Zubin Mishra
UID: 604644805
CEE/MAE M20
May 23, 2017

**Homework 7**

# 1 The Game of Life

## 1.1 Introduction

The goal in this problem is to develop a program that simulates the fate of living cells using the rules from mathematician John Conway's "Game of Life." This is accomplished using a 2D array as an arena and using the rand function to populate it. To display the cells and how they change, the MATLAB functions imagesc and drawnow are used. It will be discussed what cell patterns commonly occur, if there is a trend in the total number of living cells over time, and how different birth/death rules affect the simulation.

## 1.2 Model and Methods

The script first establishes the size of the grid and creates the array to hold the number of living cells in each generation. The script then creates the grid and populates it with approximately 10% living cells:

```
A = zeros(num_rows, num_cols);
for i = 1:num_rows
    for j = 1:num_cols
        if (ceil(10*rand) == 1)
            A(i,j) = 1;
        end
    end
end
```

This initial distribution (Generation 0) is then drawn:

```
figure
imagesc(A);
title('Generation 0');
```

The script then enters a for loop that progresses through each generation. Within the for loop a blank 2D array is created to store the next generation. Then there are a further two nested for loops to iterate through each cell of the grid. Within these nested for loops, first north, south, east, and west indices are found so as to allow for wrap-around. The number of neighboring living cells is then calculated using these indices, and based off the result, the status of the cell in the next generation is determined.

```
sum_ = A(N, col) + A(S, col) + A(row, E) + A(row, W)...
                + A(N, E) + A(N, W) + A(S, E) + A(S, W);
if A(row, col) == 1 && ~(sum_ < 2 || sum_ > 3)
    A_new(row, col) = 1;
elseif A(row,col) == 0 && sum_ == 3
    A_new(row, col) = 1;
```

The grid is then updated and drawn:

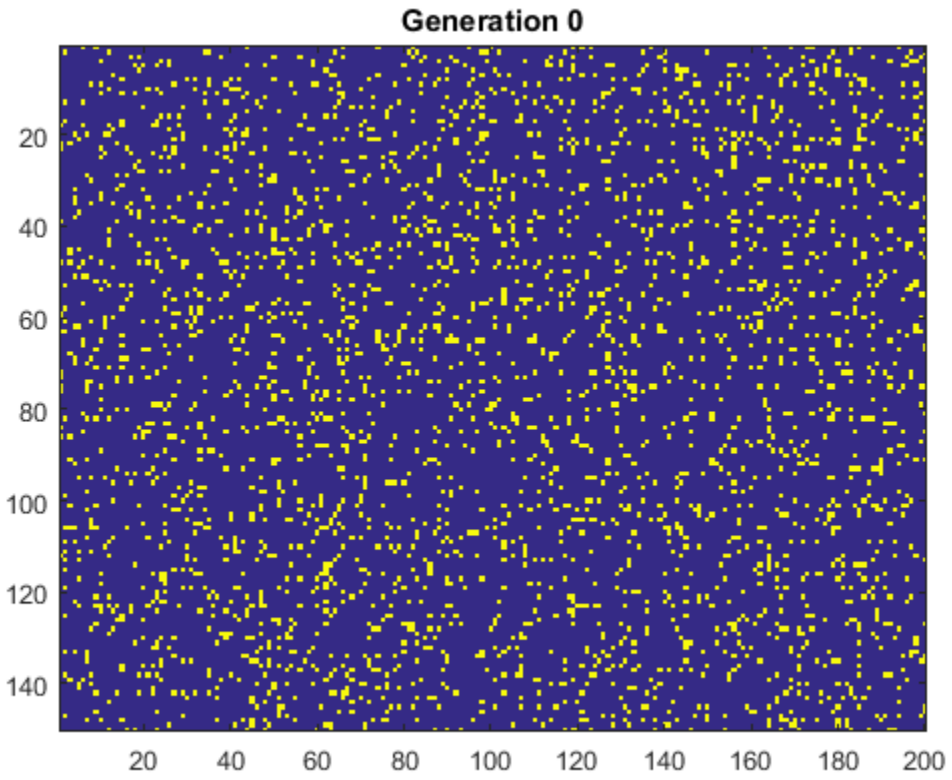
```
A = A_new;
imagesc(A);
title(['Generation ', num2str(k)]);
drawnow;
```

The total number of living cells is also stored. After all the generations have been calculated and displayed, and the outermost for loop ends, the total number of living cells is plotted against time (generations).

