Duncan Di Mauro
UID: 805163177
CEE/MAE M20
November 19, 2019

Homework 7

1 Euler-Bernoulli Beam Bending

1.1 Introduction

The goal in this problem is to solve a system of equations to study the displacement of a simply-supported steel beam subjected to a single point load. In this problem, we are given the following equations:

$$EI\frac{d^2y}{dx^2} = M(x) \tag{1}$$

$$I = \frac{a^4 - b^4}{12} \tag{2}$$

$$M(x) = -\frac{P(L-d)x}{L} \text{ for } 0 \leq x \leq d \quad \text{and} \quad M(x) = -\frac{Pd(L-x)}{L} \text{ for } d < x \leq L \tag{3}$$

where *E* is the modulus of elasticity, *I* is the moment of inertia of the cross section, *y* is the deflection, *x* is the horizontal location along the bar starting from the leftmost edge, *M* is the bending moment, *a* is the outer width of the bar, *b* is the inner width of the bar, and *L* is the length of the bar. Additionally, there are boundary conditions due to immovable supports on either side of the beam:

$$y|_{x=0} = 0 \quad \text{and} \quad y|_{x=L} = 0 \tag{4}$$

The first goal in this problem is to solve and plot the system of the equations (find the *y* values) by using and performing operations with the **A** matrix for the system (as defined in the lecture slides), along with its right-hand side vector that uses the discretized version of Equation (1) with a given number of evenly-spaced nodes. The maximum displacement should be recorded and compared to its theoretical value, which is defined as follows:

$$y_{theo} = \frac{Pc\left(L^2 - c^2\right)^{1.5}}{9\sqrt{3}EIL} \text{ where } c = \min(d, L-d) \tag{5}$$

Additionally, the absolute percent error represented by the following equation should be calculated:

$$error = \frac{abs(y_{max} - y_{theo})}{abs(y_{theo})} \times 100 \tag{6}$$

Finally, values for *d* and the number of nodes should be altered and their effects on the displacement should be discussed.

1.2 Models and Methods

First, a function `M = MomentCalc(x, P, d, L)` was created that calculates the bending moment using Equation (3). The conditions in Equation (3) were applied using `if` statements. Next, a main script

was written which first assigns all given values to their respective variables. The script also calculates the moment of inertia using Equation (2) and creates a value *dx* which is the distance between any two evenly-spaced nodes (note that the first node starts at x = 0 and the last node is at x = L). Next, the **A** matrix as shown in the lecture slides is created. This is done by setting the upper-left and bottom-right endpoints to 1, and then using a loop for the remainder of the matrix as shown below:

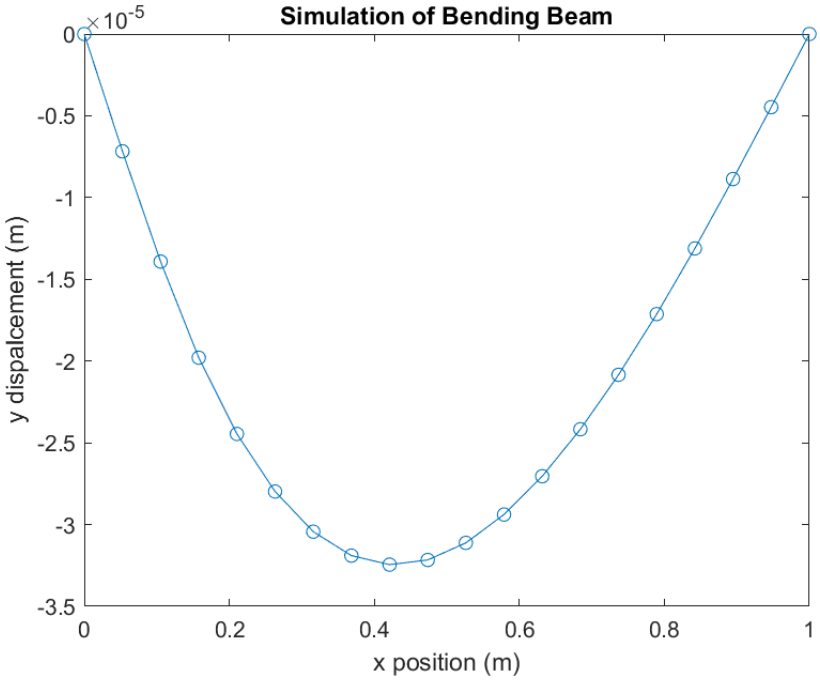
```
for k = 2:maxNodes - 1
    A(k, k-1) = 1;
    A(k, k) = -2;
    A(k, k+1) = 1;
end
```

