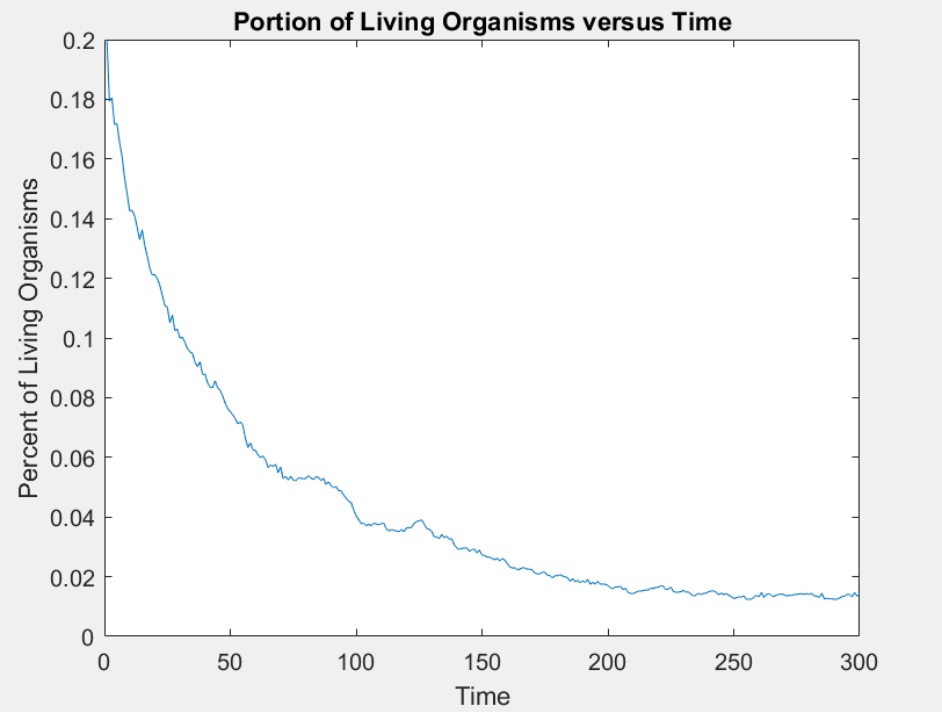




Simulations With the Chance of Premature Death:



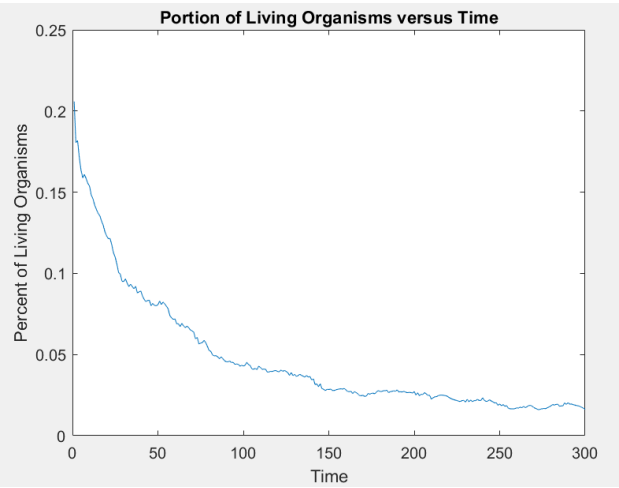


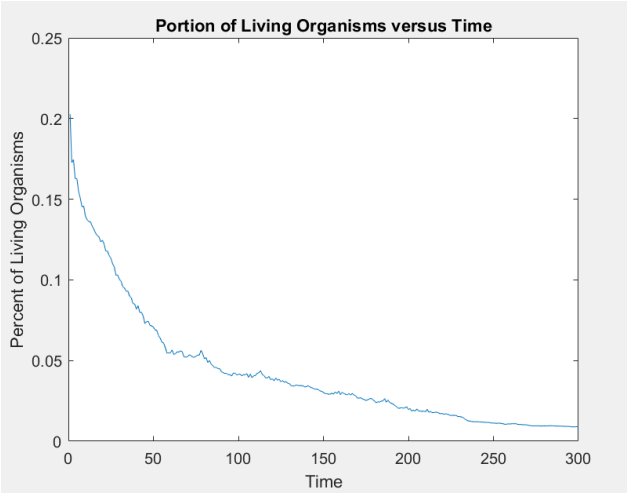


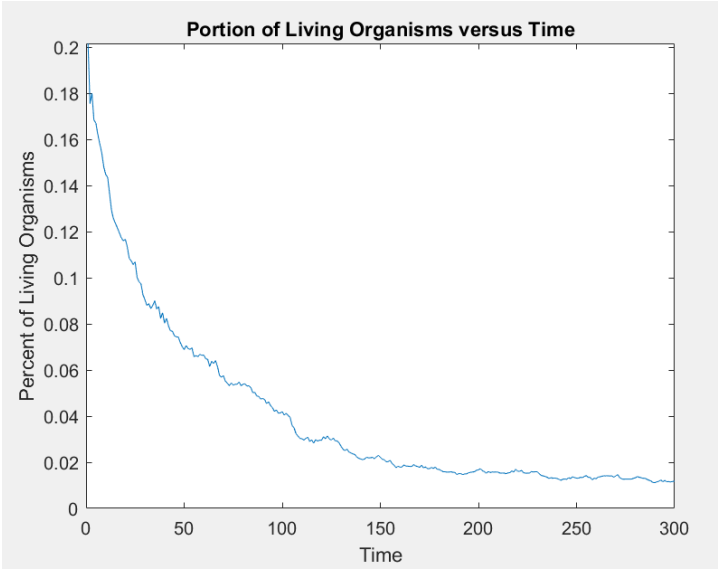
2.4 Discussion

As we can see, the Generation 1 plots that include and don't include premature death start with similar amounts of living cells. This makes sense because both of these matrices were initialized using the rand function, and rand provides arrays of uniformly distributed random numbers. However, there are many differences between these two simulations. For example, the Generation 300 plots that includes premature death has fewer living cells than the plot that doesn't include premature deaths. This makes sense because cells are more likely to die in the plot that includes pre-mature death, which may prompt the cells around it to die as well, forming a chain reaction.

Moreover, the 'Portion of Living Organisms' plot that doesn't include premature deaths never appears to stabilize. The slopes of the graph vary at different ranges, but these slopes continuously decrease. Their slopes don't appear to approach zero. On the other hand, the "Portion of Living Organisms' plot that does include premature deaths does appear to stabilize around 200 generations. To elaborate more on the slopes of these plots, we can learn about the rates of the population decline by analyzing the slopes of these graphs. Both of these simulations begin with a rapid decrease in population from roughly 0 to 50 generations. From 100 to 200 generations, the simulation without premature death has a slower rate of decay, but this rate picks up again from approximately 230-240 generations and then continues to fluctuate from 240 to 300 generations. On the other hand, the simulation with premature death decays at a slower, more constant rate. After running the simulation with premature death multiple times, I found that nearly every plot follows this same pattern: rapid population decrease from generation 0 to generation 50 and more constant decay after that, with an eventual plateau around generation 200 or 250. (This is shown in the plots shown below.) The portion of living cells in the simulation with premature death also fluctuates far less than its counterpart.







(These plots are for the simulation that include pre-mature deaths. They are included to show the general trend explained above.)

Lastly, there were many takeaways from this problem. First, I learned how to simplify a seemingly complex problem. When I first started programming this question, I thought I'd have to create different indices for North, South, East, West, Northwest, Northeast, Southwest, and Southwest. I didn't realize that I could combine aspects of North and East to make Northeast and I overcomplicated my code. After running my program and receiving a bunch of errors, I read through my code and quickly realized that there was a more efficient way to complete this assignment. Furthermore, I also learned how to create a video in MATLAB. At first, I wondered why we couldn't just submit a screen recording of our program running before realizing that using the writeVideo function was a much more efficient, and perhaps simpler, way of creating and saving videos in MATLAB.