Jeffrey Ding
UID: 104928991
CEE M20
May 19th, 2018

Homework 7

1. The Game of Life

1.1 Introduction

The goal of this problem is to simulate the fate of living cells using the rules from mathematician John Conway's Game of Life

1.2 Model and Methods

The script begins by initializing a 2D array called grid and populating the cells with zeros:

```
num_rows = 150;
num_cols = 200;
grid = zeros(num_rows, num_cols);
```

The script then creates 1D array called livingCells that will track how many cells are alive at each generation:

```
livingCells = zeros(1, 300);
livingCellsIndex = 1;
aliveCells = 0;
```

Then, the script runs a nested for loop that generates living cells into the grid with a 1/10 chance of creation:

```
for row = 1:num_rows
   for col = 1:num_cols
      if floor(rand()*10) == 0
         grid(row, col) = 1;
         aliveCells = aliveCells + 1;
      end
   end
end
livingCells(livingCellsIndex) = aliveCells;
```

Then, the script runs a while loop 300 times to simulate 300 generations:

```
generations = 1;
while generations < 300
        ….
end
```

Within the while loop, the script resets multiple variables and generates a newGrid that will be a modified copy of the previous generation's grid:

```
livingCellsIndex = livingCellsIndex + 1;
aliveCells = 0;
newGrid = grid;
```

Then, the script runs another nested for loop to check the neighbors of each cell:

```
for row = 1:num_rows
    for col = 1:num_cols
        ....
    end
end
```

Within the for loop, the script finds the currentCell and identifies neighboring indices:

```
currentCell = grid(row, col);
neighborSum = 0;
N = row - 1;
S = row + 1;
E = col + 1;
W = col - 1;
```

Then, the script checks if the neighboring indices are not in the bounds of the array. If so, the indices wrap around to predetermined indices:

```
if N < 1
   N = num_rows;
end
if S > num_rows
   S = 1;
end
if W < 1
   W = num_cols;
end
if E > num_cols
   E = 1;
```

```
end
```

Then, the script calculates the sum of the neighboring indices:

```
neighborSum = grid(N, W) + grid(N, col) + grid(N, E) + grid(row, W) + grid(row, E) +
grid(S, W) + grid(S, col) + grid(S, E);
```

Then, the script updates the current cell based on whether or not the current cell is already dead and the neighboring indices sum:

```
if currentCell == 1 && neighborSum ~= 2 && neighborSum ~= 3
        newGrid(row, col) = 0;
elseif currentCell == 0 && neighborSum == 3
        newGrid(row, col) = 1;
end
```

Afterwards, the script updates the grid and counts the number of living cells are in the new grid:

```
grid = newGrid;
for row = 1:num_rows
        for col = 1:num_cols
                if grid(row, col) == 1
                        aliveCells = aliveCells + 1;
                end
        end
end
livingCells(livingCellsIndex) = aliveCells;
generations = generations + 1;
```