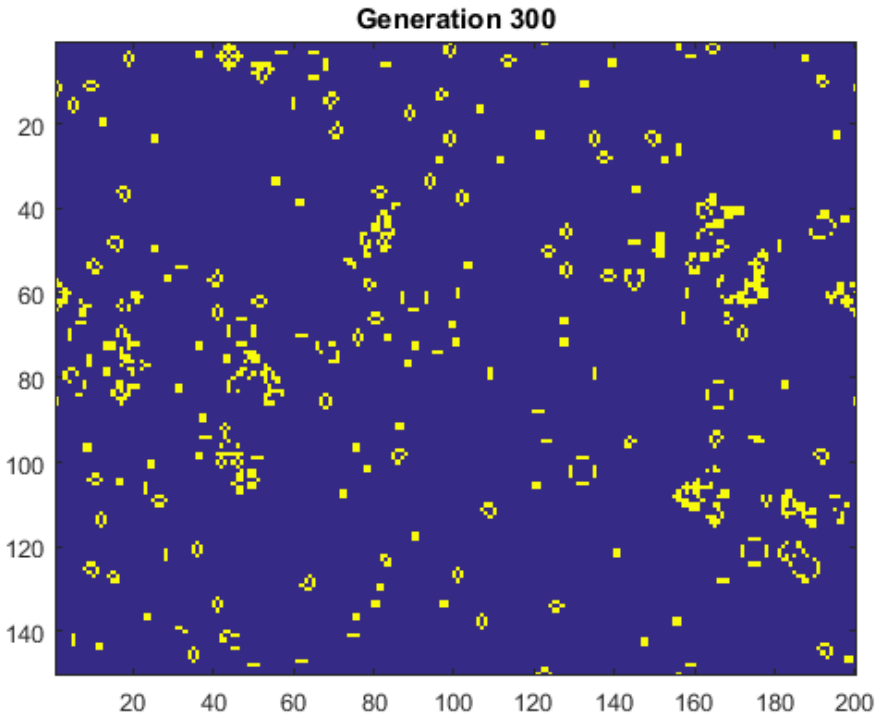
## 1.3 Results and Calculations

When the script is run with default parameters, the following output is produced (different each run):

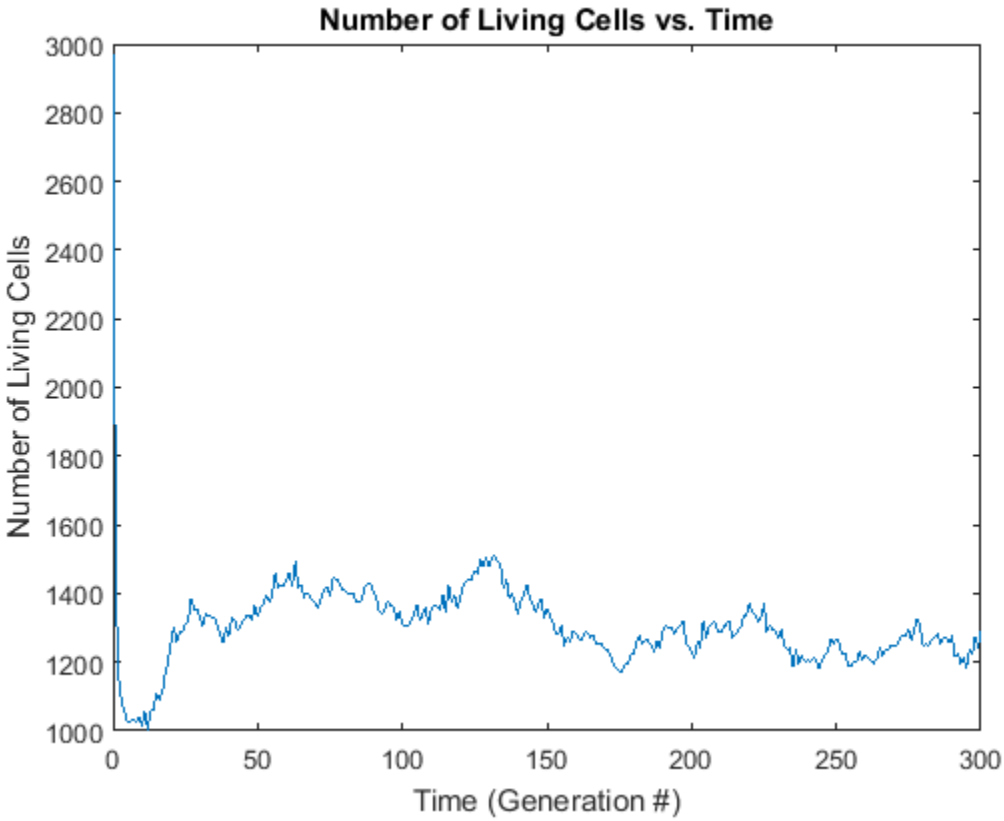
Initial distribution:



Final distribution:

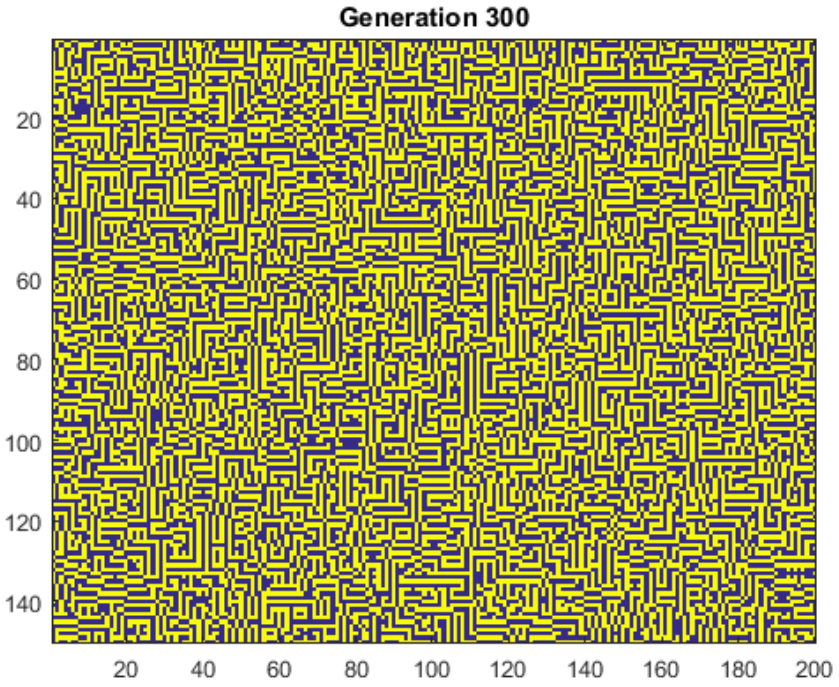


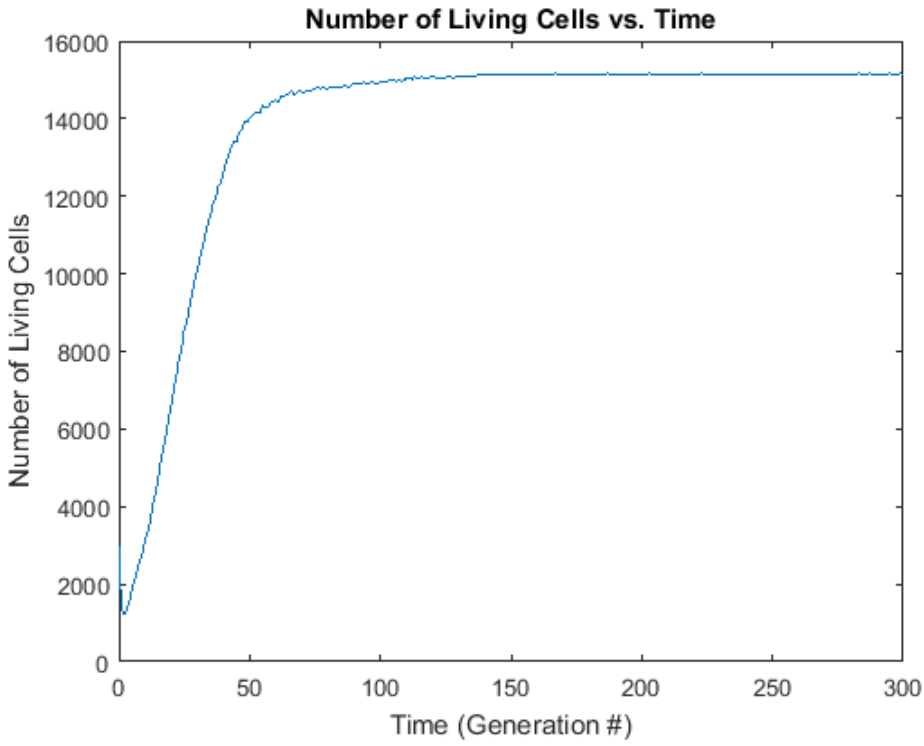

Number of living cells vs. time:



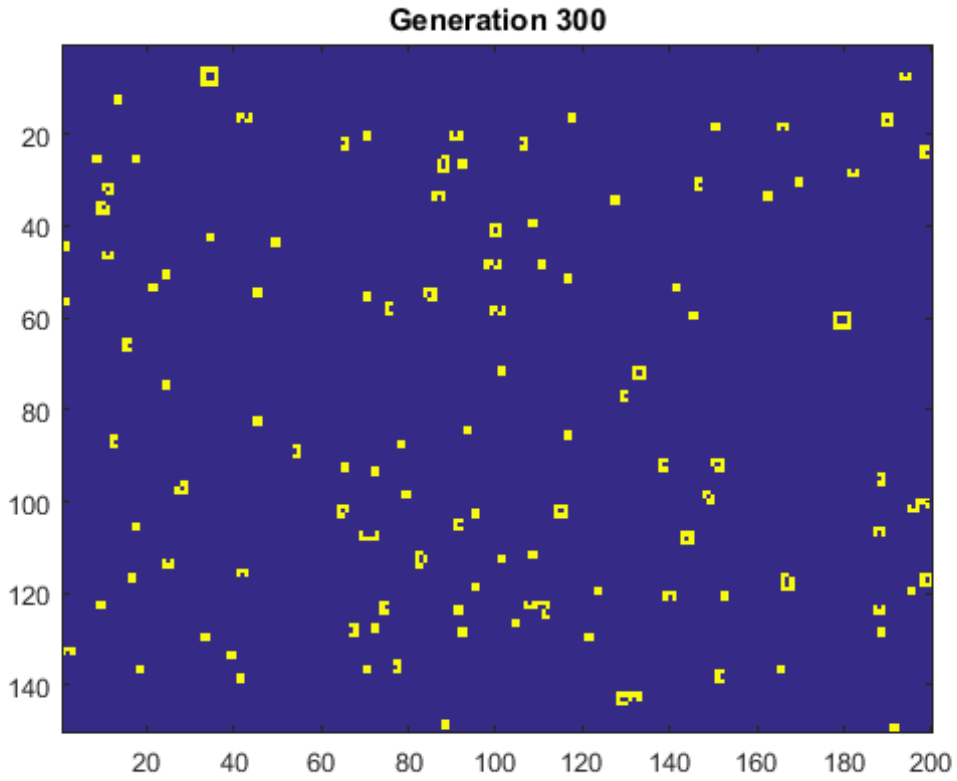
Each run has similar results.
The following output is produced when changing the birth/death conditions. Note that initial distributions are not included in the following results because they remain very similar throughout to the distribution above.

