David Curry
ID: 304755606
CEE/MAE M20
May 21, 2018

Homework 7

1 The Game of Life

1.1 Introduction

The purpose of this problem is to simulate the future of living cells according to rules set out by John Conway's theories in “Game of Life.” These rules determine whether a cell lives or dies based on its neighbors. Initially there was only a 10% chance a given cell was alive, and I set out to determine how many cells lived after 300 timesteps. I visualized this with a colorful graph that updated every timestep, and a graph of the number of alive cells versus time.

1.2 Models and Methods

The first thing I included in the script was to create an initial matrix with a 10% chance of a cell being alive. This was done by initializing the number of rows and columns and then creating a random matrix of that size with the `rand` function. This assigned value between 0 and 1 for all the values. I then used 2 nested `for` loops to create the 10% chance condition. In the loops, I used if-elseif statements to assign a 0 or 1 to every value. If the random number generated was less than 0.9, then it became a zero. If above 0.9, it became a 1. This is all shown below.

```
num_rows = 150;
num_cols = 200;
A = rand(num_rows, num_cols);
for col = 1:num_cols
        for row = 1:num_rows
                if A(row,col) <= 0.9
                A(row,col) = 0;
                else
                A(row,col) = 1;
                end
        end
end
```

I then graphed the initial condition with the `imagesc` function. I also initialized the time and alive vectors with the `linspace` and `zeros` functions, respectively. This is shown below.

```
        imagesc(A);
        title('1');
        time = 300;
        timevec = linspace(0,time,time);
        alivevector = zeros(1,time);
```

Next I used three nested `for` loops to loop thru all the timesteps, all the columns, and all the rows of the matrix. I then defined all the neighbors of any given point, even if the point lies on the edge of the matrix. For edge cases I used `if` statements to loop around to the other side of the matrix. Then I calculated the sum of all these neighbors, and used this sum to determine if the cell would live or die. This was done with `if-elseif` statements, `&&`, and `||` statements. In these `if` statements I also updated the number of alive cells after it had changed. All of this is shown below. The West and East if statements were essentially the same as the North and South statements so I left them out.

```
for k = 1:time
    alive = 0;
    for col = 1:num_cols
        for row = 1:num_rows
            A(row, col);
            N = row - 1;
            S = row + 1;
            E = col + 1;
            W = col - 1;
            if N < 1
                N = num_rows;
            end
            if S > num_rows
                S = 1;
            end
            neighbors = A(N,W) + A(N,col) + A(N,E) + A(row,W) + A(row,E) +
            A(S,W) + A(S,col) + A(S,E);
            if A(row,col) == 1
                if neighbors == 2 || neighbors == 3
                    A_new(row,col) = 1;
                    alivevector(k) = alivevector(k) + 1;
                else
                    A_new(row,col) = 0;
                end
            else
                if neighbors == 3
                    A_new(row,col) = 1;
                    alivevector(k) = alivevector(k) + 1;
                else
                    A_new(row,col) = 0;
                end
```

After all this, I ended the column and row loops and then updated the *A* matrix inside the timestep loop. I also used `imagesc` and `drawnow` in the time loop to graph every timestep on the same graph as a moving picture. This is shown below.

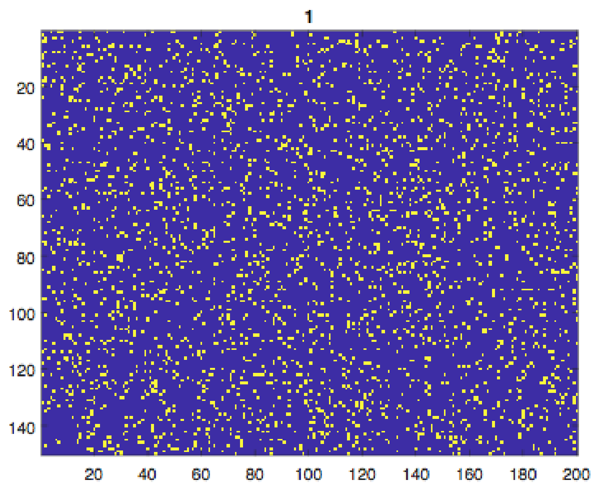
```
        A = A_new;
        imagesc(A);
        title(k);
        drawnow;
```

