Matthew Lacaire
UID: 205288439
CEE/MAE M20
November 23, 2020

Homework 7

Problem 1:

a. Introduction

In problem 1, I implement the Euler-Bernoulli method of solving beam bending, with provided pressure, location of pressure, and geometries of a pipe. I utilize two different functions that calculate the bending moment M(x) and the contrived vector *b*, which is used to solve a system of matrices that yields deflection per x position.

b. Models and Methods

I set up three files: the main code, Bvector.m, and momentcalc.m.

In the main file, I define constants, define *N* (the number of x points used to calculate deflection at), and create a linspace of *N* points from 0 to the given length (discretization points). I define an empty vector of length N to store y values at each x (y values being deflection) I also compute easily-computable combinations of these functions, such as moment of inertia *I*, which is computed as follows:

```
I = pi*(ro^4 - ri^4) / 4;
```

Where ro is outer radius, and ri is internal radius.

In the next process, I pass the x array and the required constants to the *momentcalc* function (N, P, L, d), which returns the M(x) of each x in one array. This directly models the code in equation (2) in the homework. In pseudocode:

```
For each array entry
If x is less than or equal to d (coordinate of force)
>Do some calculation to the array value, return
If x is more than d
>Do some other calculation to the array value, return
```

After now getting an M(x) array, I pass it into the Bvector function. This Bvector function has the same behavior as the momentcalc function (it gets each entry of the array and provides a calculation, storing the result in an array of equal length which is returned).

The b vector originates from a matrix representation of an approximation of equation (1) in the homework. Essentially, we turn the equation below into matrix form, but only after we approximate each y value using the following logic:

$$\Rightarrow y_{k+1} - 2y_k + y_{k-1} = \Delta x^2 M(x)/(EI)$$

We rearrange, and b is the matrix defined as the matrix multiplication of Ay, where y is a column matrix of ordered y values at positions x to be solved. The function returns a vector *b* after running calculations on each member of the Mvals array.

I also populate an A matrix that follows the created construction of Ay=b. I use a double for loop to do that, going row-by-row and column-by-column filling the matrix. This code looks like:

```
for i = 1:N %Populating matrix a per set of rules explained in
discussion.
    for j = 1:N
        if (i == j) && (i == 1 || i == N)
            A(i,j) = 1;
        elseif (i == j)
            A(i,j) = -2;
            A(i,j-1) = 1;
            A(i,j+1) = 1;
        end
    end
end
```

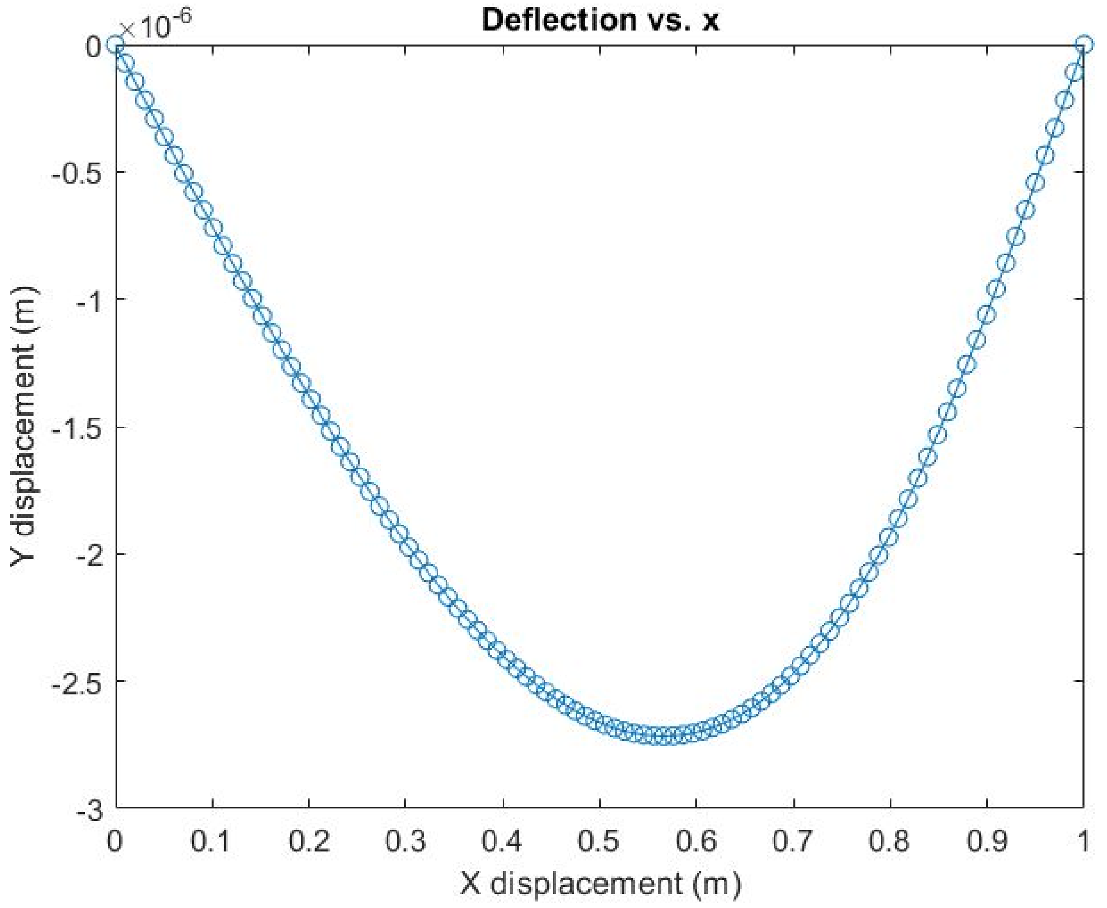
In the critical step, I solve the equation Ay = B by using MATLAB's numerical tools. Since I want y, I would normally do: y = A(inverse)*B. Matlab does this by the syntax y = A\B. I do this, and plot x vs y (position vs. deflection). I also use the min() command to find the minimum value of the deflection (maximum deflection). I use the associated address in the array to find the associated x and I print results. See below:

```
[Value, Location] = min(y); %Finding the value (max deflection)
and associated address.

fprintf('For N = %.0f, we have max displacement %dm, which
occurs at x = %.3fm\n', N, Value, x(Location));
```