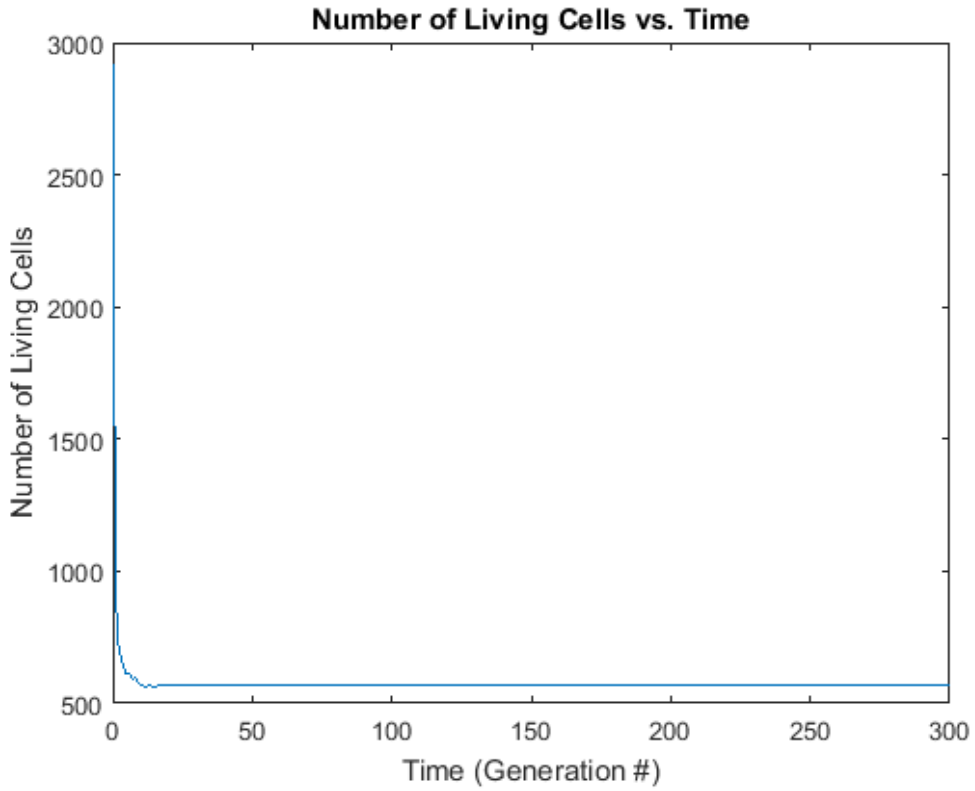
When allowing cells to survive with 4 neighboring cells:



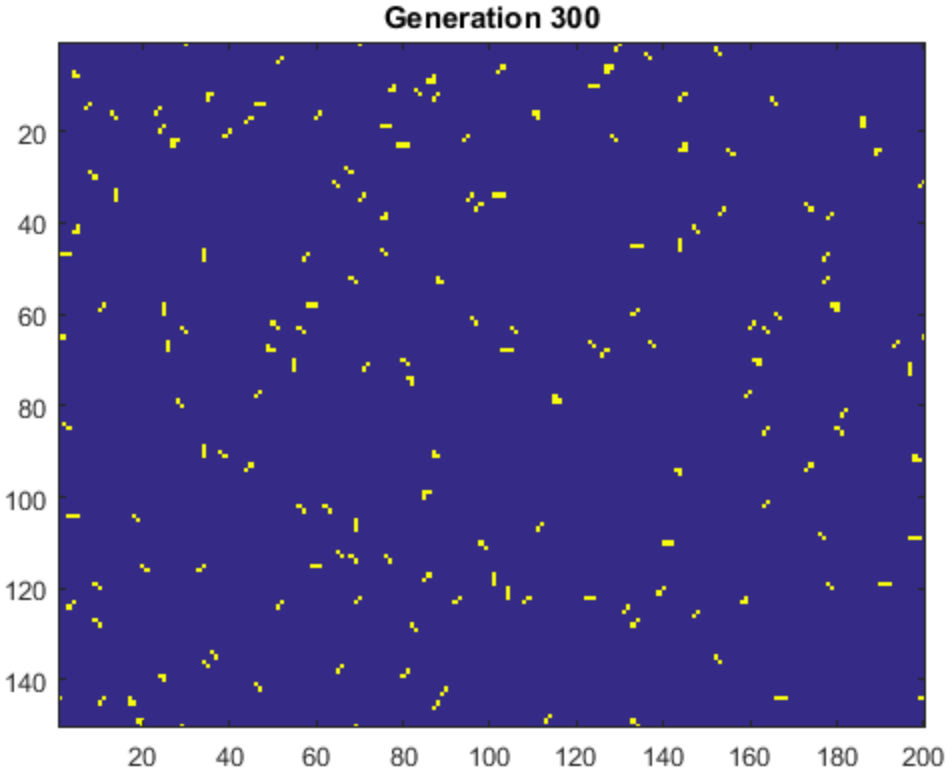



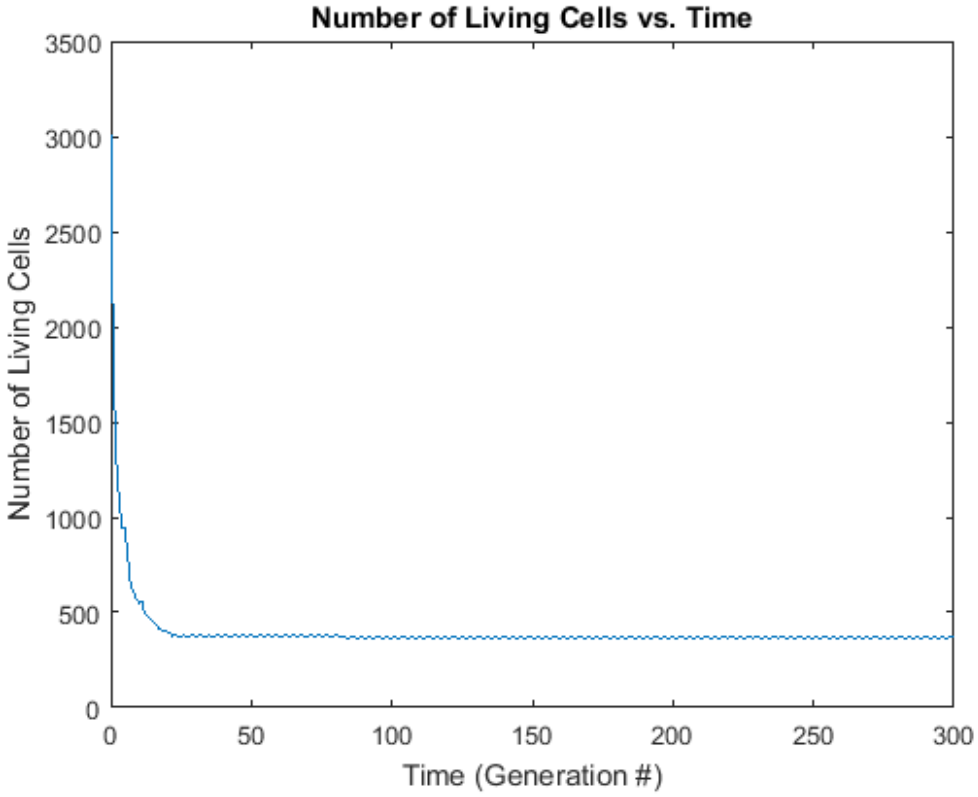
When counting diagonal cells as 0.7 instead of 1 and rounding the resulting sum:



When counting directly adjacent cells as 1.4 instead of 1 and rounding the resulting sum:



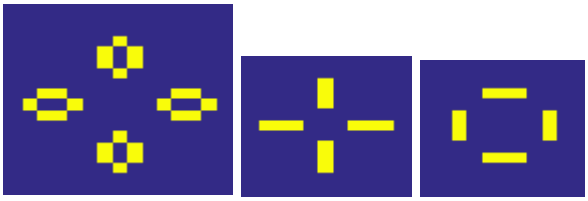



## 1.4 Discussion

There were a number of commonly occurring cell patterns that persisted across generations. One type could be considered as loops of varying size, coming in the following variations:

There were also groups of 3 cells that would alternate between horizontal and vertical each generation:

It was also common to see these formations of cells in a square cross formation:



The total number of living cells followed a fairly consistent trend across runs of the script. The population would start off at around 3000 and then quickly dip. The population then stabilized and would typically range around 1000-1500.

When changing the rules so that cells could survive with 4 neighbors, the final distribution ended up looking almost maze-like, with the grid being approximately half living cells, half dead cells. The number of living cells appears to dip shortly, and then climb quickly, and stabilize at around 15000.

When changing the rules so that each diagonal neighbor only counted for 0.7 instead of 1, and then rounding the sum (to get an integer), the final distribution ended up being sparsely populated, with small ‘blocks’ of cells scattered around. The number of living cells started at around 3000 and then quickly dipped and stabilized at a little more than 500.

When changing the rules so that each directly adjacent neighbor counted for 1.4 instead of 1, and then rounding the sum (to get an integer), the final distribution ended up being sparsely populated once again. This time, there were several more diagonal formations of cells than with other rulesets. The number of living cells once again started at about 3000 and then quickly dipped, this time settling at less than 500.

## 2 Euler-Bernoulli Beam Bending

### 2.1 Introduction

The goal in this problem is to develop a program that solves a system of equations to study the displacement of a simply-supported aluminum beam subjected to a single point load. This is accomplished using a discretized second derivative governing equation. A matrix and a vector are built using this discretized equation and displacement is solved for using the '\' operator in MATLAB. It will be discussed how the error in displacement changes as the number of discretization points increases, and it will be discussed how far to the left and right the location of the maximum displacement can be shifted.

### 2.2 Model and Methods

The script solves for the displacement using the following governing equation:

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

When discretized, this becomes:

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

Boundary conditions are established as follows:

$$y\big|_{x=0} = 0 \text{ and } y\big|_{x=L} = 0$$

In order to solve for the displacements, a matrix-vector multiplication relationship is established,

$$\mathbf{A}y = b$$

where the matrix A contains coefficients, the vector y contains the displacement, and the vector b contains the right-hand side of the discretized governing equation. Knowing A and b allows for the calculation of y.

The script begins by establishing the constants required. It then establishes the array of nodes (x-values), calculates the distance between nodes, and establishes the array to hold the bending moment values. The bending moment values are then calculated following the piecewise function:

$$M(x) = \begin{cases} \dfrac{-P(L-d)x}{L} & \text{for } 0 \le x \le d \\ \dfrac{-P(L-x)d}{L} & \text{for } d < x \le L \end{cases}$$

The moment of inertia of the cross section is calculated following:

$$I = \frac{1}{12}bh^3$$

The modulus of elasticity is also established.

The matrix A is then built:

```
A = zeros(node_num);
A(1,1) = 1;
A(node_num, node_num) = 1;
for i = 2:node_num-1
    A(i,i-1) = 1;
    A(i, i) = -2;
    A(i, i+1) = 1;
end
```

Then the vector b is calculated:

```
B = zeros(node_num, 1);
for i = 2:node_num-1
    B(i) = dx^2*M(i)/(E*I);
end
```