Next, the b matrix (right-hand side vector) as shown in the lecture slides is created. This is done by first creating a maxNodes by 1 matrix filled with zeroes, and then filling the inner nodes using a loop going from nodes 2 to maxNodes – 1. The values for these nodes are obtained from the right-side of the discretized version of Equation (1) using the central, second-derivative stencil developed in discussion. The loop finds the x-value (the horizontal distance of the node from the left side of the bar) with `x = dx*(nodeNumber - 1)` and finds the value of M(x) by using a call to our moment calculation function.

Then, the matrix containing all y values is calculated with `y = A\b_matrix` (equation given in discussion slides) and the y-values are plotted against their relevant x-values. The maximum displacement value and its minimizer is calculated by applying the `min` function to the *y* matrix. Additionally, the theoretical maximum displacement value is calculated using Equation (5) and the error is calculated using Equation (6).

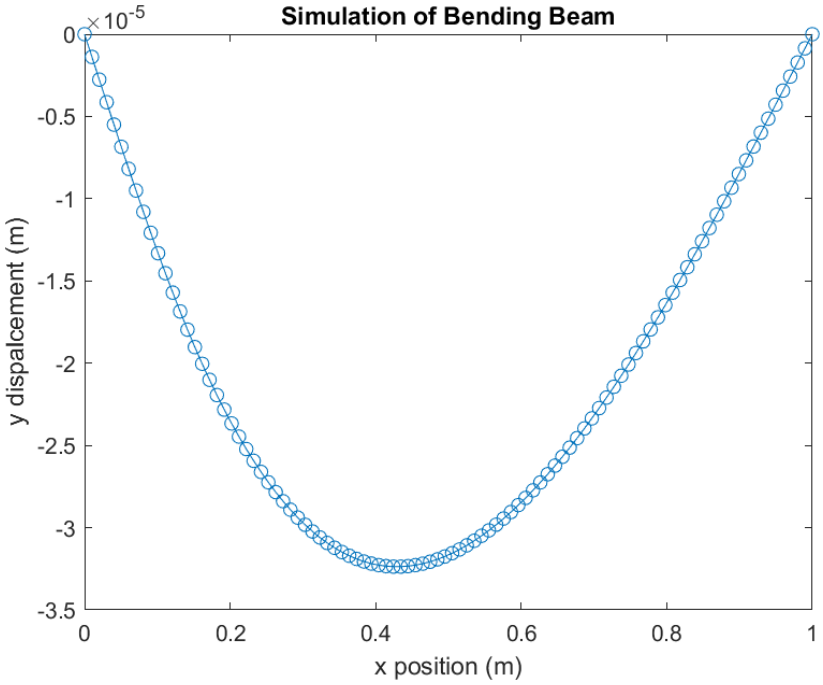
1.3 Calculations and Results

When the main script is executed with 20 nodes and a value *d* = 0.15, the following figure is produced:



During this run of the script, maximum displacement was found to occur at $x = 0.42105$ and $y = -3.245 \times 10^{-5}$. The value for the theoretical maximum displacement was stored in the workspace and was found to be `ytheo = -3.237e0-05`.

When the main script is executed with 100 nodes and a value *d* = 0.15, the following figure is produced:



During this run of the script, maximum displacement was found to occur at $x = 0.4242$ and $y = -3.269 \times 10^{-5}$. The value for the theoretical maximum displacement was stored in the workspace and was found to be `ytheo = -3.237e0-05`.

Additionally, the simulation was run for more values of `maxNodes` with `d = 0.15`. The resulting values of the absolute percent error between displacement values are summarized in the table below:

Value of `maxNodes` (Number of Discretization Points)	Value of `percent_error` (absolute percent error)
3	0.8110
5	0.8110
10	0.3859
20	0.2492
100	0.0014

Furthermore, the simulation was run for various values of `d` with `maxNodes = 100`, and the resulting minimizer values, as shown in the workspace, are summarized in the table below:

Value of `d` (distance of applied force from left of beam)	Minimizer Value (x-value at location of maximum displacement)
0.0001	0.4242

