Jared Rivera

804603106

CEE/MAE M20

January 31, 2017

Homework 3

1 The Three Species Problem

1.1 Introduction

The goal of this program is to model changes in population density over time using the Lotka-Volterra differential equations specific to three species, X, Y, and Z. These equations have to be integrated and the population densities, x, y, and z, have to be displayed over time, which will be done using the forward Euler method of numerical integration. The results for a certain period chosen (t=[0,10] in this case) will then be printed to the screen in table format. The code will use a loop to iterate through the many steps needed to make the integration accurate, and the step size of the loop will be changed in order to see how much the step size changes the amount of time the code takes to run. Finally, the initial population densities of the three species will be varied in order to see if there are any boundary conditions that allow the species to live in harmony in the long run.

1.2 Model and Methods

The first step in this problem is to determine when the code should be timed using `tic` and `toc`. For this script, they were placed before the initialization of the population values and after the printing of the table, respectively. Next was to initialize the population densities with what was given in the problem statement (though this was changed for experimentation multiple times). Next came determining the timing and step for the loop. This was done using a similar method to what was used in class for the yeast example, and is given below.

```
tfinal=10;
deltat=0.01;
tstep=ceil(tfinal/deltat);
```

Next, the loop is created with the conditions of it being shown below.

```
for k=1:tstep
```

This takes k, the count variable in the loop, and starts it from 1, iterating it by 1, until performed tstep times. This value changes based upon the step size used in each run. The continuous Lotka-Volterra equations for this problem are simplified into discrete functions, the example of x is shown below.

$$x_{k+1} = x_k + \Delta t(1.2x_k\left(1-\frac{x_k}{4}\right) - 1.3x_k y_k - 0.7x_k z_k$$

Which is then converted into code in MATLAB as

```
xnew=xold+deltat*(1.2*xold*(1-(xold/4))-1.3*xold*yold-0.7*xold*zold);
```

This equation, as well as the y and z components, are fed into the loop and after each iteration the old values are given a new value, that being the newly calculated “new” values. This ensures that the loop doesn’t keep using the initial data repeatedly. For the printing of the values in the table, logical statements must be used to only print ones at certain time steps and also to ensure the formatting of the table is uniform, these logical statements are shown on the next page.

```
if mod(k,50)==0&&k~=1000
        fprintf('%.1f\t\t%.2f\t%.2f\t%.2f\n',(k/100),xold,yold,zold);
    elseif k==1000
        fprintf('%.0f\t\t%.2f\t%.2f\t%.2f\n',(k/100),xold,yold,zold);
    end
```

This ensures that only every fifty of the hundreds of data points are outputted, and also creates a separate condition for t=10 since that number will throw off the alignment of the table otherwise.

1.3 Calculations and Results

When the values used in the problem statement are put into the program and it's run, the following is outputted into the command window

```
Time  X     Y     Z

0.0   2.00  2.50  3.00

0.5   0.70  0.80  1.20

1.0   0.53  0.53  0.90

1.5   0.50  0.41  0.78

2.0   0.51  0.34  0.71

2.5   0.55  0.29  0.65

3.0   0.62  0.26  0.60

3.5   0.71  0.23  0.54

4.0   0.85  0.20  0.46

4.5   1.03  0.17  0.37

5.0   1.27  0.15  0.27

5.5   1.59  0.12  0.17

6.0   1.98  0.09  0.09

6.5   2.43  0.06  0.04

7.0   2.86  0.03  0.01

7.5   3.24  0.02  0.00

8.0   3.53  0.01  0.00

8.5   3.72  0.00  0.00

9.0   3.84  0.00  0.00

9.5   3.91  0.00  0.00

10    3.95  0.00  0.00

Elapsed time is 0.005973 seconds.
```

A major difference to be noticed in the trend for each species is that X's population undergoes oscillation then a steep increase, while Y and Z's populations steadily decline.

