Predator-Prey Model

Luke Vaughan
MATH 441 Numerical Analysis
UNC Asheville, Spring 2020

Abstract

Lotka-Volterra equations are systems of differential equations that are used to model predator-prey interactions. In this project, the predators being modeled are foxes, and the prey are rabbits. We can gain fundamental understanding of the predator-prey dynamics of foxes and rabbits by creating a "time history graph," which plots the population numbers for both species (dependent variables) versus time (independent variable). We can gain more understanding by creating a "phase plane," which plots one dependent variable versus another. So phase planes show how the two populations affect each other directly. To create a phase plane, we can implement numerical methods such as Forward Euler, Backward Euler, the Trapezoidal Method, and Newton's Method in R. The goal for this project is to investigate each method in order to find out which one is most effective for analyzing a predator-prey model.

1 Lotka-Volterra Equations

At times, when we study ordinary differential equations, we discuss various methods that can be used to solve the equations analytically. Often, however, we are forced to acknowledge that a differential equation cannot be solved analytically. Some equations are too complex to be dealt with by hand. In turn, we must rely on numerical methods to approximate the solutions.

Lotka-Volterra equations serve as an example of such complexity. These predator-prey models are given as systems of differential equations that cannot be solved analytically. Consider a predator-prey model involving rabbits and foxes. The system takes the form

$$\frac{dR}{dt} = (\alpha - \beta F)R$$
$$\frac{dF}{dt} = (\gamma R - \delta)F$$

where R is the number of rabbits at time t and F is the number of foxes at time t. In the first equation, the "interaction term" $-\beta FR$ is negative because foxes hunt rabbits for food. Similarly, in the second equation, the interaction term γRF is positive because foxes gain nourishment needed to survive by hunting rabbits. If the number of foxes approaches 0, the rabbits thrive, shown by a positive rate of change. For confirmation, plug 0 into the first equation for F, and the equation becomes $dR/dt = \alpha R$. On the other hand, if the number of rabbits approaches 0, the foxes do not thrive. Loss of food causes the derivative to be negative, which can be seen when we plug 0 into the second equation for R, and the equation becomes $dF/dt = -\delta F$.

These observations are significant because they help us gain conceptual understanding of predator-prey dynamics. But if we continue to work with the equations by hand, it will prove to be ineffective. Analytic methods must be abandoned. So we should ask ourselves: what numerical methods are available to us? Some methods are discussed in Section 3.

2 Time History Graph

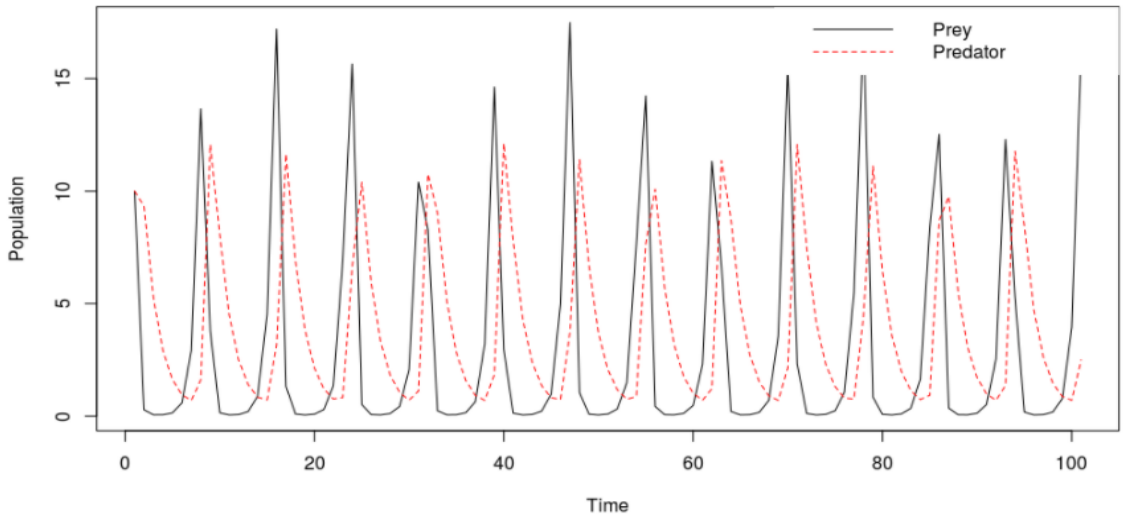
Source [2] provides specific examples of Lotka-Volterra equations, given as