0.1	0.4242
0.15	0.4242
0.2	0.4343
0.3	0.4444
0.4	0.4747
0.5	0.5051
0.6	0.5253
0.7	0.5556
0.8	0.5657
0.85	0.5758
0.9	0.5758
0.9999	0.5758

1.4 Discussion

This problem was very straightforward and involved the relatively new concept of matrices. I learned how to create specific matrices, including the use of `for` loops to fill a matrix with certain values. The first observation in this problem is that the theoretical value `ytheo` stays the same if `d` is constant. As we change the value `maxNodes`, `ytheo` stays the same, but our maximum displacement value changes (this can be seen by the two graphs above). This also means that the error will change with different values of `maxNodes`. The results of the first table show that the error decreases as the number of nodes increases. This makes sense, since the value of `ytheo` occurs at a very specific point, and we are increasing the amount of points at which we are taking y-values. As the number of nodes approaches infinity, the error approaches 0, since our calculated maximum displacement value will get more accurate and therefor closer to the theoretical value. Furthermore, the results of the second table show that there are a finite number of locations on the beam at which the maximum displacement can occur, given that only the location of applied force is changing. As we can see from our table, the range of x-values at which the maximum deflection can occur is $0.4242 \leq x \leq 0.5758$.

2 The Game of Life

2.1 Introduction

The goal in this problem is to simulate the fate of living cells on a grid (matrix) of given dimensions using the rules from Conway's famous "Game of Life." The code should produce a plot of the cells on the grid and should update the plot for every generation that is run (you should be able to see the cells live/die real-time). The first generation should include a 2D array of randomly distributed cells where each cell has a 0.2 probability of being alive and 0.8 probability of being dead. In the matrix, an alive cell should be represented by a "1" while a dead cell should be represented by a "0." Furthermore, the cells should behave as if the grid is wrapped around onto itself (for example, the top of the grid is connected to the bottom and the left of the grid is connected to the right). A plot of the percentage of living cells vs. time (generation number) should be created. Additionally, the script should include a variation to Conway's Game of Life in which any living cell has a 1% chance of prematurely dying during each generation. Finally, a video of the simulation should be created.

2.2 Models and Methods

First, the script assigns all given values to respective variables. Then, a matrix of given dimensions is created which contains randomly distributed numbers from 0 to 100,000 using `randi`. A second matrix representing our cells is then created, which takes the first matrix and converts a number into a 1 if it is originally greater than 80,000. If it is less than 80,000, the number is converted into a zero (this method allows for each cell to have a 20% chance of being alive and 80% chance of being dead). Next, a vector representing time (generation number) is created, along with a vector representing the number of alive cells in a given generation (this is found by taking the `sum` of all values in our cell matrix, since alive cells are represented by 1s and dead cells are represented by 0s). The `CellsAlive` vector will be updated during the main loop in the script. A plot of the initial generation (our initial cell matrix) is then created using `imagesc`.

A `for` loop (the main loop in the script) is created that updates our cell grid for a `MaxGenerations` amount of generations. The loop applies the following rules to each cell in the grid:

1. A living cell with either 2 or 3 living neighbors survives on to the next generation.
2. A living cell with fewer than 2 or more than 3 living neighbors does not survive on to the next generation due to isolation or overcrowding, respectively.
3. A dead cell with exactly 3 live neighbors becomes a living cell in the next generation.
4. Each living cell has a 1% chance of randomly dying

To enforce these rules, we find the amount of living neighbor cells for any given cell, and use `if` statements to update the cell based upon the rules. In order to account for rule 3, we perform separate neighbor checks for alive cells and dead cells. The script checks the number of living neighbors by taking the numerical sum of all cells in its region (the area which includes both the rows and columns on either side of the cell). This is shown below:

```
AliveNeighbors = sum(sum(CurrentGeneration(row-1:row+1,column-
1:column+1)))
```

If the cell is alive, we can subtract 1 from the found value (since the cell is not a neighbor to itself). However, since the grid “wraps around” onto itself, the above code will not work for cells on a boundary of the grid. Therefor, we must restrict the above code for inner-cells, which is simply achieved by:

```
for row = 2:(num_rows-1)
    for column = 2:(num_cols-1)
        if  Current cell is alive/dead … (apply rules)
```