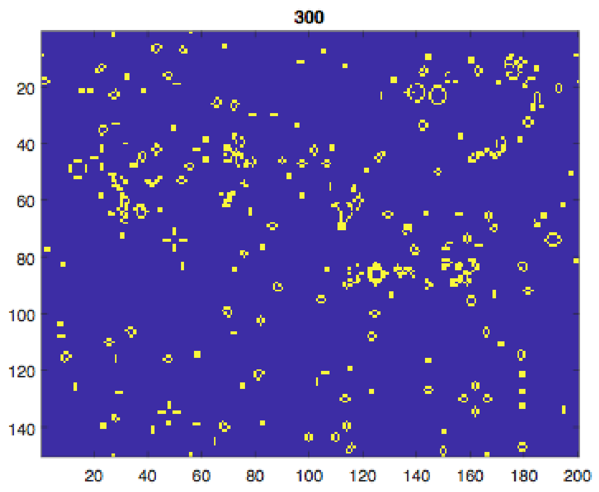
Finally I graphed the number of alive cells versus time with the `plot`, `xlabel`, `ylabel`, and `title` functions. This is shown below.

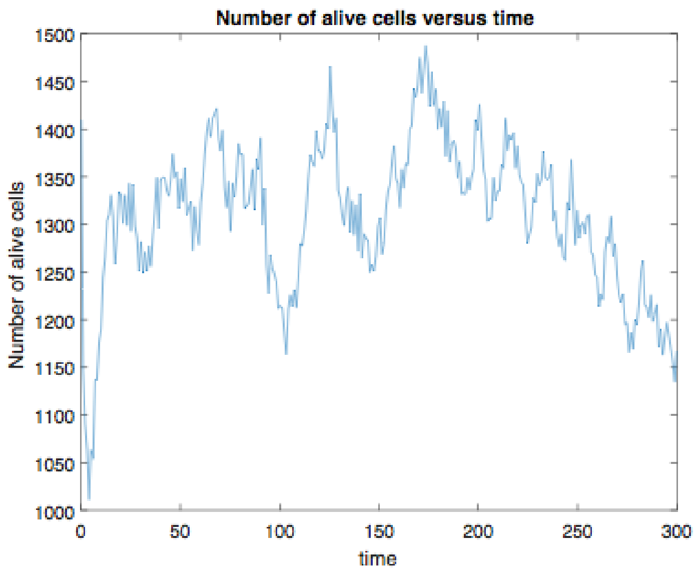
```
figure;
plot(timevec,alivevector);
xlabel('time');
ylabel('Number of alive cells');
title('Number of alive cells versus time');
```

1.3 Calculations and Results

After running the script, the following is outputted. The actual output is a moving image so I just included the first generation and the last generation and a graph of the number alive versus time.