$$\frac{dR}{dt} = \left(2 - \frac{1}{2}F\right) R$$
$$\frac{dF}{dt} = \left(\frac{1}{5}R - \frac{3}{5}\right) F$$

and incorporates the given values for $\alpha, \beta, \gamma, \delta$ in R code. The code is written for the purpose of creating a time history graph. Time history graphs are noteworthy because they are more digestible than phase planes (see Section 4). We can take one look at a time history graph and understand what is going on, whereas, a phase plane might make us scratch our heads at first glance.

Use the following code in R to create the time history graph.

```
install.packages("deSolve")
library(deSolve)

LotVmod = function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    dR = (alpha - beta*F)*R
    dF = (gamma*R - delta)*F
    return(list(c(dR, dF)))
  })
}

Pars = c(alpha = 2, beta = 0.5, gamma = 0.2, delta = 0.6)
State = c(R = 10, F = 10)
Time = seq(0, 100, by = 1)

out = as.data.frame(ode(func = LotVmod, y = State, parms = Pars, times = Time))

matplot(out[,-1], type = "l", xlab = "Time", ylab = "Population")
legend("topright", c("Prey", "Predator"), lty = c(1,2), col = c(1,2), box.lwd = 0)
```



Source [2] did not provide units for the values, so we should make two assumptions. First, the populations are given in thousands of rabbits and foxes. Second, the time is given in years. The initial conditions are $R(0) = 10$ and $F(0) = 10$, so at the beginning of the simulation, there are 10 thousand rabbits and 10 thousand foxes. As the simulation progresses, the rabbit and fox population numbers rise and fall. The foxes hunt the rabbits, and several rabbits die as a result. A few years later, several foxes die of starvation. Then the rabbits reproduce while being hunted less frequently. Finally, the foxes hunt the growing number of rabbits, and the cycle repeats. As mentioned earlier, we can easily see what is going on in the time history graph, and it confirms our expectations.

Note that the time history graph plots foxes and rabbits versus time. A phase plane, on the other hand, would plot foxes versus rabbits, which would show us how the two populations affect each other directly. By creating a phase plane, we could gain an interesting glimpse into predator-prey dynamics. Now that we are interested in creating a phase plane, the need for numerical methods becomes apparent. Three numerical methods are introduced in the following section. The methods are used to approximate the solution to a "toy problem." The three methods are Forward Euler, Backward Euler, and the Trapezoidal Method.

3 Numerical Methods

We require fundamental understanding of numerical methods before we move forward. In situations such as these, it is often helpful to study simple examples ("toy problems"). Consider an example in which we are working with a single differential equation rather than a system of two equations. Pair the differential equation with an initial condition to form the initial value problem

$$\begin{aligned} \frac{dy}{dx} &= y \\ y(0) &= 1 \end{aligned}$$

The exact solution to the initial value problem is $y = e^x$. Use Forward Euler to approximate this curve.

3.1 Forward Euler

Forward Euler, Backward Euler, and the Trapezoidal Method are three algorithms that provide successive points, (x_n, y_n), to approximate a curve that satisfies a differential equation $dy/dx = f(x, y)$. The x-values for the points are predetermined by a stepsize $\Delta x = h$. The y-values for the points are provided by the algorithms in unique ways. Begin with the Forward Euler algorithm. The derivative is approximated as

$$\frac{dy}{dx} = \frac{y_{n+1} - y_n}{\Delta x}$$

Isolate y_{n+1} to derive the Forward Euler algorithm:

$$y_{n+1} - y_n = \Delta x \frac{dy}{dx}$$

$$y_{n+1} = y_n + \Delta x f(x_n, y_n)$$

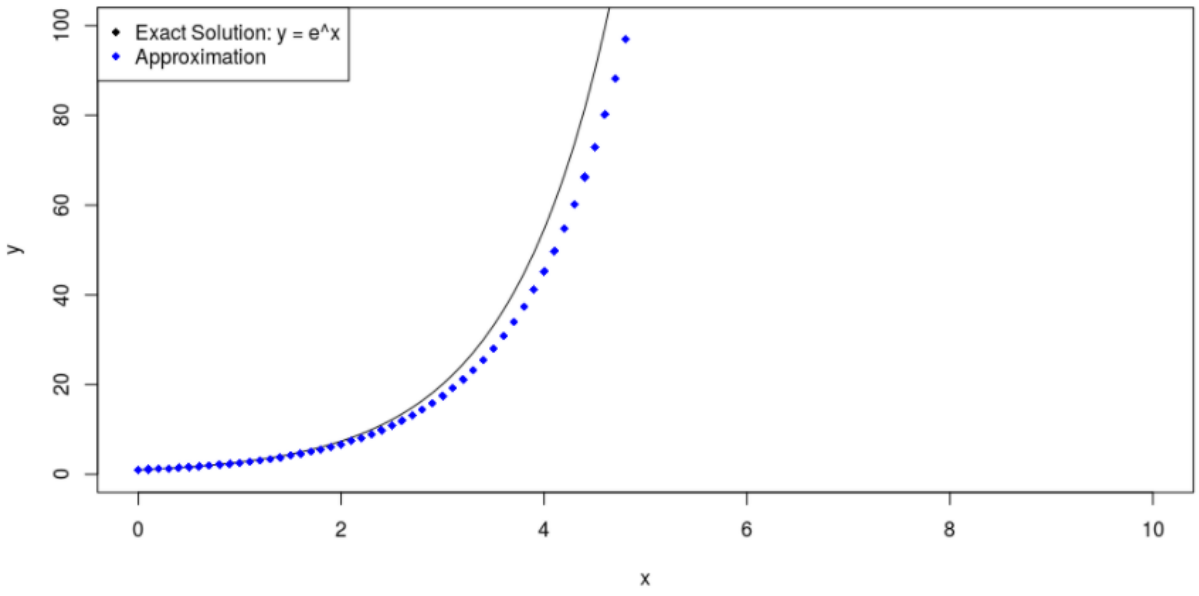
$$\text{where } f(x_n, y_n) \text{ denotes } \frac{dy}{dx} \text{ evaluated at step } n.$$

Note that we were able to solve the equation *explicitly* for y_{n+1}. It only appears on one side of the Forward Euler algorithm.

The following code is written in R for the purpose of implementing Forward Euler.

```r
#Forward Euler: simple example
#dy/dx = y

h = 0.1
x = seq(0,5,by=h)
y = rep(0,length(x))
x[1] = 0
y[1] = 1
slopes = rep(0,length(x))
slopes[1] = 1

diffeq = function(y) {
  return(y)
}

for (i in 1:(length(y)-1)) {
  y[i+1] = y[i] + h*diffeq(y[i])
  slopes[i+1] = diffeq(y[i+1])
}

exactsoln = function(x) {
  return(exp(x))
}

x_other = seq(0,10,by=0.1)
plot(x_other,exactsoln(x_other),type='l',xlab='x',ylab='y',ylim=c(0,100))

points(x,y,col='blue',pch=18)
legend('topleft',c('Exact Solution: y = e^x','Approximation'),pch=c(18,18),col=c('black','blue'))
```

In the following plot, the curve $y = e^x$ is shown alongside its approximation from Forward Euler.



As we can see, the Forward Euler approximation is fairly accurate. In fact, the local error is $\mathcal{O}\Delta x^2$, so Forward Euler is first-order accurate. Note that the numerical method consistently underestimates the y-values as x increases by $\Delta x = h = 0.1$. Next, let us derive Backward Euler and use it to approximate the solution to the same toy problem.

3.2 Backward Euler

This numerical method is quite similar to Forward Euler. The derivation for Backward Euler begins the same way, with the derivative dy/dx being approximated as $dy/dx = (y_{n+1} - y_n)/\Delta x$. Then we attempt to isolate y_{n+1}. However, we evaluate the derivative at a different spot to obtain

$$y_{n+1} = y_n + \Delta x f(x_{n+1}, y_{n+1})$$

$$\text{where } f(x_{n+1}, y_{n+1}) \text{ denotes } \frac{dy}{dx} \text{ evaluated at step } n+1.$$

Note that this equation must be solved *implicitly* for y_{n+1}. Despite our attempt to isolate it, y_{n+1} still appears on both sides of the Backward Euler algorithm. This complicates matters a bit. However, in the toy problem, $f(x_{n+1}, y_{n+1})$ is simply equal to y_{n+1} because the differential equation is given as $dy/dx = y$. Therefore, in this case, we can rearrange the algorithm and isolate y_{n+1} on the left-hand side:

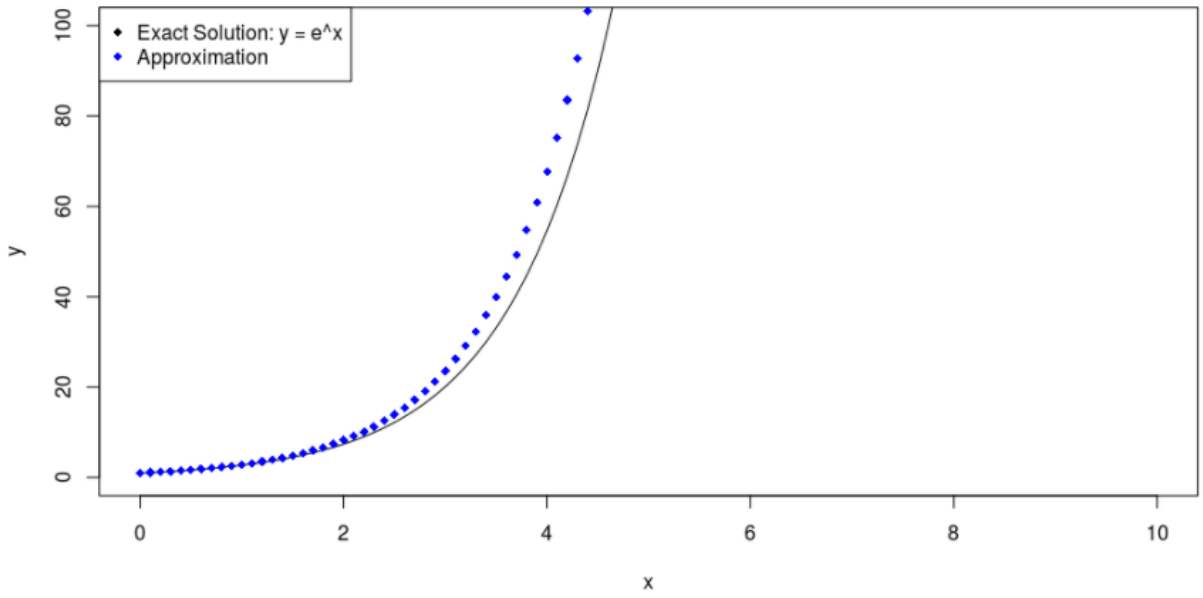
$$y_{n+1} = y_n + \Delta x(y_{n+1}) \implies y_{n+1} - \Delta x(y_{n+1}) = y_n$$

$$y_{n+1}(1 - \Delta x) = y_n \implies y_{n+1} = \frac{y_n}{1 - \Delta x}$$

To implement Backward Euler in R, we can use the previous code for Forward Euler. The only change we need to make is in the loop.

```
for (i in 1:(length(y)-1)) {
  y[i+1] = y[i]/(1-h)
  slopes[i+1] = diffeq(y[i+1])
}
```

The following plot shows the curve $y = e^x$ alongside its new approximation from Backward Euler.



The Backward Euler approximation is also fairly accurate. The local error is $\mathcal{O}\Delta x^2$, so Backward Euler is first-order accurate as well. Note that the numerical method consistently overestimates the y-values as x increases by $\Delta x = h = 0.1$. Finally, let us take one last look at the toy problem, and use the Trapezoidal Method to approximate its solution.

3.3 Trapezoidal Method

This numerical method is basically a combination of Forward Euler and Backward Euler. The Trapezoidal Method algorithm is:

$$y_{n+1} = y_n + \frac{\Delta x}{2}[f(x_n, y_n) + f(x_{n+1}, y_{n+1})]$$

Note that this equation must also be solved *implicitly* for y_{n+1}. It appears on both sides of the Trapezoidal Method algorithm. However, $f(x_{n+1}, y_{n+1})$ is still simply equal to y_{n+1}. Therefore, once more, we can rearrange the algorithm and isolate y_{n+1} on the left-hand side:

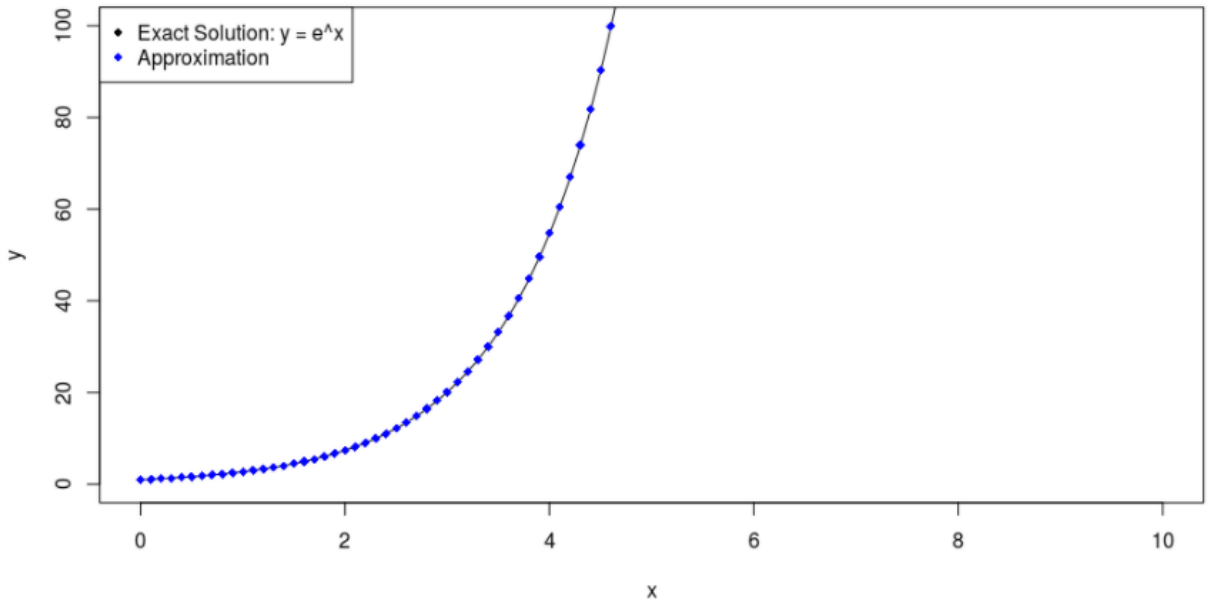
$$y_{n+1} = y_n + \frac{\Delta x}{2} f(x_n, y_n) + \frac{\Delta x}{2} y_{n+1} \implies y_{n+1} - \frac{\Delta x}{2} y_{n+1} = y_n + \frac{\Delta x}{2} f(x_n, y_n)$$

$$y_{n+1}\left(1 - \frac{\Delta x}{2}\right) = y_n + \frac{\Delta x}{2} f(x_n, y_n) \implies y_{n+1} = \frac{y_n + \frac{\Delta x}{2} f(x_n, y_n)}{1 - \frac{\Delta x}{2}}$$

To implement the Trapezoidal Method in R, we can use the previous code for Backward Euler. Once again, the only change we need to make is in the loop.

```
for (i in 1:(length(y)-1)) {
  y[i+1] = (y[i] + (h/2)*(diffeq(y[i])))/(1-(h/2))
  slopes[i+1] = diffeq(y[i+1])
}
```

The following plot shows the curve $y = e^x$ alongside its new approximation from the Trapezoidal Method.



The Trapezoidal Method is clearly the most accurate of the three numerical methods seen thus far. The local error is $\mathcal{O}\Delta x^3$, so the Trapezoidal Method is second-order accurate. Let us keep this accuracy in mind as we move forward.

4 Phase Plane

Now that we have gained fundamental understanding of numerical methods, let us revisit the Lotka-Volterra equations from Section 2. We can use Forward Euler to create a phase plane.