Additionally, I use the provided theoretical max deflection equation and compare. See next sections.

c. Calculations and Results

The results seem to check out with those provided initially in the problem statement, and below is a snapshot of the graph that's created:



This is the displayed text:

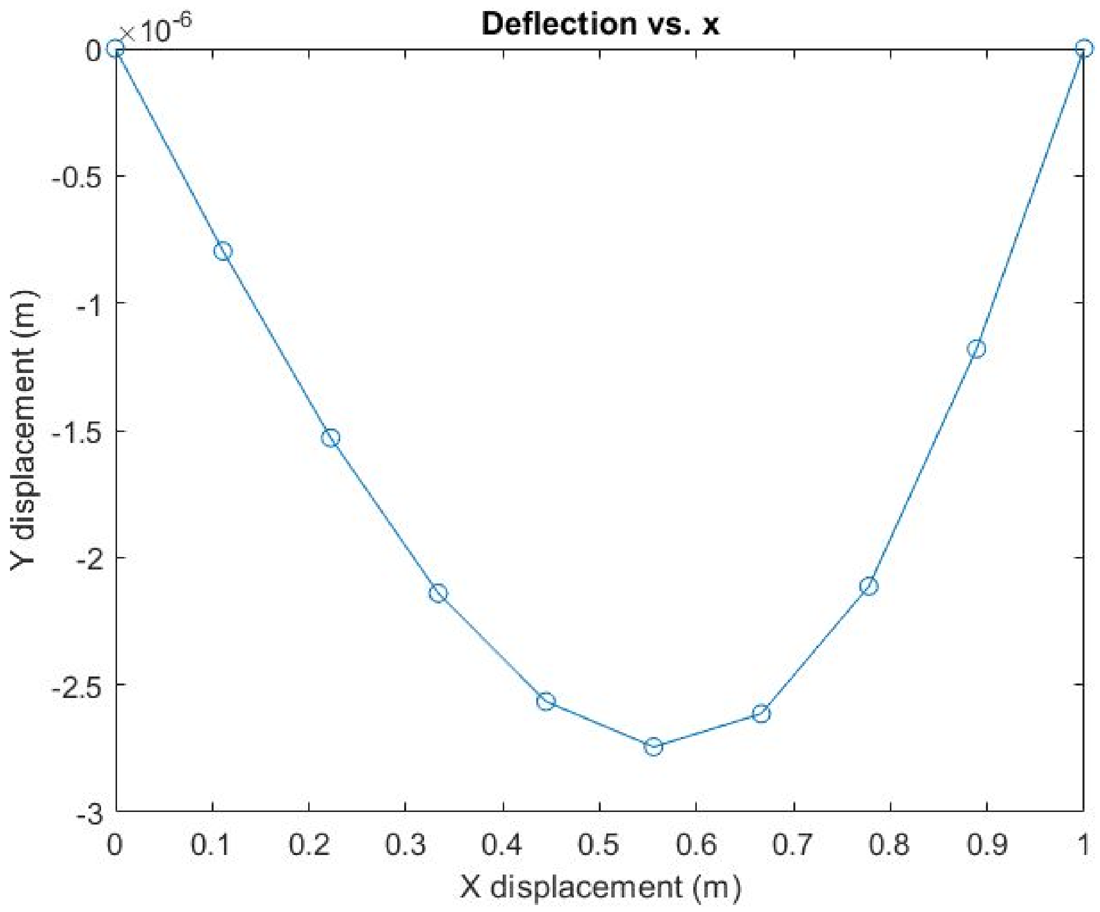
```
For N = 100, we have max displacement -2.716114e-06m, which
occurs at x = 0.566m

Error between theoretical max deflection and measured is
2.272232e-10
```

