Duncan Di Mauro
UID: 805163177
CEE/MAE M20
October 22, 2019

Homework 3

1 Relationship of Two Competing Populations

1.1 Introduction

The goal in this problem is to create a script that computes the population of two species under competition. Additionally, the script should create a labelled graph that plots the populations of each species over a given time interval. The populations of these two species are governed by the following Lotka-Volterra equations:

$$\frac{dx}{dt} = x(\alpha - \epsilon x - \beta y) \quad (1)$$

$$\frac{dy}{dt} = y(-\gamma + \rho x_k - \delta y) \quad (2)$$

where x, y are the populations (thousands) of species X and Y. In this case, the relationship between these two species is modeled using α = 3, β = 2, γ = 2, ρ = 1.5, ϵ = 0.3, and δ = 0.2.

1.2 Models and Methods

The forward Euler method was used on the given Lotka-Volterra equations to find the discretized governing equations for the two populations. The resulting equations are shown below:

$$x_{k+1} = x_k + \Delta t\big(x_k(\alpha - \epsilon x_k - \beta y_k)\big) \quad (3)$$

$$y_{k+1} = y_k + \Delta t\big(y_k(-\gamma + \rho x_k - \delta y_k)\big) \quad (4)$$

Where x_k and y_k represent the populations of X and Y at step k, and x_{k+1} and y_{k+1} represent the values of X and Y at step k + 1.

First, the script assigns all given values to variables (ex. `rho = 1.5`). Next, a value for the number of steps needed to go from time t = 0 to t = 10 was found, and was assigned to the variable `t_steps`. This was done by applying the `ceil` function to the equation `(t_final - t_initial)/delta_t)`. This function rounds the resulting value up to the next integer. This is necessary so that the number of steps is an integer. Additionally, we round up so that we have enough steps to reach (and possibly move a little past) the given final time, as opposed to rounding down which would get us very close to, but would not reach, t = 10.

Next, a `for` loop with the condition `for k = 0:t_steps` is used to record the population values for X and Y at every step, starting from their initial values until their values at the final step. A new variable representing time is assigned to `t_initial + k*delta_t`. This is necessary so that when we plot our results, we can use this time value as the x-axis. Equations (3) and (4) are then assigned to their respective variables `xkp1` and `ykp1`, which are equivalent to x_{k+1} and y_{k+1}. The `plot` function is then used to graph the population values as points (species X is green and species Y is red). The `hold on`

function is used after each `plot` function in order to make sure that the points recorded to the graph do not disappear once the loop repeats or ends. More functions are used to add a legend and axis-labels to the graph (note that the legend and labels should be created after the `for` loop ends to reduce the time it takes to run the code). Finally, the variables used in equations (3) and (4) representing x_k and y_k are set equal to the new population values `xkp1` and `ykp1`. This is necessary so that once the loop repeats, it uses the population values obtained from the previous step of *k*. An `if` command is inserted before x_k and y_k are set equal to the new population values, which is shown below:

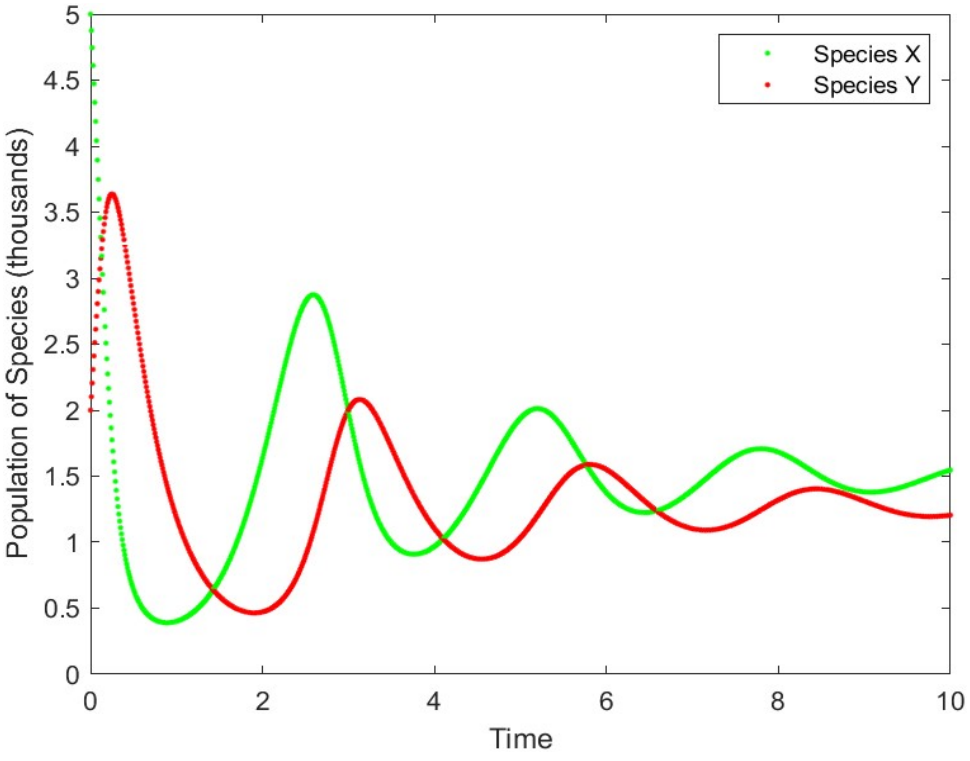
```
if k == t_steps
    x_final = xk;
    y_final = yk;
end
```