1.4 Discussion

The fact that in the initial populations given X crowds out Y and Z tells the user that X is a very aggressive species. It has the smallest initial population, but seems to use the other two as prey and grow quickly as they are available. Through experimenting with other population sizes more can be learned about the species. Across the board X seems to be the most dominant and aggressive species. Any time it has a larger initial population it will crowd out the other two, but its primary prey is Z. If there isn't any initial Z, then Y will dominate. In many cases, even if it is smaller than Y or Z, X can manage to come back and crowd them out anyway.

Y seems to be a moderate species. If it heavily outnumbers X then it will crowd out the other two populations. Finally, Z is the bottom of the food chain. It doesn't seem to crowd out the other two and its population goes to zero quickly as long as a X population is present. However, if there is absolutely no X to attack Z then Z will dominate Y. The food chain seems to be that X eats Z, Z eats Y, and Y eats X, with X being the most aggressive, Y being moderate, and Z being docile.

As far as timing the program goes, it takes longer to run with smaller step sizes. This makes sense as smaller step sizes means more iterations the computer must run through, opposed to the smaller amount as the step size increases. With the step being t=0.01, the program takes between 0.006 and 0.009 seconds to run, and this varies by trail likely due to the computer's available processing ability at that moment. In general, though, smaller step size increases time needed to run the program.

2 The Pocket Change Problem

2.1 Introduction

The goal of this program is to calculate the average number of coins that will be given to a consumer as change after a purchase. The program assumes that only the least amount of coins per value of change is given and that there is an equal chance in the long run of getting any value of change from 0 to 99 cents. The script can be modified to allow denominations that don't currently exist in the American system (i.e. a 12-cent coin) and will determine the average number of coins in that scenario. This program will serve as an easy check to see which combination of denominations allows for the least amount of change on average.

2.2 Model and Methods

The first step in this problem is to assign the coins to variables and assign proper values to each. The standard set will the those used in the American system and the variable names will hold the corresponding names used in the American system as well. Next, the amount of coins given back as change is set to zero as one would start with zero coins before the change was given to them. Next, the main loop is started. It loops through all possible cent values for change (0-99) and assigns a value corresponding to that count number to the change variable as seen below

```
for k=0:99
change=k;
```

At this point, the change must be divvied up into coins and given to the customer. This is done with a chain of while loops who's logical conditions check the value of the change and give the appropriate coin in return. An example is shown below

```
while change>=quarter
change=change-quarter;
coincount=coincount+1;
```

In this case the coin in question is the quarter. As long as the value of the change is larger or equal to that of a quarter, the value of the quarter is deducted from the change value and the coin count is raised by 1 to reflect the addition of a quarter. The change is then passed onto loops for smaller denominations. Once the change is depleted and the coins are given and counted, the average across the range is taken, as shown below

```
avgcoincount=coincount/100;
```

This is simple algebra. The result is then printed to the screen.

2.3 Calculations and Results

When the program is run using the denominations given in the problem statement, the following is outputted to the command window

```
Average Number of Coins: 4.70
```

2.4 Discussion

The average is a decimal, which doesn't make much intuitive sense since a person cannot receive a fraction of a coin as change. But this can just be taken to mean that on average one can expect 4 or 5 coins as change, but 5 more often than 4. This number changes, however, as the values of denominations of coins and available change values change.

For example, if the penny is removed from circulation and everything is rounded up to the nearest nickel, the program returns an average value of 2.69 coins. Again, this means that one will receive 2 or 3, but more likely 3. This makes intuitive sense because each denomination can be made up of less than 4 of the lesser valued ones (i.e. a dime equals two nickels). An exception to this, however, is the penny. There are five pennies per nickel. This means that a coin that appears commonly in change is being removed and therefore less coins are needed. Just by looking at the code one can see this. If every value is rounded to the next highest nickel, there is no need to the penny while loop. This eliminates many iterations that can potentially up the coin count expression.

After experimenting with many coin values, one specific one rises to the top as providing the lowest average coin count. In this system, the quarter will be valued at 40 cents, the dime at 11 cents, the nickel at 3 cents, and the penny at 1 cent still. This provides an average coin count of 4.12, which is 13% decrease. This may not seem like a lot but it may lower production costs of coins and would allow consumers to carry fewer coins. A downside to this system, though, would be the math. It's more complicated to give someone change in this system than in the system based around 5 cents seen in the current American system.