The associated theoretical solution for these parameters is ~2.7159e-6. Note that the theoretical solution is positive; this is just convention in "deflection". The deflection is, in reality, downwards, but its magnitude is positive (never negative). I deal with this discrepancy by doing abs() of the resulting y, and finding the difference between this and the theoretical. This is printed in the second message.

d. Discussion

In this section, I answer parts d and e of the prompt. Below is a table of the graph for different N values:

N	Error	Compile Time (with *Run and Time* option)
5	9.723131e-08	0.310s
10	2.820038e-08	0.315s
100	2.272e-10	0.352s



Plot for N = 10

As to be expected, the function takes longer to compile for larger N; we simply have larger matrices/vectors that need to be solved and calculated. Interestingly, the time isn't perfectly linear, implying there's probably some necessary resource-consuming process that increases compile time in each case.

Also, as expected, the error decreases with increasing N. With more points, we can more closely reach the point x where the max y occurs. With fewer points, we're farther away, and calculate the *approximate* y at this point only. For example, for N = 5, there's a point at x = 0.5, which is somewhat close to the x value where y is maximized. More points mean we get closer, to reiterate.

For part e, I simply elect to modify the d value in the code and run repeatedly. For d = 0, we get absolutely no deflection as expected. Same is true for d = L. Understanding that for lim d > 0 we get the farthest-left max y point, I plug in d = 0.001, and get x to be 0.424m. Using the same logic and knowing L = 1, I elect to choose d as 0.999, and I get x of max deflection to be 0.576m. Because we're fairly close to the points 0 or L, we can conclude that the x of max deflection *approximately* always lays between these values, 0.424m and 0.576m for L = 1m. I say *approximately* because for d close enough to L or 0, the range will be slightly wider.

Problem 2:

e. Introduction

In problem 2, I simulate Langton's *ant* and visualize it with the imagesc() function. I also record the video automatically using built-in MATLAB capabilities. There is only one script file associated with the code.

f. Models and Methods

The program is actually organized in a fairly simple manner. It starts with declarations, and then iterates a for loop a set number of iteration times. Code that displays results is added, and I also graph black squares out of total squares over time.

I initialize everything by defining a number of rows and columns. Then, I create a grid which is the playing board with these given dimensions. I define a coordx and coordy, an initial (random) coordinate within the constraints. I do that with the following two lines:

```
coordx = ceil(rand()*num_rows); %Creating random initial x
coordinate
coordy = ceil(rand()*num_cols); %Random initial y
```

I define a numIter, or the number of iterations. I create a 1D array of length numIter called totalblackspots. For each loop, this is later filled with the proportion to be graphed.

Importantly, I create a "direction", which is set initially to 1. In my logic, when direction is 1,2,3 or 4, the ant is moving North, East, South, or West, respectively. This is confusing; if MATLAB had an easily-implementable enum feature, I would have used that instead.

I add the line v = VideoWriter('LangtonWalk') to tell the program I am to write a video with this title. I create the object v here.

In the for loop, I take something very important into consideration: the program:

1. Discerns the color of the square the ant is on
2. Reacts by turning according to rules
3. Changes color of square
4. Moves forward one

The order of these steps was key, and after experimenting with different orders, massively different results were generated.

In a little more detail, the pseudocode is as follows:

```
For i= 1:numIter

If grid is white
>Check direction, change direction according to rules.
>Make grid black
>Move one unit in new direction
 If grid is black
>Check direction, change direction according to rules.
>Make grid white
>Move one unit in new direction
```

Compute value of totalblackspots(i), the proportion of black spots to total spots for that given iteration.

Generate, title, display, and save the resulting figure.