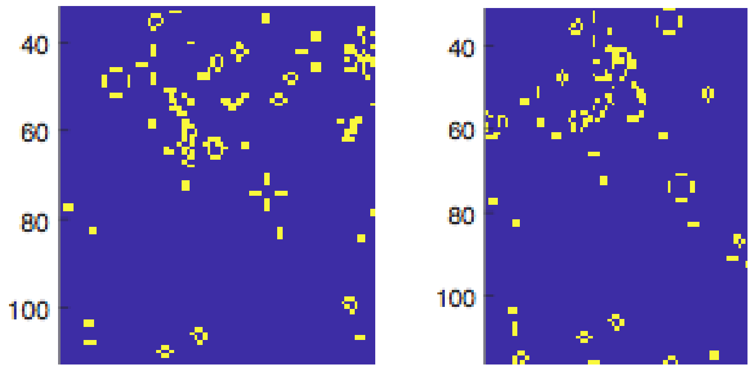




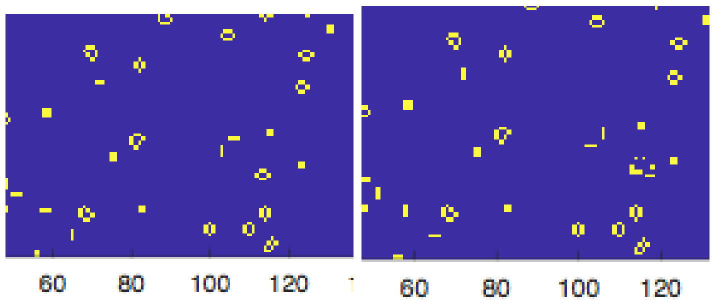


1.4 Discussion

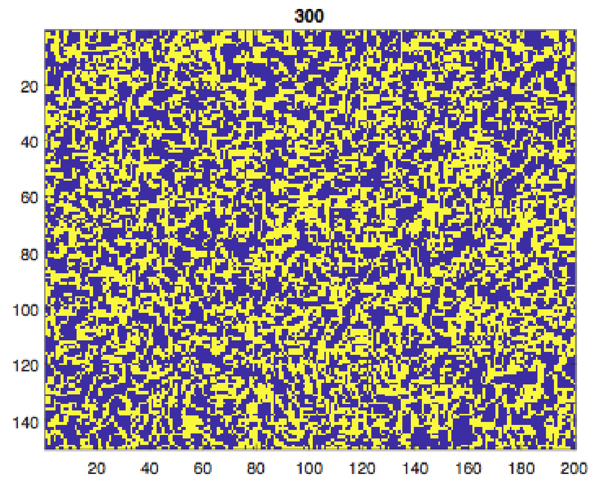
As shown in the graphs, the number of living cells drastically changed from the first generation to the last generation. Watching the moving image, it was clear that there were a few patterns. The cross design and the circle design would switch every generation, shown around the 75 mark in the graphs below.

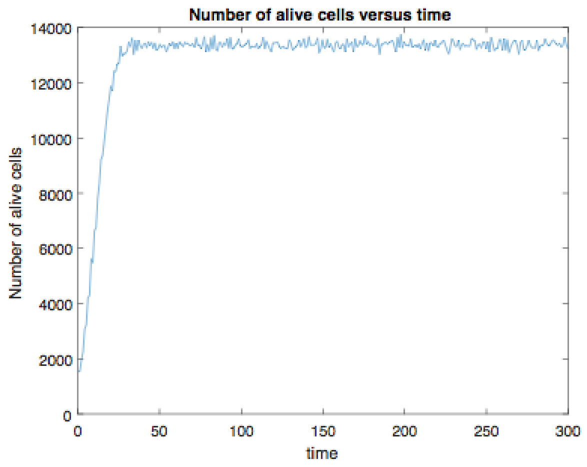


Another pattern I noticed was when a group of cells was in an 'L' shape. These cells would flip directions every generation. This is shown below by the 100 mark shown below



From the graph above of the number of alive cells versus time, I do not think the number of alive cells follows a trend. The graph is very jagged with large jumps in between some generations. However, after a while the general trend does seem to be decreasing in the number of alive cells. I also tried changing the rules to allow 4 neighbors for survival, whether or not the cell was already alive. This is shown in the graphs below.





This rule change allowed for many more cells to be alive, and it produced a trend that stayed mostly constant after a short amount of time. Perhaps the previous results based on John Conway's theories would level off after a longer period of time.

2 Euler-Bernoulli Beam Bending

2.1 Introduction

The purpose of this script is to study the deflection of a simply-supported aluminum beam that is subjected to a downward point-source force. The ends of the beams do not deflect and the rest of the beam bends according to the Euler-Bernoulli equation for bending moments. This has a second derivative, so I converted the equation into a matrix calculation by discretizing the equation. I then graphed the deflection versus the distance along the beam to determine the maximum deflection.

2.2 Models and Methods

I first defined all the initial conditions necessary for the calculations. I used the `zeros` function to initialize the *A* matrix. Most of this is repetitive, so only a few lines of this is shown below.

```
I = (pi/4)*(R^4 - r^4);
nodes = 20;
step = L/(nodes-1);
A = zeros(nodes,nodes);
```

Next I defined the *A* matrix according to the discretized equations. This is done with two nested `for` loops to iterate through all of these values in the matrix. In the loops, I used `if-elseif` statements to assign values to each point. This is shown below.

```
for row = 1:nodes
      for col = 1:nodes
            if row == 1 && col == 1
                  A(row,col) = 1;
            elseif row == nodes && col == nodes
                  A(row,col) = 1;
            elseif row == col
                  A(row,col) = -2;
                  A(row, col+1) = 1;
                  A(row,col-1) = 1;
            end
      end
end
```