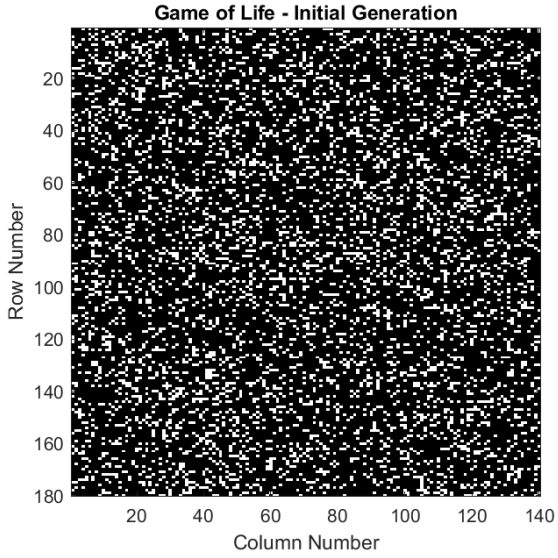
The script creates `for` loops for the other scenarios, including when the cell is in a specific corner or along a specific wall (not a corner), and changes the code to perform a neighbor check on the relevant neighbors that are “wrapped” on the other side(s) of the grid. All these changes (the changing of 0s and 1s) are applied to the matrix `CurrentGeneration`, and are stored in a matrix `NewGeneration`. Once all the neighbor calculations are performed, `CurrentGeneration` is set equal to `NewGeneration`. Then, the updated cell grid is plotted using `imagesc` and the `drawnow` function is used so that the cell-grid can be seen updating real-time. Furthermore, the script sets a variable `frames(GenerationNumber)` equal to `getframe(gcf)`, which stores a picture of the current

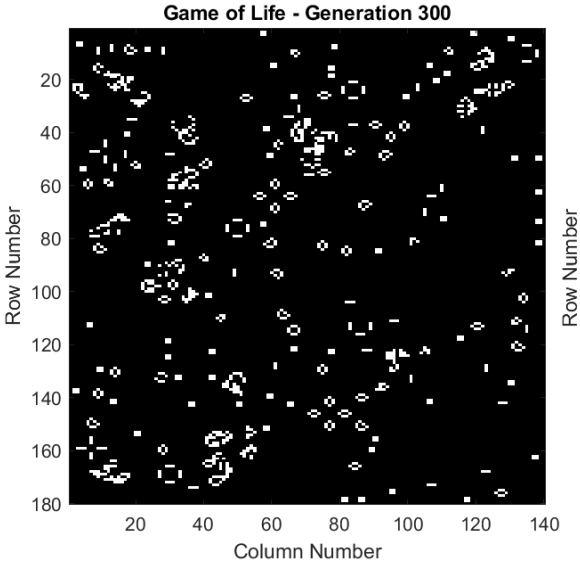
plot (the cell-grid). Afterwards, the `CellsAlive` vector stores the number of total alive cells for each generation. The main loop then ends.

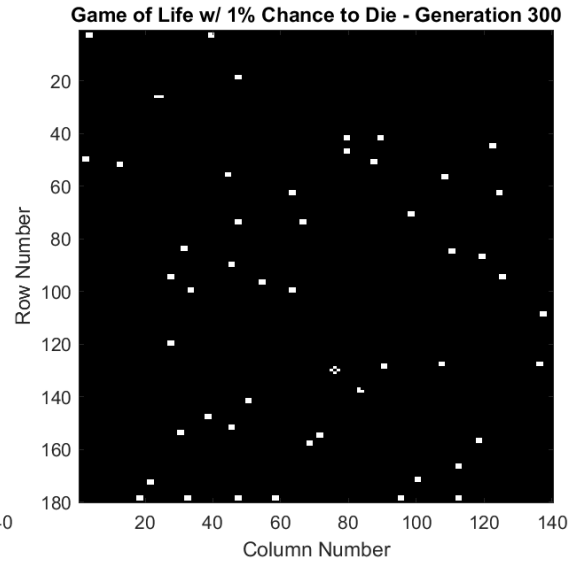
A vector containing the portion of alive cells for each generation is then created via `PortionLiving = CellsAlive/(num_rows*num_cols)`. Next, a figure is created that plots the portion of alive cells vs. the respective generation number. Finally, the video writer creates a writer object `v` with 15fps using the `VideoWriter` and `v.FrameRate` functions. The writer object is then opened, and a `for` loop is created to convert each image stored in `frames` into a frame which is then put into `v` via the `writeVideo` function. Finally, the loop ends and the writer object `v` is closed.