Then, the script animates the results:

```
imagesc(grid);
drawnow;
```

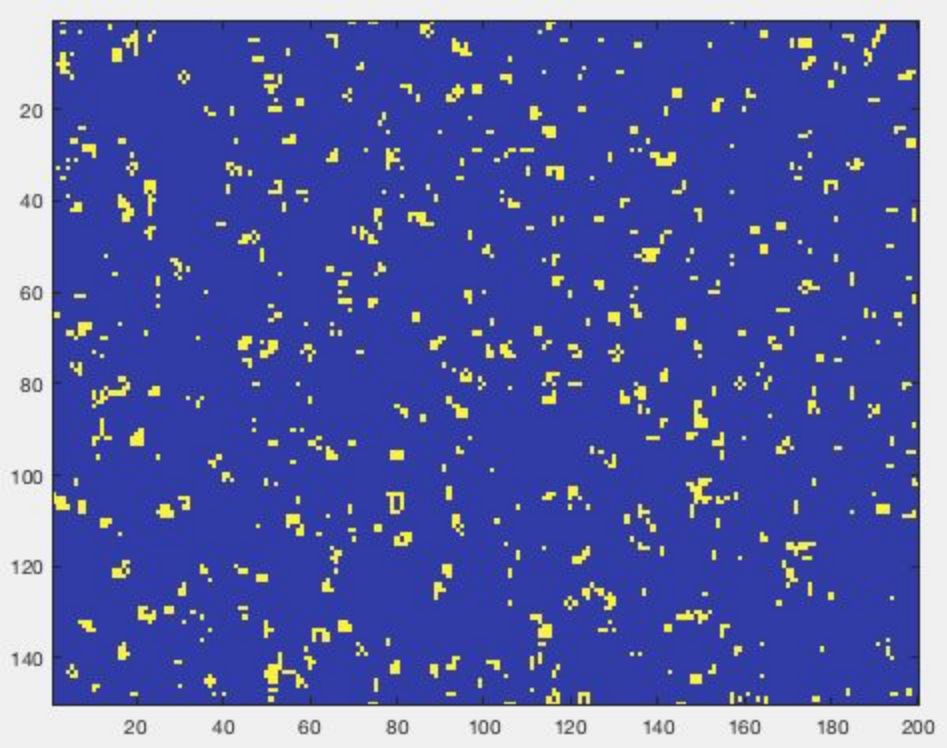
After the passing of all generations, the script plots the living cells as a function of generations:

```
plot(livingCells);
```

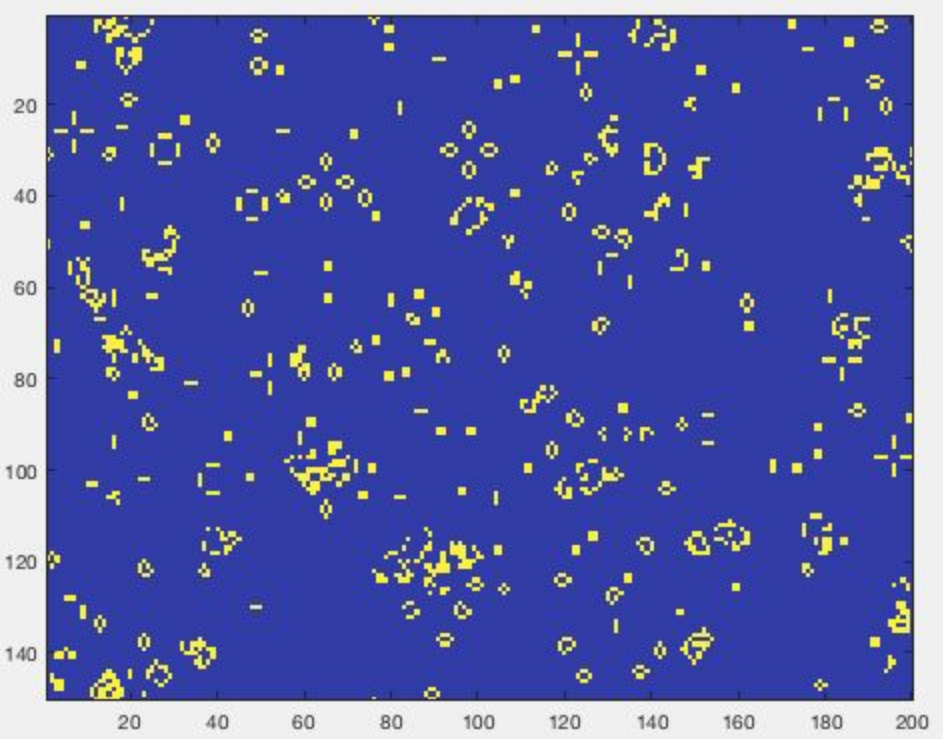
1.3 Calculations and Results

When the program is executed the following graphs are created:

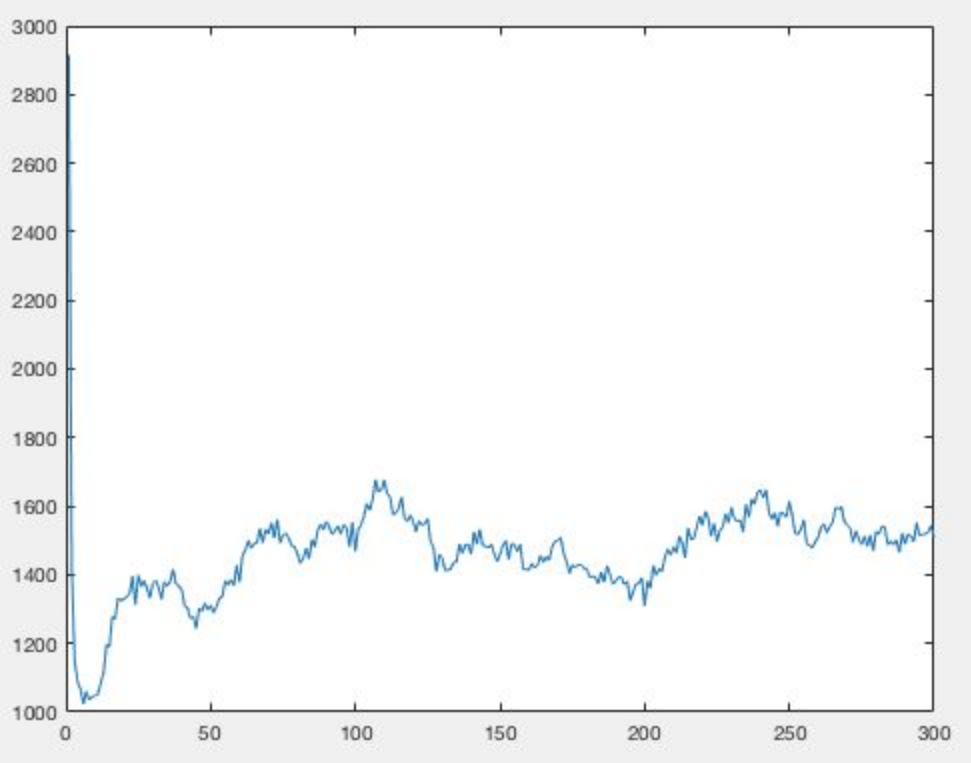
Generation 1



Generation 2:



Living Cells as a function of generations:



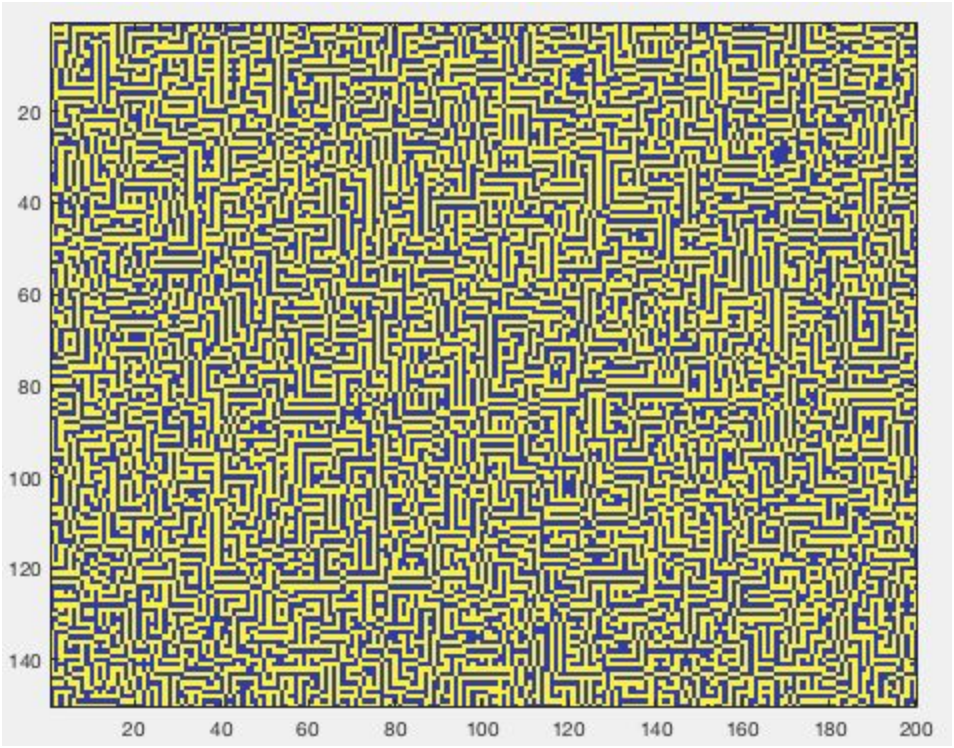
The data shows that the initial population plummeted, but stabilized over the generation at around 1500.

