Jared Rivera

804603106

CEE/MAE M20

February 7, 2017

Homework 4

1 Epidemic

1.1 Introduction

The goal of this problem is to model changes in population over time during an epidemic. Three sub-populations will be kept track of, those being the people that are infected, susceptible, or removed meaning they are dead or immune. This will be done by integrating the Lotka-Volterra equations using the Euler forward integration method. The population values will be tracked over time by updating an array inside of the loop in which the integration is performed. The epidemic will be tracked in two cases. The first is a normal spread of disease, while the second involves the development of a vaccine after the initial outbreak. The vaccine will be tested to see if it is more effective by being released a day early or by reducing the time it takes to vaccinate the population. A plot of the population changes over time will be produced, and the final values of each subgroup will be printed to the command window at the end of the time span. Finally, both situations will be timed to see which takes longer to run.

1.2 Model and Methods

The first step in this problem is to initialize the constants of the differential equations using what was provided in the problem statement. This varies between parts b and c but the method is the same. The general equations are shown below

$$s_{k+1} = s_k - \Delta t(\beta i_k s_k)$$
$$i_{k+1} = i_k + \Delta t(\beta i_k s_k - \gamma i_k$$
$$r_{k+1} = r_k + \Delta t(\gamma i_k)$$

Next is to set up the time variables for the loop, plot, and arrays. This is done in a similar fashion as last week where a final value and a step size is specified. Next, the arrays are initialized by creating an empty array with the `zeros` function and assigning the first value of the array as the initial populations, as seen below

```
i=zeros(1,npoints);
    i(1)=iold;
```

Now the loop is started and the iterations of the Euler integration are carried out to completion. During the process, the arrays are continually updated with the population values at each time value

```
s(k+1)=snew;
```

Where `snew` is the value calculated by the current Euler method iteration. The loop carries through to completion, and the arrays are used to plot the sub-population values versus time using the plot function and built in features to customize the plot, as seen on the following page

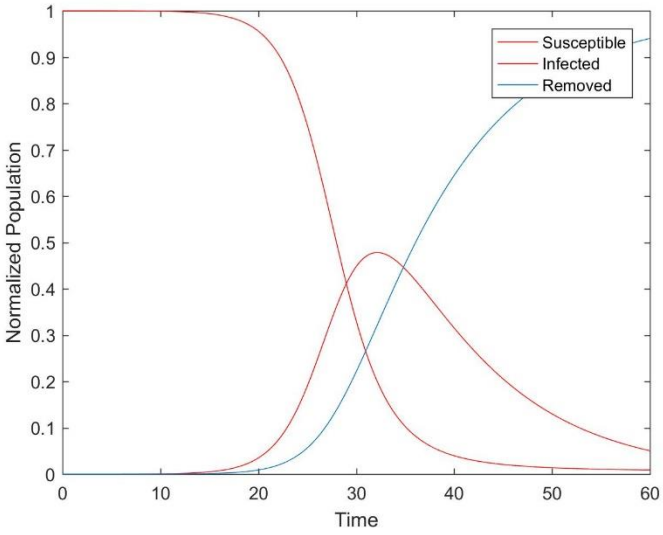
```
plot(t,s,'r',t,i,'r',t,r);
legend show;
legend('Susceptible','Infected','Removed');
xlabel('Time');
ylabel('Normalized Population');
```

Where the plot function creates the plot of each population versus time, the legend specifications create the legend for the figure, and the label functions label the axes. Finally, the final sizes of the population groups are printed to the command window.

1.3 Calculations and Results

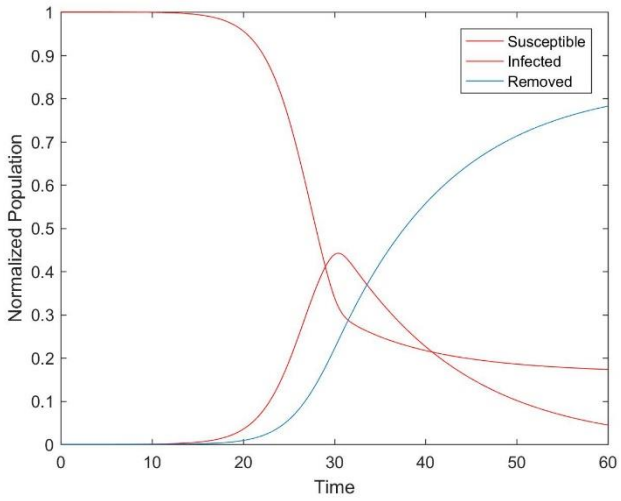
This section will be broken up into two parts, one for the normal epidemic(b) and one for the vaccinated epidemic(c) with tvac=28 and n=5.

For part b, when the script is run the following plot is displayed, and what follows is printed to the command window:




```
Final Size of Population Groups

Susceptible: 0.04 million

Infected: 0.20 million

Removed: 3.76 million

Elapsed time is 0.658492 seconds.
```

Similarly, running part c yields what is displayed on the following page.




```
Final Size of Population Groups

Susceptible: 0.69 million

Infected: 0.18 million

Removed: 3.13 million

Elapsed time is 0.671417 seconds.
```

One thing that can be seen from the two different results is that the removed population is lower in the case of the vaccination than in the case of the normal spread at the end of 60 days, and it is important to note that the removed population doesn't differentiate between the dead and the immune.