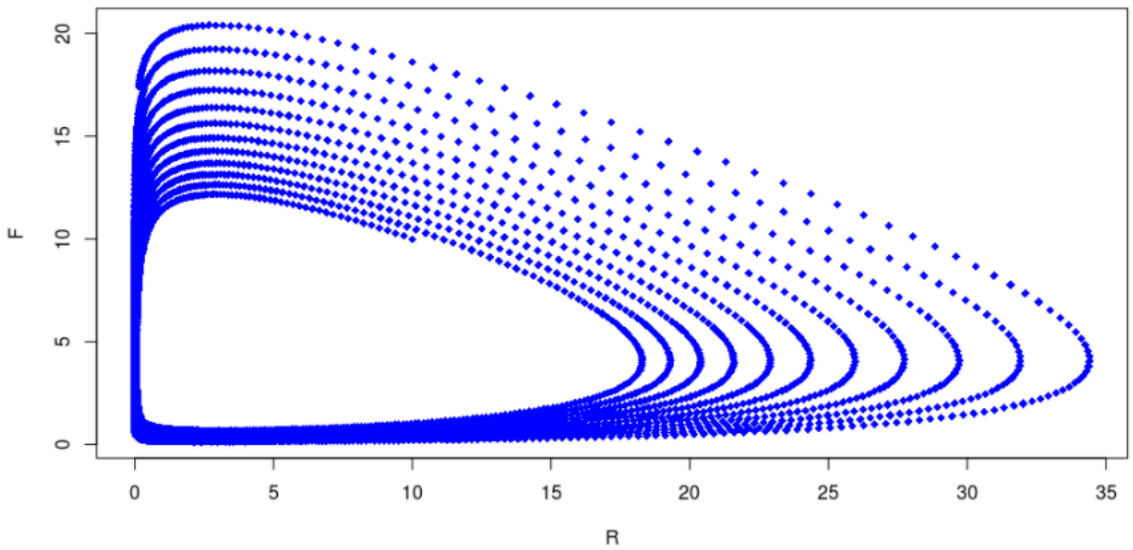
$$R_{n+1} = R_n + \Delta t f(t_n, R_n) \qquad F_{n+1} = F_n + \Delta t g(t_n, F_n)$$

Note that $f(t_n, R_n)$ is the expression on the right-hand side of the first differential equation in the system. Similarly, $g(t_n, F_n)$ is the expression from the second equation in the system. Therefore, in this case, the Forward Euler algorithm becomes the following:

$$R_{n+1} = R_n + \Delta t \left(2 - \frac{1}{2}F_n\right) R_n \qquad F_{n+1} = F_n + \Delta t \left(\frac{1}{5}R_n - \frac{3}{5}\right) F_n$$

Use the following code in R to create the phase plane.

```
deltat = 0.01
tcount = 0
tfinal = 100
n = 10000
R = rep(0,100)
F = rep(0,100)
R[1] = 10
F[1] = 10

while (tcount <= tfinal) {
  for (i in 1:n) {
    R[i+1] = R[i] + (deltat*(2-0.5*F[i])*R[i])
    F[i+1] = F[i] + (deltat*(0.2*R[i]-0.6)*F[i])
    tcount = tcount + deltat
  }
}

plot(R,F,col='blue',pch=18)
```



When we created the time history graph, we worked with a specific predator-prey model with values retrieved from source [2]. Note that the same exact predator-prey model is visualized here in the phase plane. However, the values seen in the two graphs do not quite match. In the time history graph, the maximum population value for rabbits is about 18 thousand, and the maximum population value for foxes is about 13 thousand. But in the phase plane, the maximum value for rabbits is about 34 thousand, and the maximum value for foxes is about 21 thousand. How can this be explained?

Although Forward Euler and Backward Euler are fairly good approximations, the errors (the underestimations and overestimations mentioned earlier) inherent in the numerical methods are to blame. Consider the point (10, 10). At the beginning of the simulation, there are 10 thousand rabbits and 10 thousand foxes. The successive points predicted by the numerical method move counter-clockwise and become more and more extreme. So Forward Euler spirals out over time. Thus, we expect Backward Euler to spiral in over time.

Recall that Forward Euler is explicit and Backward Euler is implicit. As a result, the setup is more complicated for Backward Euler. To handle this, we can utilize Newton's Method for nonlinear systems. So we are using two numerical methods at once.