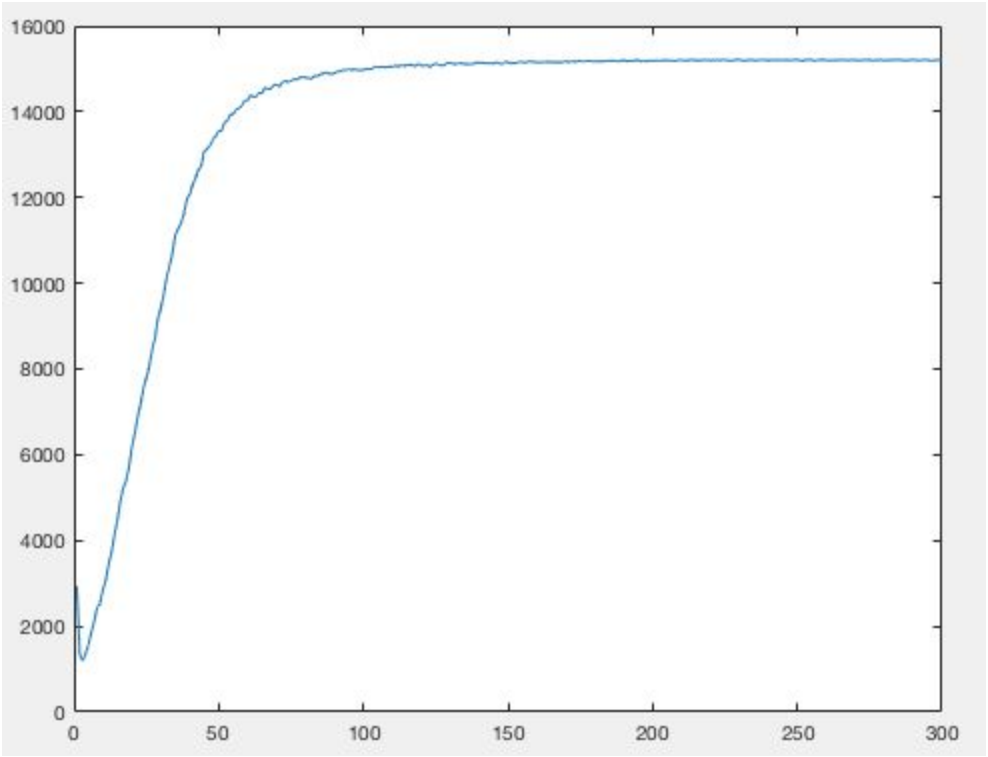
1.4 Discussion

The data shows that the initial population of around 3000 plummeted greatly in the first few generations. This is due to overpopulation, which greatly reduced the population. However, after the initial drop in population, the numbers quickly increase and stabilize around the 1500-1600 mark. This is due to the creation of a sustainable population that is resistant to overpopulation and/or isolation.