Using MATLAB's '\' operator, y can now be calculated:

```
y = A\B;
```

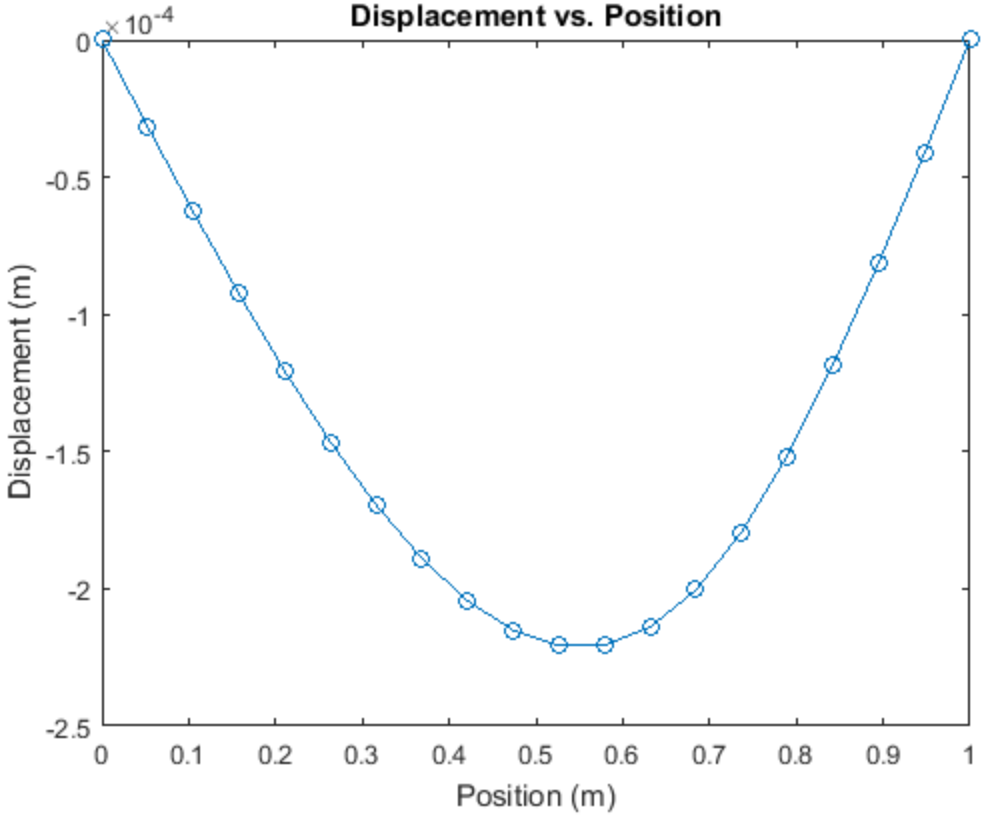
The displacement is then plotted. After that, the theoretical maximum displacement is calculated following:

$$y_{\max} = \frac{Pc(L^2 - c^2)^{1.5}}{9\sqrt{3}EIL} \quad \text{where } c = \min(d, L-d)$$

Finally, the error from max displacement is calculated.

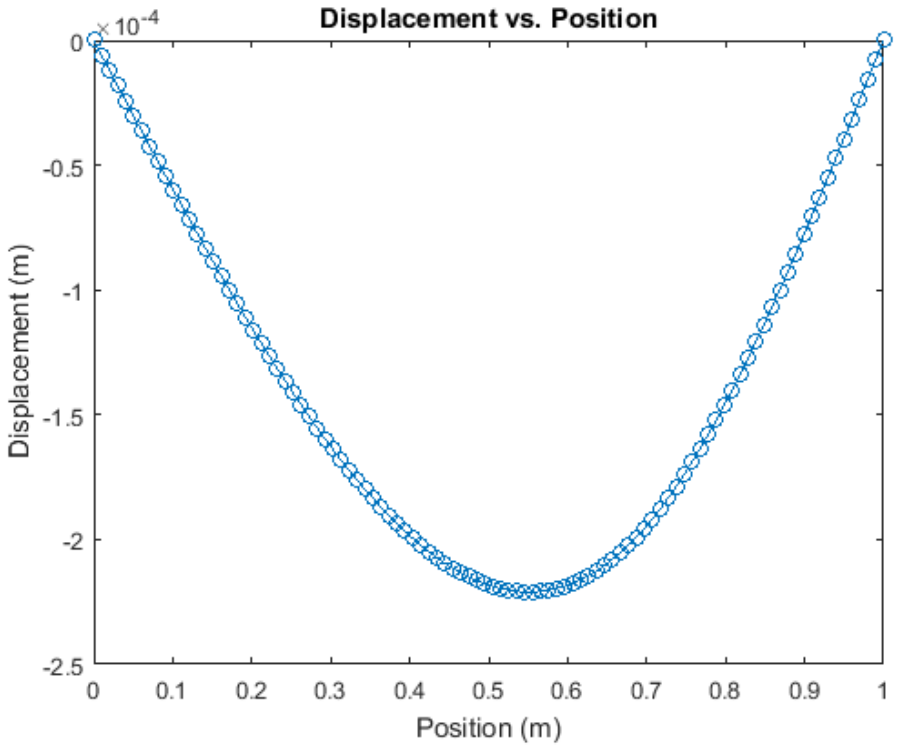
## 2.3 Results and Calculations

When the script is run with default parameters, the following output is produced:



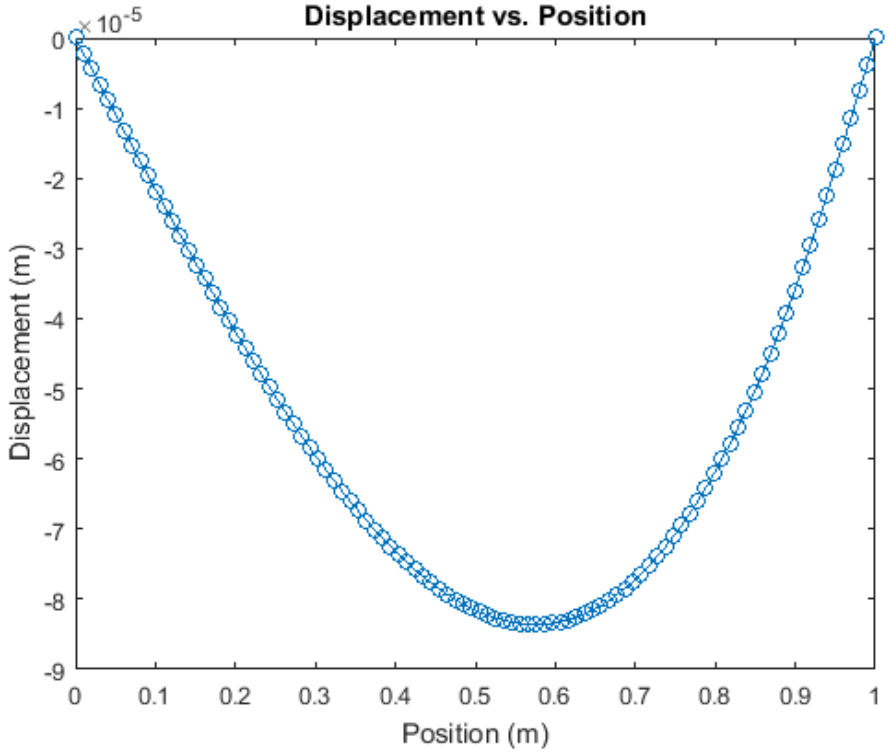

Max displacement = -2.2071e-4 m at 0.5263 m with an error of 2.7216e-7 m

When the number of nodes is increased to 100:



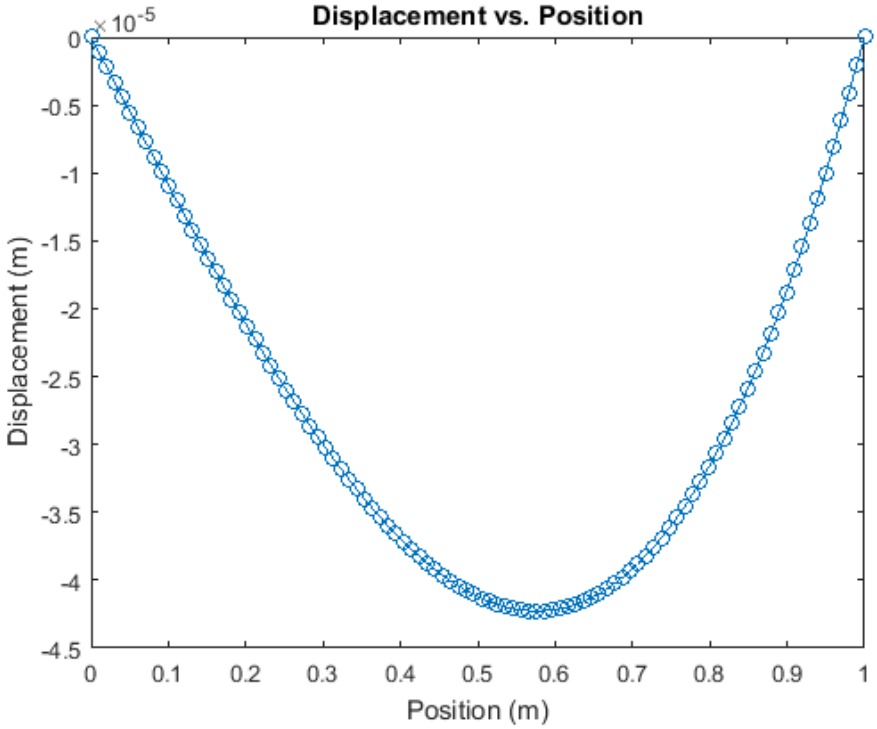

Max displacement = -2.2097e-4 m at 0.5556 m with an error of 1.1258e-8 m

When the force is applied at 0.9m:



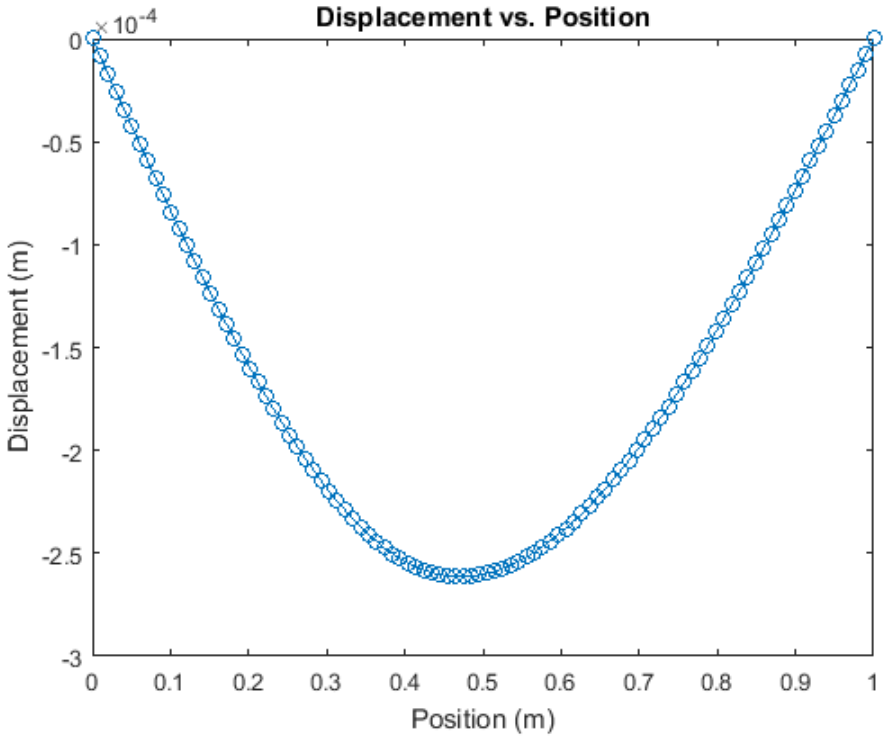

Max displacement = -8.3594e-05 m at 0.5758 m with an error of 8.9044e-9 m

When the force is applied at 0.95m:



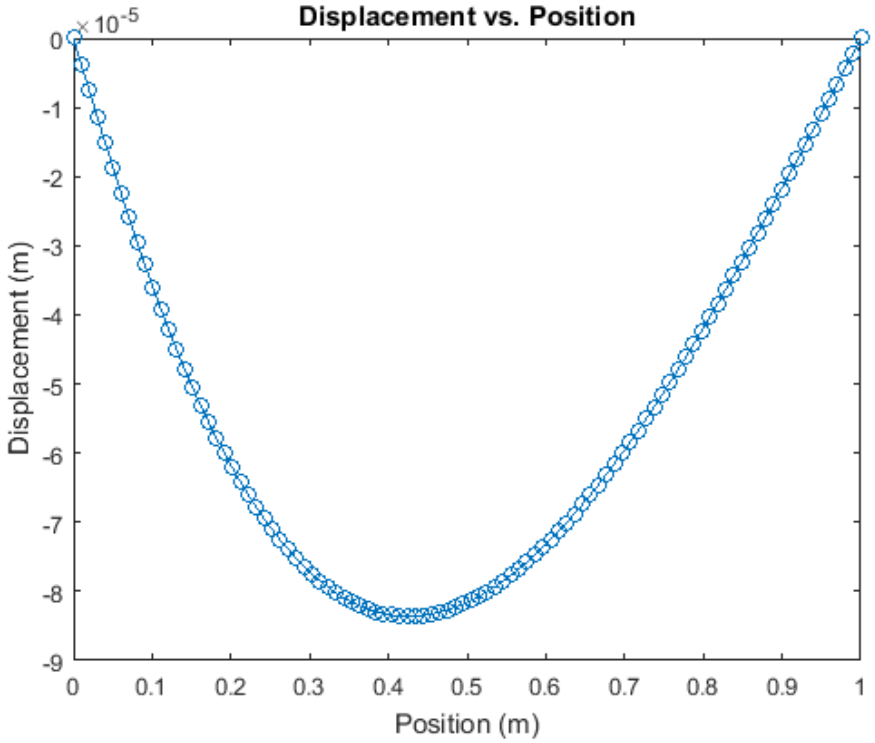

Max displacement = -4.2274e-05 m at 0.5758 m with an error of 5.4641e-09 m

When the force is applied at 0.4m:



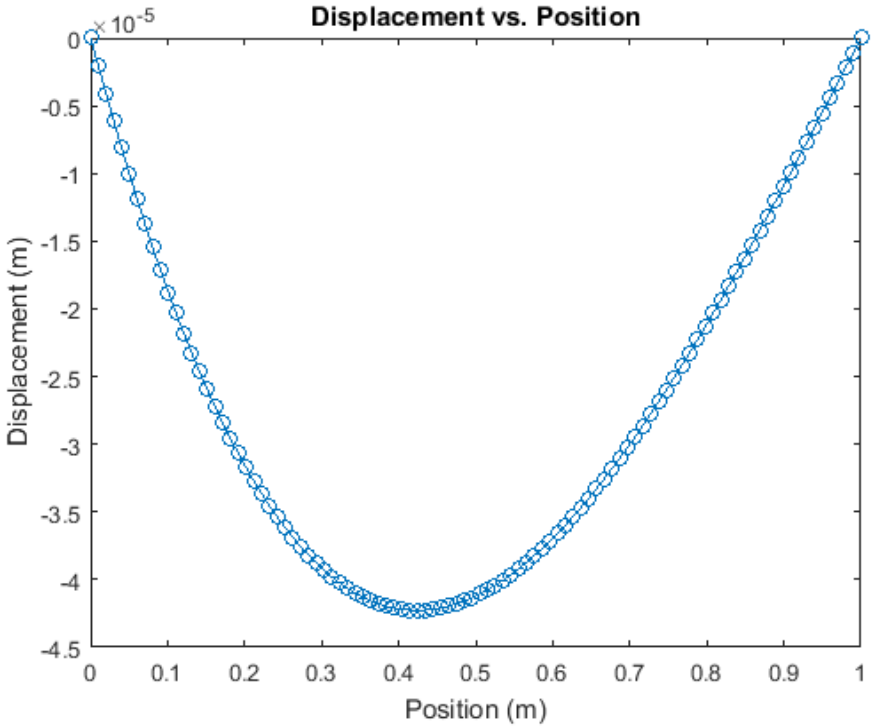

Max displacement = -2.6130e-04 m at 0. 0.4747 m with an error of 7.9253e-09 m

When the force is applied at 0.1m:




Max displacement = -8.3594e-05 m at 0.4242 m with an error of 8.9044e-09 m

When the force is applied at 0.05m:




Max displacement = -4.2274e-05 m at 0.4242 m with an error of 5.4641e-09 m

## 2.4 Discussion

When increasing the number of discretization points to 100, the error in the maximum displacement was reduced by a factor of approximately 24. This is very close to what is expected, considering that the error is $O(\Delta x^2)$ and $\Delta x$ was reduced by factor of approximately 5.

When changing the location of the applied force, the farthest right the location of maximum displacement moved was 0.5758 m. The farthest left the location of maximum displacement moved was 0.4242 m.