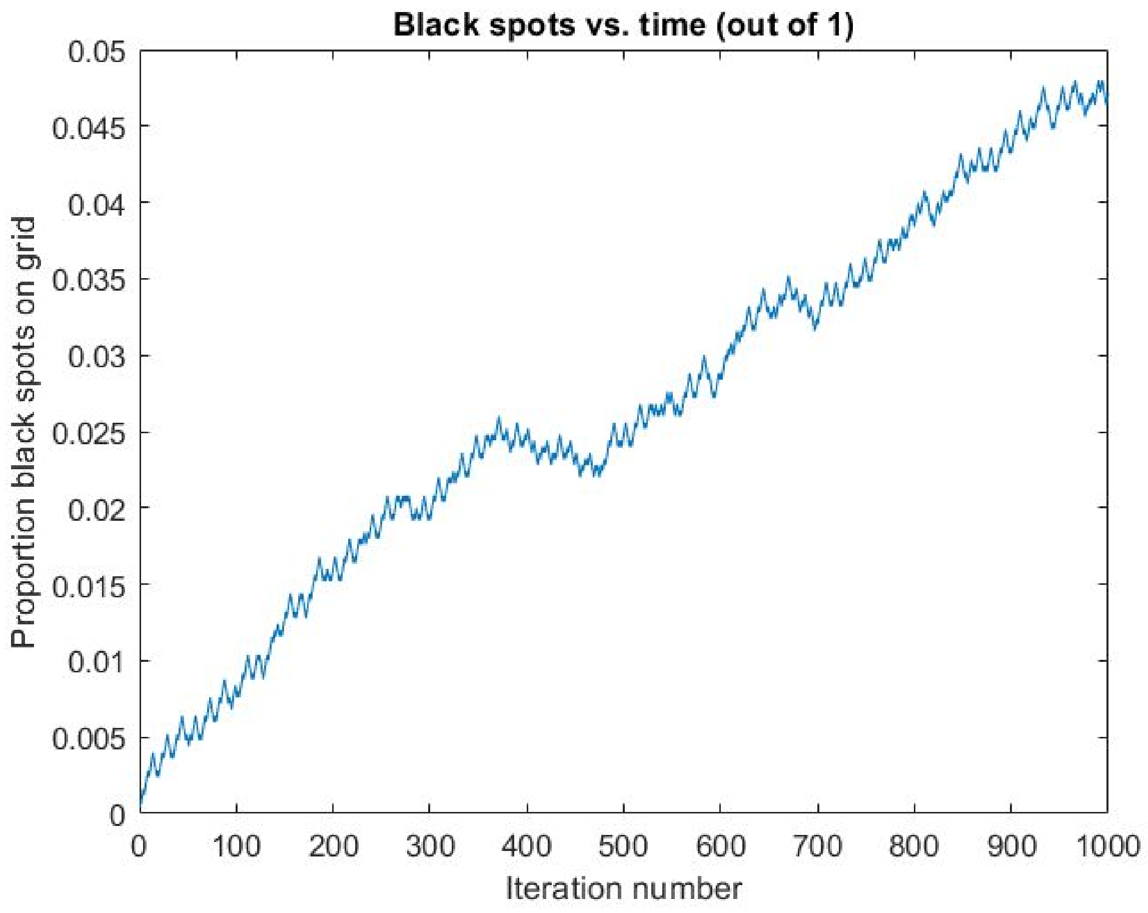
Something to note is that the totalblackspots of i is obtained by summing the entire grid. Because black spots are "1" and white spots are "0", the sum, or sum(sum(grid)), counts the number of black spots exactly. I have to use the sum command twice, because only using sum once results in a 1D vector that contains sums of columns.

Also, within the for loop I have the "wrapping-around" logic. This is short, and is pasted below:

```
    if coordy > 50 %Logic for the "wrapping around" conditions.
        coordy = 1;
    elseif coordy == 0 %If y is zero (top) we wrap around to
bottom.
        coordy = 50;
    elseif coordx > 50 %If we're too far right, we wrap around
to x = 1.
        coordx = 1;
    elseif coordx == 0 %If x is too far left, we wrap around to
right side.
        coordx = 50;
    end
```

g. Calculations and Results

Below is the resulting plot of iteration number versus “totalblackspots”.



Note here that the absolute maximum is 1, considering that this is the number of black spots out of the entire size of the grid. Multiplying by 100 gives the percent of black boxes.

Additionally, the link to the video of the imagesc animation is here: https://youtu.be/ldelf92-0bs. As expected, the graph is able to "wrap around" easily.

h. Discussion

This project truly shows the ability of MATLAB as a convenient script-writing platform. For C++ or other compiling languages (vs. scripted), modeling things with 2D graphics and saving videos is many times more complicated than simply using imagesc(), drawnow(), and other related commands. Also, as mentioned before, the order of the operations (whether to discern first, change color first, etc.) is absolutely critical in getting this configuration. Switching orders (as I did at first) results in strange patterns and/or diagonal streaks across the board. This order is utterly critical and it's a testament to the scrupulousness needed in programming a delicate program.