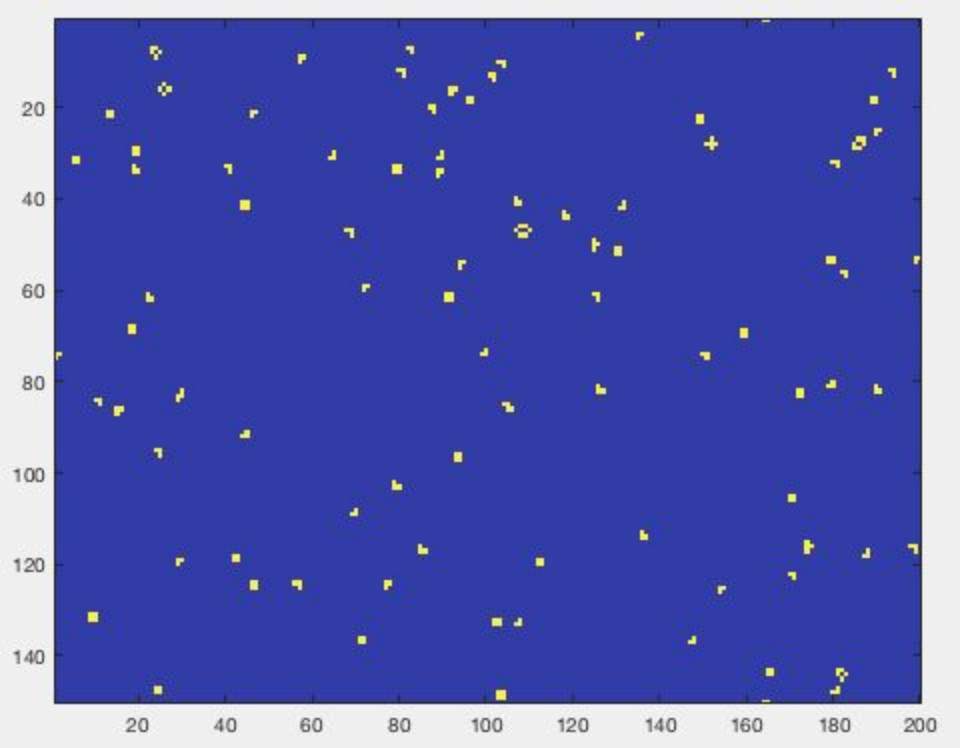
A commonly occuring cell pattern that persists over multiple generation is the cycle of increase to decrease back to increase in population. This cycle is due to the stabilization of the ecosystem that eventually leads to overpopulation that causes another decrease until the ecosystem stabilizes again. This cycle seems to happen every 50 years. This is most visible in the last graph. The total number of living cells over time follow this cycle pretty accurately.

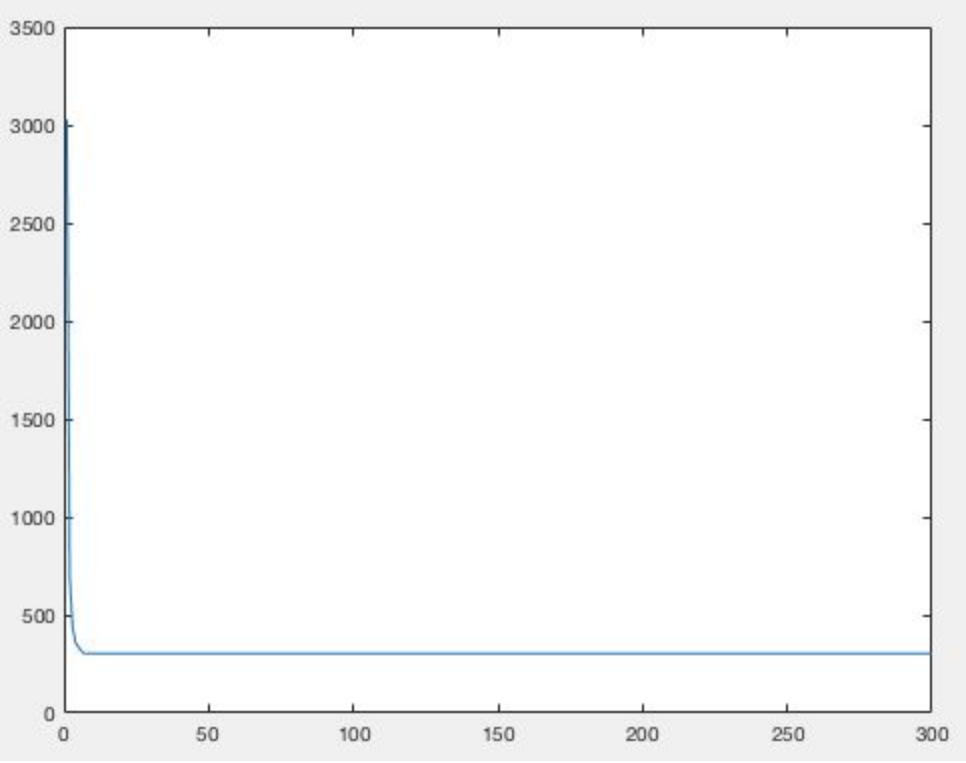
Changing the rules of the simulation can drastically alter the simulation. For example, by changing the limits of over-population, you can easily population nearly the entire array. By making the upper limit of death 4, the following graphs are created:



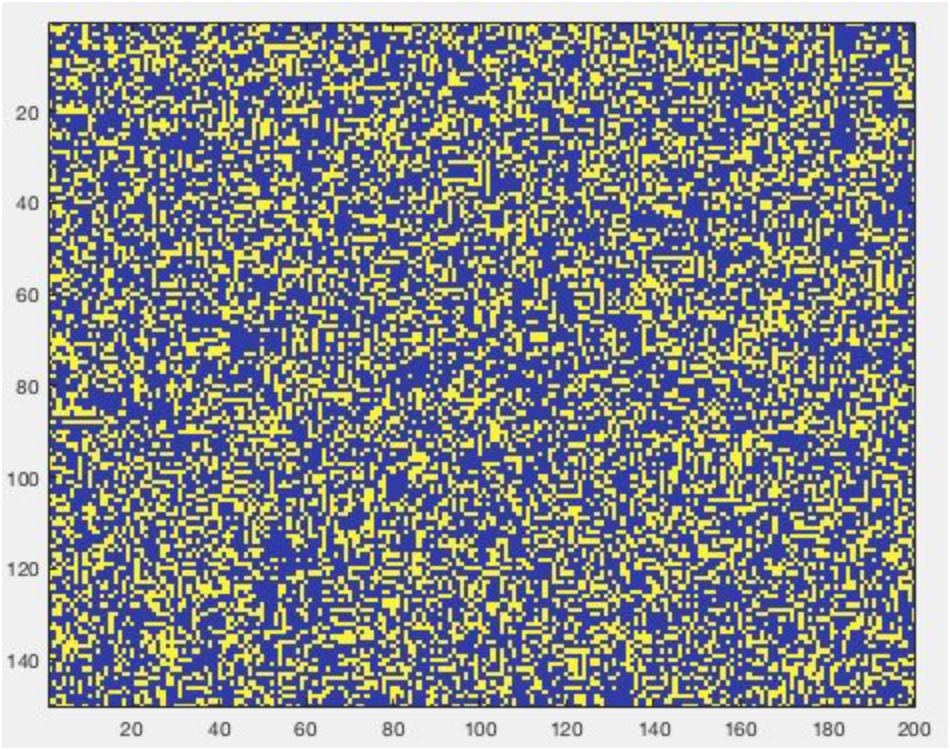


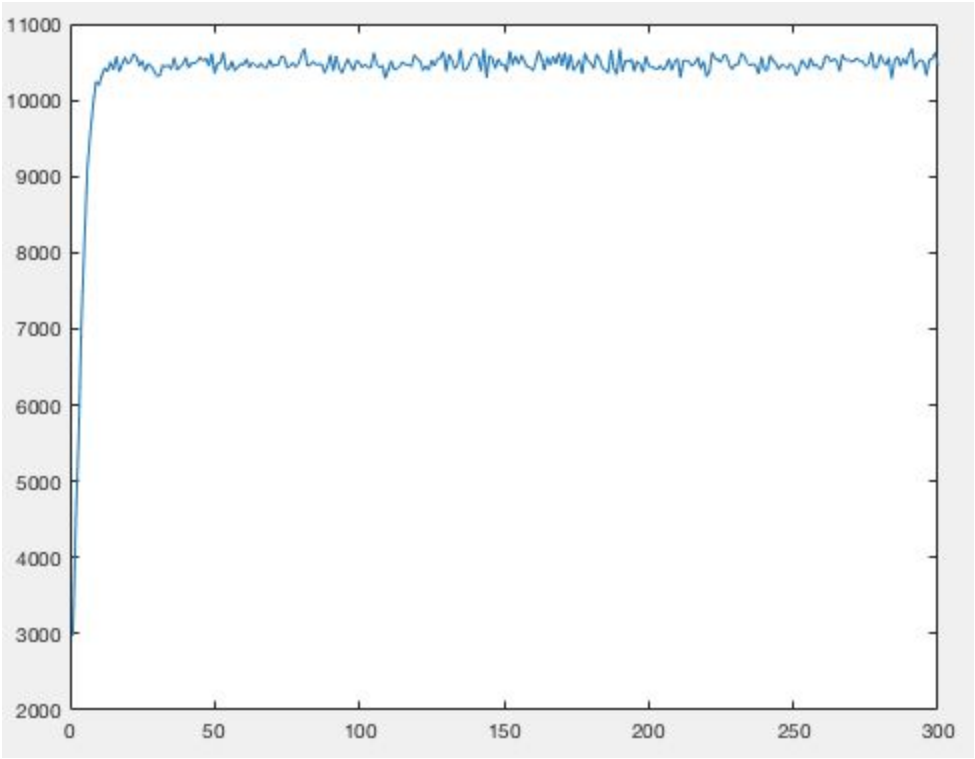
For whatever reason, when I change the alive condition to 4 neighbors, the population actually stagnates at around 400.





When I decreased the neighboring elements to live to 2, the population surged and then hit a point where the population just became erratic:


20
40
60
80
100
120
140
20
40
60
80
100
120
140
160
180
200


11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
0
50
100
150
200
250
300

2. Euler-Bernoulli Beam Bending

2.1 Introduction

The goal of this problem is to solve a system of equations to study the displacement of a simply-supported aluminum beam subjected to a single point load.

2.2 Model and Methods

The script begins by declaring the constants of this specific problem:

```
L = 1;     % bar length
R = 0.013; % circular-tube cross section outer radius
r = 0.011; % circular-tube cross section inner radius
P = 2000;  % applied force
d = 0.75;  % distance of applied force
E = 70*10^9;     % modulus of elasticity
I = (pi/4)*(R^4-r^4); % moment of inertia of the cross section
```

Then, the script creates an x array that evenly spaces out x distances in 20 spaces from 0 to L:

```
x = linspace(0, L, 20);
```

Then, the script creates the A matrix representing the spaced node of the bar and discretizing all the interior points using the central, second-derivative stencil:

```
A = zeros(length(x), length(x));
A(1,1) = 1;
A(length(x),length(x)) = 1;
for row = 2:length(x)-1
   A(row,row-1) = 1;
   A(row,row) = -2;
   A(row,row+1) = 1;
end
```

Then, the script forms the right-hand matrix b by creating a 20x1 size matrix of zeros and calculating the middle values using the equations given in the problem:

```
b = zeros(length(x),1);
for index = 2:length(x)-1
   b(index,1) = ((x(index)-x(index-1))^2)/(E*I);
   if x(index) <= d
```

```
        b(index,1) = b(index,1)*(P*(L-d)*x(index))/L;
    else
        b(index,1) = b(index,1)*(P*d*(L-x(index)))/L;
    end
end
```

Then, the script solves the system of equations and finds the maximum y value and the x value where that y value coincides:

```
y = A\b;
[yMax, yIndex] = min(y);
xIndex = x(yIndex);
```