1.4 Discussion

As stated in the results section, the removed population grows in the case of the vaccination. Upon further inspection, one can see that these numbers, relative to the normal case, come from the susceptible population. Rather than dramatically shrinking the infected group, the vaccine makes people less likely to contract. Operating under this logic, one can safely assume that the larger removed population in the vaccinated case is due to higher numbers of immune people, rather than higher numbers of dead people.

After changing values of tvac and n, it seems that a reduction in the time it takes to develop a vaccine is more effective than a reduction in the time it takes to administer it. This makes intuitive sense because if a vaccine is introduced earlier, it takes makes people immune to the disease and begins to reduce the population base from which the infected can spread the disease. In the case of faster administration, the disease has more time to spread to more people before the vaccine can reach any subset of the population. This can be shown numerically by the fact that a reduction of tvac by one reduces the maximum infected population by thousands more than a single day reduction in n does.

The most effective combination of tvac and n values is a reduction in both. It's easy to assign a value of 1 to each and call it a day, but a realistic program should have realistic parameters. Assuming it takes substantial time for research and development as well as transportation of the vaccine, my

best estimate is tvac=14 and n=3. Two weeks for vaccine development and three days for implementation for a population of 4 million people seems realistic as it is comparable to this epidemic occurring in LA and every medical professional working at full capacity (except those infected in the problem statement that is).

As can be seen in the results section, the case of the vaccination takes longer to run in MATLAB than the normal case. This makes sense as the β value in the vaccination case is a function that must be iterated throughout the implementation of the script, while the normal case assigns only a single value.

2 DNA Analysis

2.1 Introduction

The goal of this problem is to write a program that can analyze sequences of DNA and determine the location and length of different protein coding segments, all to be applied to human chromosomes. This will be done by iterating through the sequence by codon and determining the locations of start and stop codons, as well as the distance between them. The results will then be analyzed in order to determine the maximum, minimum, and average lengths of protein coding segments and also to determine the amount of functional DNA in the chromosome. All this will be done by use of iteration through a loop and by updating protein segment length arrays. The results will finally be printed to the command window.

2.2 Model and Methods

The first step in this problem is to import the array holding the chromosome data, which is done using the `load` function. Next, the total length of the chromosome in terms of number of bases is calculated using the `length` function. Right before the loop is started to check the codons, the array that will hold the length values of the protein coding segments is initialized and the start criteria for the loop is set to false. The for loop now begins and iterates by three to look for the start codon as long as one hasn't been found, which is specified by setting the `start` variable to false, as seen below

```
if start==0 &&(dna(k)==1 && dna(k+1)==4 && dna(k+2)==3)
                        start=k;
                          end
```

This logical statement checks for the right codon and resets the start variable so that the loop begins looking for a stop codon. This is done using similar logical gates, and once a stop codon is found,

```
stop=k+2;
```

From here, the total length of the protein coding sequence can be found by

```
slength=stop-start+1;
```

Using the modulus function to make this this is an actual codon and not a fragment, the loop then determines whether or not to save the length into the array. If the length is an actual codon, it is saved.

```
if mod(slength,3)==0
        lv=lv+1;
lengthvalues(lv)=slength;
        start=0;
```

The start criteria is set back to false to alert the loop to once again begin searching for a start codon. Next, using the built in statistical functions the average, maximum, and minimum length values are found by taking the non-zero portion of the array and applying the functions, example shown below

```
avglength=mean(lengthvalues(1:lv))
```

Where lv is the total number of segments saved into the array. Finally, these values are printed to the command window in the specified format.

2.3 Calculations and Results

Once the program is run, the following is printed to the command window

```
Total protein coding segments: 4294

Average length: 89.35

Maximum length: 2967

Minimum length: 6
```

2.4 Discussion

One can see right away that there is massive variation between the length values and that due to the smaller value of the average, the shorter segments are much more common than the long ones. On top of that, only 30.34% of the total chromosome segment is used in giving directions for protein coding. This can be calculated easily by multiplying the average length by the total number of segments and dividing the answer by the total number of bases in the chromosome segment.

There are three available stop codons, but only one start codon. The stop codons don't come up in the same frequency either. In order to see the frequency at which each stop codon appears, a counter for each was placed in the beginning of the script and set to zero. Each time the script found a stop codon in the correct position, it incremented the appropriate counter by one. It was found that the most commonly occurring stop codon is the 431, or TGA codon. The least common was 413, or TAG. The variability in stop codons may be due to mutations in the human genome over the course of evolution, and accounting rigorously for mutation makes this problem a lot more complicated.

Bases in a codon may be deleted, added, or replaced. This can make the current search for start and stop codons miss real life codons, and can also shift the window through which the program searches for codons. One way to account for mutation in this script would be by looking at individual bases in a codon more thoroughly. Rather than practically using the exclusive logicals, one could build in some allowable error. For instance, each base in a codon could be assigned a true or false value dependent upon whether or not it matched a real value being searched for.

Assuming the programmer is generous, maybe they'll allow for 33% error in each codon. This means that only two need to match the real codon the program is searching for. This could be done using a string compare of a lot of logicals. If the codon is a 66% match, the program could then use if statements to decide where to move the window. If the first and last bases of the codon match the real codon being searched for, providing a [1 0 1] type of match, the window could iterate normally. On the flip side of that, if the first two bases were determined to be part of the codon and it is assumed the middle codon was deleted, providing a [1 1 0] type of match, the window could be iterated by two so that it doesn't get off track. Similar logic can be applied to the case of an addition of a base.