The strategy is to rearrange the Backward Euler algorithms as follows.

$$R_{n+1} = R_n + \Delta t\left(2 - \frac{1}{2}F_{n+1}\right)R_{n+1} \qquad \Longrightarrow \qquad R_{n+1} - R_n - \Delta t(2R_{n+1}) + \Delta t\left(\frac{1}{2}R_{n+1}F_{n+1}\right) = 0$$

$$F_{n+1} = F_n + \Delta t\left(\frac{1}{5}R_{n+1} - \frac{3}{5}\right)F_{n+1} \qquad \Longrightarrow \qquad F_{n+1} - F_n - \Delta t\left(\frac{1}{5}R_{n+1}F_{n+1}\right) + \Delta t\left(\frac{3}{5}F_{n+1}\right) = 0$$

Denote the expressions on the left-hand sides as f_1 and f_2:

$$\vec{f} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} R_{n+1} - R_n - \Delta t(2R_{n+1}) + \Delta t(\frac{1}{2}R_{n+1}F_{n+1}) \\ F_{n+1} - F_n - \Delta t(\frac{1}{5}R_{n+1}F_{n+1}) + \Delta t(\frac{3}{5}F_{n+1}) \end{bmatrix}$$

Since these expressions are set equal to 0, we are using Newton's Method to find the roots of f_1 and f_2. We make an initial guess, then implement an algorithm until we land within a certain tolerance. We are trying to approximate the roots as accurately as possible. Newton's Method for nonlinear systems is given as

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} - [\vec{f}\,'(\vec{x}^{(k)})]^{-1}\vec{f}(\vec{x}^{(k)})$$

Note that $\vec{f}\,'(\vec{x}^{(k)})$ is the Jacobian matrix of partial derivatives:

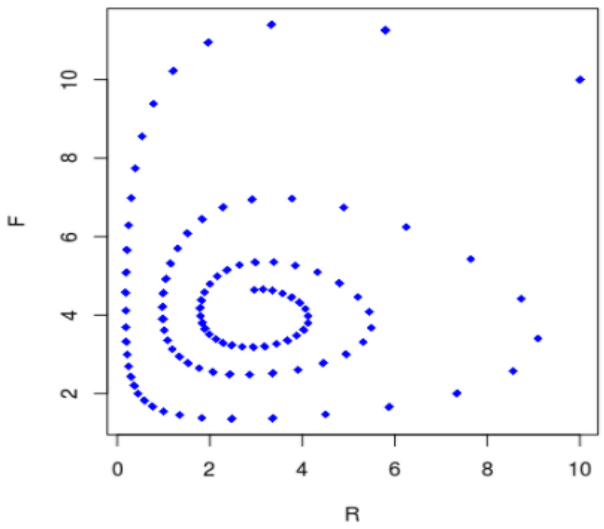
$$\begin{bmatrix} \frac{\partial f_1}{\partial R_{n+1}} & \frac{\partial f_1}{\partial F_{n+1}} \\ \frac{\partial f_2}{\partial R_{n+1}} & \frac{\partial f_2}{\partial F_{n+1}} \end{bmatrix} = \begin{bmatrix} 1 - 2\Delta t + \frac{1}{2}\Delta t F_{n+1} & \frac{1}{2}\Delta t R_{n+1} \\ -\frac{1}{5}\Delta t F_{n+1} & 1 - \frac{1}{5}\Delta t R_{n+1} + \frac{3}{5}\Delta t \end{bmatrix}$$

We assert that the Newton's Method algorithm will iterate until the value of the L2 norm is within the tolerance, denoted as "tol." So we are iterating until

$$\sqrt{f_1^2 + f_2^2} < \text{ tol.}$$

When we implement this in R, we create a function called `f` to represent $\vec{f}$. The function `jacob` represents the Jacobian matrix, and the function `newton` represents the Newton's Method algorithm. The nonlinear system is then solved, and a new phase plane is created. As expected, Backward Euler spirals in over time.

```
#Backward Euler: predator-prey model
#Implement Newton's method for nonlinear systems

deltat = 0.2
results = matrix(c(rep(0,200)),nrow=2,ncol=100)
results[,1] = c(10,10)

f = function(x,x0,deltat) {
  return(c(x[1]-x0[1]-deltat*2*x[1]+deltat*0.5*x[1]*x[2],x[2]-x0[2]-deltat*0.2*x[1]*x[2]+deltat*0.6*x[2]))
}

jacob = function(x,deltat) {
  return(matrix(c(1-2*deltat+0.5*deltat*x[2],-0.2*deltat*x[2],0.5*deltat*x[1],1-0.2*deltat*x[1]+0.6*deltat),2,2))
}

newton = function(x0,tol,deltat) {
  x = x0
  while(sqrt(sum(f(x,x0,deltat)^2))>tol) {
    x = x-solve(jacob(x,deltat))%*%f(x,x0,deltat)
  }
  return(x)
}

for (i in 1:99) {
  results[,(i+1)] = newton(results[,i],0.001,deltat)
}

R = NULL; F = NULL
for (i in 1:100) {
  R[i] = results[1,i]
  F[i] = results[2,i]
}
plot(R,F,col='blue',pch=18)
```



Next, let us try a new approach. We expect the Trapezoidal Method, paired with Newton's Method, to provide the best approximation seen thus far.