This is necessary so that we have variables to print the correct final values. If we did not have this, we would instead have to print the values for `xk` and `yk`, which would have gone through an extra step after they were set equation to `xkp1` and `ykp1`. It is also interesting to note that we could have started the loop at *k* = 1, instead of inserting this `if` statement. However, if we did this, we would not have values on our graph for *t* = 0. Finally, the loop ends, the figure is saved, and the final population values are printed to the screen.

1.3 Calculations and Results

When the program is executed, the following output is printed to the screen and the following figure is created:

```
Final Species Population (thousands)
Species X:  1.546713
Species Y:  1.205382
```



1.4 Discussion

This problem was very straightforward and did not require any advanced problem-solving techniques, other than the knowledge of a few new functions (`ceil`, `plot`, `hold on`, `xlabel`, `ylabel`, `legend`, and `saveas`). Through this assignment, I was able to further my understanding of loops and realize that I could assign values within a loop to points on a graph in order to create "lines". The graph produced shows species X and species Y stabilizing out over time. When the X population is low, the Y population increases and vice versa. This makes sense because the two populations are in competition. For example, if population Y decreases, then population X has less competition. At the same time, however, population Y also has less competition. Therefor, the population of Y should increase, eventually leading to the decrease of population X, and the cycle continues until eventually stabilizing. This stabilization can be shown by the decrease in peak and trough magnitude for both species over time.

Additionally, the time elapsed while running this script was recorded for various values of `delta_t` using the `tic` and `toc` functions. The results are shown below:

Elapsed times for `delta_t`:

`delta_t = 0.01`: Time Elapsed = 10.320649 seconds
`delta_t = 0.03`: Time Elapsed = 5.702631 seconds
`delta_t = 0.05`: Time Elapsed = 3.323247 seconds
`delta_t = 0.1`: Time Elapsed = 2.471652 seconds.
`delta_t = 0.2`: Time Elapsed = 2.034067 seconds.

It is clear that as `delta_t` gets smaller, the time to run the code increases. This is understandable, as there are more points to graph and the `for` loop in the script repeats more times. The time to run the script appears to be exponential as well, since there is a much greater time increase from `delta_t = 0.03` to `delta_t = 0.01` as opposed to the difference between `delta_t = 0.2` and `delta_t = 0.1`.

2 Improved computation of population

2.1 Introduction

The goal of this problem was to modify the code in problem 1 to incorporate an effect of other factors on the species X. More specifically, we needed to incorporate the following factor that takes effect at t_i = 5, and finishes at t = 10 (the same final time value that we are given in problem 1):

$$\alpha(t) = \alpha \times e^{\frac{-(t-t_i)^2}{2\sigma^2}} \tag{5}$$

where σ = -0.4.