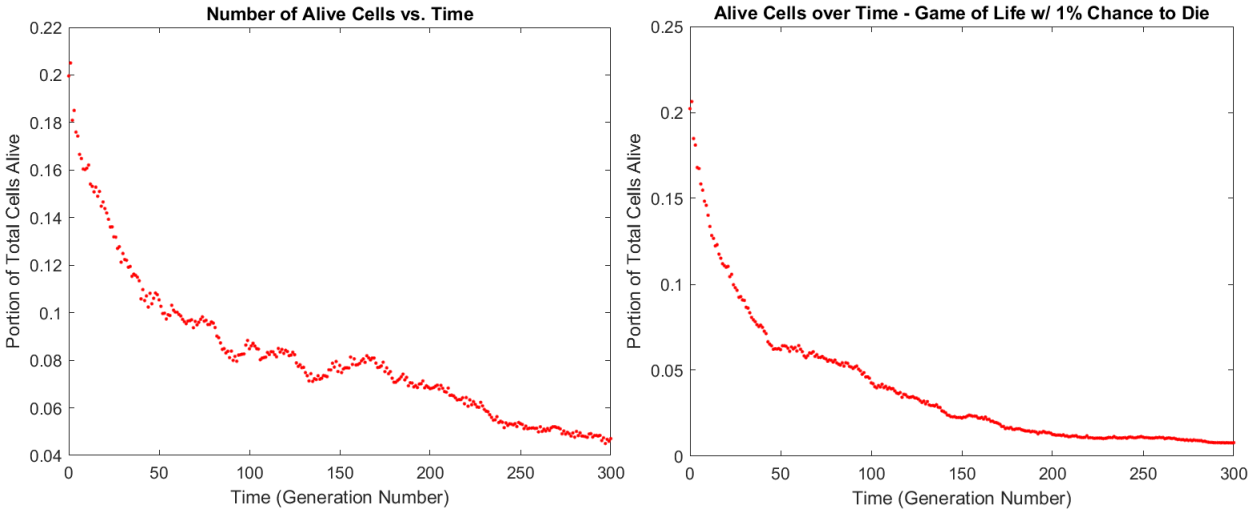
2.3 Calculations and Results

When the script is executed and finishes running, the following figures are created. Note that the script was run two times – once with normal rules and once with each cell having a 1% chance of prematurely dying. The variation with the added rule is denoted in the figure's titles.









2.4 Discussion

This problem involved many new functions including the plotting of a matrix (`imagesc`, `drawnow`) and other functions utilized to create a video. Additionally, the function `randi` was used to randomly distribute numbers in a matrix. In the cell-grid (Game of Life) plots above, we can observe that the final generation for both the regular and variated versions of the script included far less cells (white spots) than their initiation generations (initial generation for the variated version is not shown, but it contains an almost equal number of white dots as the initial generation for the regular version). After many runs of both versions, this was shown to be true for all trials. Additionally, the final generation for the variated Game of Life grid contains far less cells than the final generation for the regular Game of Life grid, which was also found to be true for all trials. This makes sense, as every cell in the variated version has a 1% chance to die, which, when "killed," will often cause other cells around it to die due to rule 2 of the game. These observations are also shown in the scatterplots above, where both trials decrease in portion of living cells, and the variated version has less cells per generation on average than the regular version. Furthermore, it is important to note that, while these comparisons between versions hold true after many trials, there are other factors that change within an individual Game of Life version. For example, in the scatterplot above for the regular version, the plot appears to converge to a number around 0.05. After many trials, the scatterplot continues to appear to trend downwards to a number around 0.05, however it also often approached 0.06. Within each individual trial, there is a lot of variation in terms of the spikes and steepness of the scatterplot at individual points. However, in every trial, the slope is shown to generally decrease over time. Furthermore, there appears to be less variation in the portion of cells once the generation number passes 250 (generally). The plots do not necessarily converge completely to a certain number (become horizontal), however if the amount of generations were to be increased, it is likely that the plot would, as the slope continues to decrease. The scatterplot involving the variated Game of Life has many of the same properties as the regular, such as a consistent decrease in slope and variation as time increases. However, this version appears to have much less variation that the regular version overall, and the slope appears to decrease much quicker (and therefor the graph stabilizes/converges much quicker). After many trials, this stabilization point for the variated version appeared to start roughly at around generation 160 and converge to a value of around 0.01.