Then, the script finds the theoretical max y value and calculates the error percentage between the theoretical and the actual:

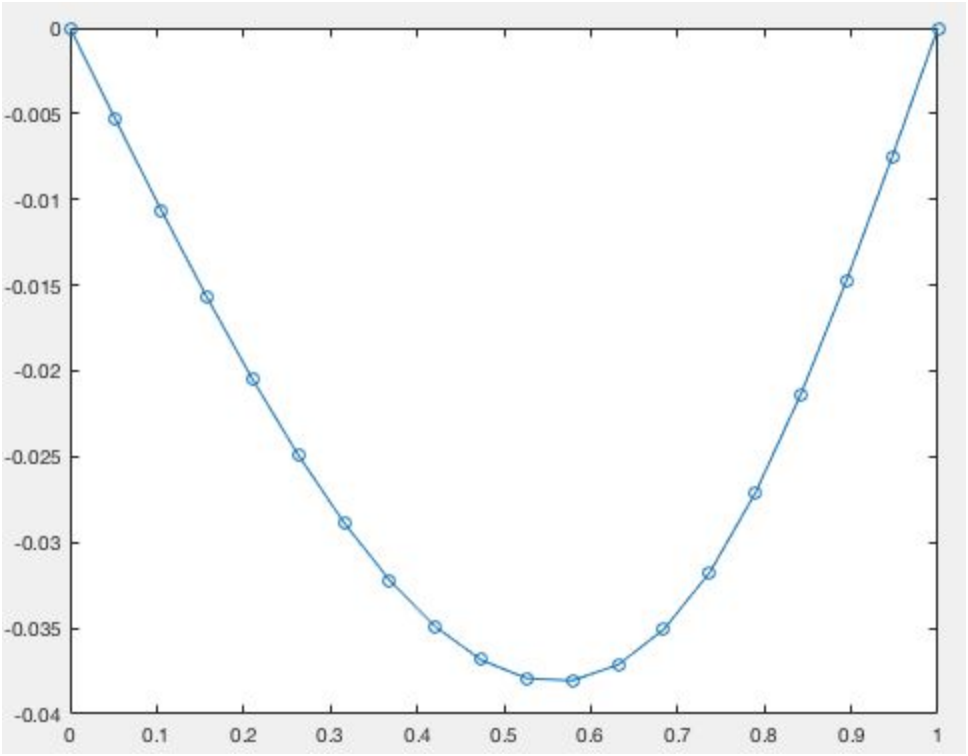
```
c = min(d, L-d);
yMaxTheoretical = (-P*c*(L^2-c^2)^1.5)/(9*sqrt(3)*E*I*L);
error = abs((yMaxTheoretical - yMax)/yMaxTheoretical)*100;
```

Finally, the script plots the displacement as a function of x:

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

2.3 Calculations and Results

When the program is executed the following graphs are created:



As the graph shows, the maximum displacement is -0.0381 and the x coordinate that corresponds to that value is 0.5789. The theoretical error is around 0.0169% because the theoretical maximum displacement is -0.038.

2.4 Discussion

The graph reflects a negative parabolic function that have zeros at x = 0 and x = 1. The graph has a min value between x = 0.5 and 0.6.

According to my calculations and the program, the maximum displacement is -0.0381 (negative because the force is in the downward direction). The maximum y-displacement occurs at x coordinate 0.5789.

After calculating the theoretical maximum displacement, which is -0.038, the error percentage between the theoretical and the actual is 0.0169%.

When I increased the number of discretization points to 100, the error of the maximum displacement became smaller. This makes sense because the approximation now has more discrete points thus becoming more accurate and precise.

When experimenting with the d value, the x value that denotes where the maximum y displacement is varies between 0.4 and 0.6, depending on how far right and how far left the applied force is. Using this info, I have created a range of x values that always contain the location of the maximum displacement excluding when d = 0 or d = L as xMin = 0.42 and xMax = 0.5758.