I then defined a *b* and *M* vector with the `zeros` function. After that I updated both vectors with a `for` loop. In the loop, I used `if-elseif` statements to assign values to each *M* and *b* value based on the given equations. This is shown below.

```
b = zeros(nodes,1);
M = zeros(1,nodes);
for k = 1:nodes
      if k < d*nodes
            M(k) = (-P*(L - d)*step*(k-1))/L;
      else
            M(k) = (-P*d*(L - (step*(k-1))))/L;
      end
      if k == 1 || k == nodes
            b(k) = 0;
      else
            b(k) = ((step^2)*M(k))/(E*I);
      end
end
```

Finally I created the deflection vector *y* by dividing *A* by *b*. I then created an *x* vector with the `linspace` function. I plotted these vectors with the `plot`, `xlabel`, `ylabel`, and `title` functions. This is all shown below.

```
y = A\b;
x = linspace(0,1,nodes);
plot(x,y,'o-');
xlabel('Distance along beam');
ylabel('Deflection');
title('Deflection versus the distance along beam');
```

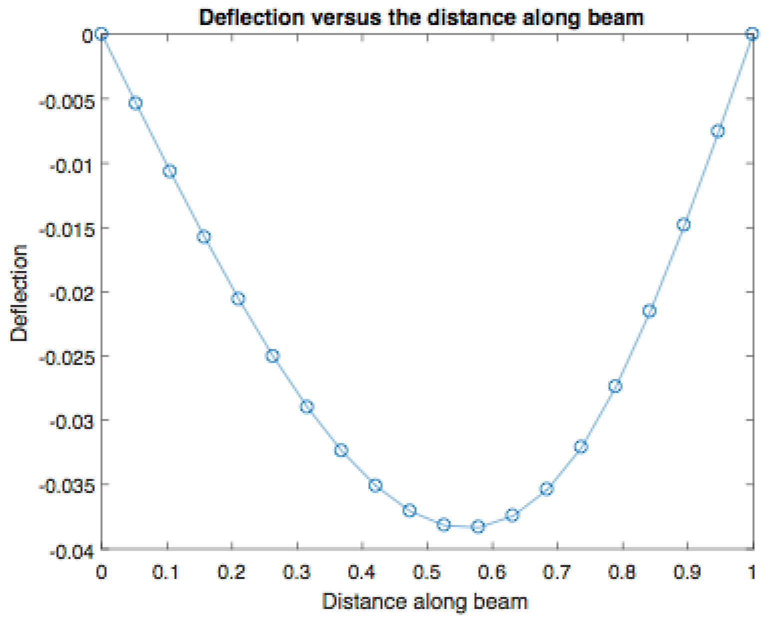
I also had to compare the calculated maximum displacement with the theoretical displacement. To do this I found the calculated max by using the `min` function. I then determined where this occurred on the beam with a `for` loop that updated the *maxi_x* variable once it found the maximum displacement. The theoretical maximum was calculated using a equation that was given. Finally I calculated the error between the two and printed the results with the `fprintf` function. This is all shown below.

```
maxi = min(y);
for k = 1:nodes
      if y(k) == maxi
            maxi_x = x(k);
      end
end
c = min(d,L-d);
max2 = (P*c*(L^2 - c^2)^1.5)/(9*sqrt(3)*E*I*L);
error = abs(maxi - max2);
fprintf('Maximum displacement occurred at %.3f meters\n', maxi_x);
fprintf('Maximum displacement was %.5f meters\n',maxi);
fprintf('The error between the actual and calculated maximums is %.5f
meters\n',error);
```

2.3 Calculations and Results

After running the code, the following is printed and plotted.

```
Maximum displacement occurred at 0.579 meters
Maximum displacement was -0.03833 meters
The error between the actual and calculated maximums is 0.00028 meters
```



2.4 Discussion

As shown above, this method was very accurate since the error was so small. I increased the number of nodes across the beam to 100 nodes and the error decreased even more. This is shown in the output below.

```
Maximum displacement occurred at 0.556 meters
Maximum displacement was -0.03805 meters
The error between the actual and calculated maximums is 0.00001 meters
```

I also experimented with the placement of the force to find the range over which the maximum displacement would occur. I used *d = 0.99* and *d = 0.01* to find this range. Both case's outputs are shown below, in order.

```
Maximum displacement occurred at 0.579 meters
Maximum displacement occurred at 0.421 meters
```

From these outputs I can say that the maximum deflection should always be in between 0.4 and 0.6 for a one meter section of a beam, or in between 40% and 60% of the total length.