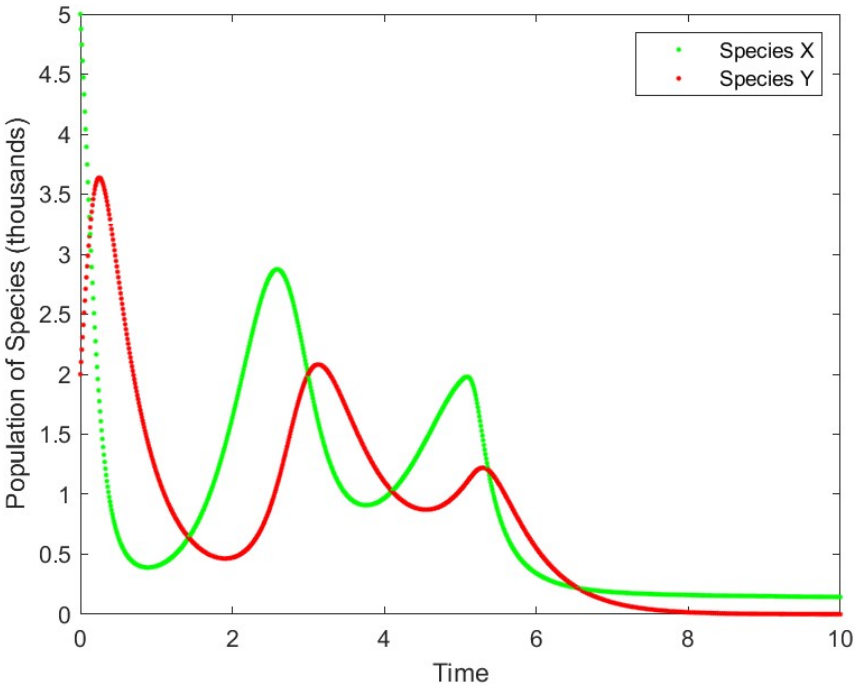
2.2 Models and Methods

First, two new variables `sigma = -0.4` and `t_alphafactorstart = 5`, which represent σ and t_i, respectively, are added to the list of variables at the beginning of the script. Next, a new `if` statement is added after the line that assigns a variable to `t` in the `for` loop. This statement says that if `t` is greater than or equal to `t_alphafactorstart`, then `alpha` is equal to the right side of equation (5). The `if` statement then ends, and the rest of the script is the same.

2.3 Calculations and Results

When the program is executed, the following output is printed to the screen and the following figure is created:

```
Final Species Population (thousands)
Species X:  0.143660
Species Y:  0.000471
```



Since the population is in thousands, the final population of Species Y is 0.471, which is less than 1.

2.4 Discussion

This problem was very simple and straightforward, and did not involve the use of any new functions or ideas. However, the effect of the additional factor on alpha produces interesting results. The graph that is produced is the same as the graph in problem 1 up until the time value equals 5. After the time equals 5, both populations decrease relatively quickly, and the final value of species Y is less than 1. We can assume that this means there is only 1 member of the population left, or the population is extinct. Since this factor affected species X and resulted in both of the populations dropping relatively quick, we can

deduce that species X must have been the prey and species Y must have been the predator. When the population of the prey continued to decrease due to this new factor, so did the predator population.

3 Write a script for permutation

3.1 Introduction

The goal of this problem was to write a script that takes integers *n* and *r* as inputs and outputs the value of $P(n,r)$ from the following equation:

$$P(n,r) = \frac{n!}{(n-r)!} \tag{6}$$

3.2 Models and Methods

The script first uses two calls to the `input` function to obtain values for variables `n` and `r`. Next, the script computes the factorials of *n* and (1 – *r*) and assigns them to the variables `n_factorial` and `nmr_factorial` (nmr means "n minus r"). The factorials are found using the following code:

```
n_factorial = prod(1:n);
nmr_factorial = prod(1:n-r);
```

In this case, the `prod` function is used to multiply all the numbers of an array together. The arrays we created start at 1 and increase in steps of 1 until the value that we are taking the factorial of. This also works when taking the factorial of 0, since the only value in the array will be 1. This code assumes that the user will only enter integers, and that *n* will always be equal to or greater than *r* (there is no indication on the homework that errors are necessary). Next, the value of `n_factorial` is divided by the value of `nmr_factorial` and assigned to the variable `P`, which is then printed to the command window.

3.3 Calculations and Results

When the user inputs a value of 9 for *n* and 2 for *r*, the following result is printed to the screen:

```
Input value for n: 9
Input value for r: 2

P(n,r) = 72
```

Additionally, when the user inputs a value of 1000 for *n* and 0 for *r*, the following result is printed to the screen:

```
Input value for n: 1000
Input value for r: 0

P(n,r) = NaN
```

3.4 Discussion

This problem is very straightforward and allows the use of the `prod` function, along with some knowledge of arrays, in order to write a simple script. There are multiple ways that the factorial of a number can be calculated without using the `factorial` function in MATLAB, however this is the easiest way in my opinion. However, the script produces an interesting result if we enter 1000 for *n* and 0 for *r*. The correct answer is 1, since the numerator and denominator are equal to each other and are real numbers. However, MATLAB says that the results is "NaN", which means "not a number." I assume this is because MATLAB is unable to handle a number as large as the factorial of 1,000, and so it assumes that it is infinite. Then, MATLAB attempts to divide infinity by infinity, which is not a number. One way to fix this specific problem is to create an `if` statement that prints 1 if *r* = 0, and prints the result of the permutation equation if *r* does not equal 0.