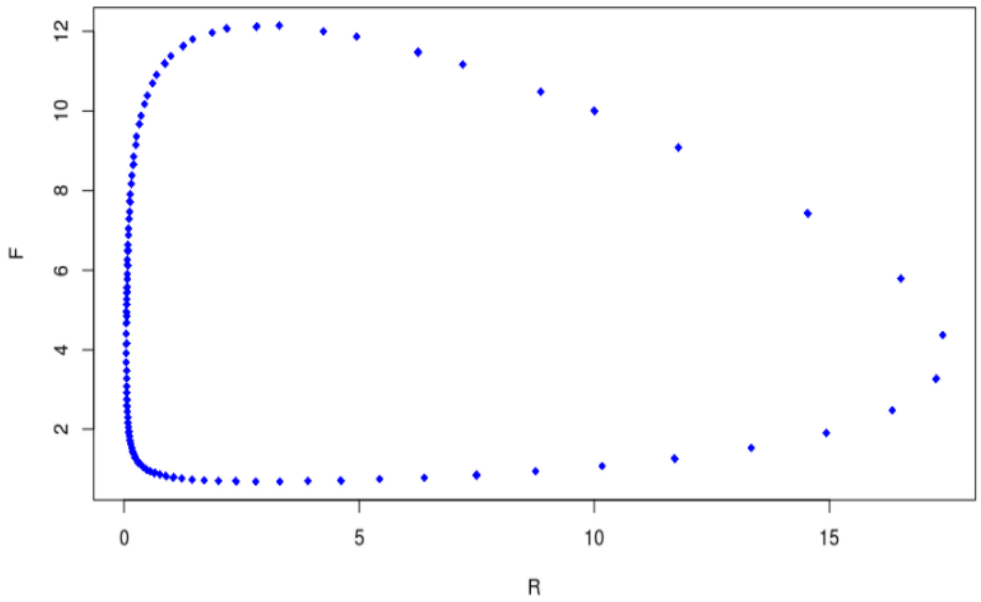
Since the Trapezoidal Method is implicit, we use the same strategy from before (Backward Euler paired with Newton's Method). After we rearrange the equations in order to set the expressions equal to 0, the new $\vec{f}$ is

$$\vec{f} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} R_{n+1} - R_n - \Delta t R_n + \frac{\Delta t}{4} R_n F_n - \Delta t R_{n+1} + \frac{\Delta t}{4} R_{n+1} F_{n+1} \\ F_{n+1} - F_n - \frac{\Delta t}{10} R_n F_n + \frac{3\Delta t}{10} F_n - \frac{\Delta t}{10} R_{n+1} F_{n+1} + \frac{3\Delta t}{10} F_{n+1} \end{bmatrix}$$

The new Jacobian matrix is

$$\begin{bmatrix} 1 - \Delta t + \frac{\Delta t}{4} F_{n+1} & \frac{\Delta t}{4} R_{n+1} \\ -\frac{\Delta t}{10} F_{n+1} & 1 - \frac{\Delta t}{10} R_{n+1} + \frac{3\Delta t}{10} \end{bmatrix}$$

The updated $\vec{f}$ and the updated Jacobian can be used to create a new phase plane in R. The code is omitted for brevity. It is nearly identical to the Backward Euler code, except for some changes made to the `f` and `jacob` function arguments.



5 Conclusion

In Section 2, we described the predator-prey model as a "cycle." Numerical methods generally struggle to remain on a closed orbit since errors are inherent in the approximations. In Section 4, we noticed that Forward Euler spiraled out over time and Backward Euler spiraled in over time. So they both failed to remain on a closed orbit. However, the Trapezoidal Method succeeded. The phase plane portrays a cycle as expected. Also, the values seen in the time history graph and the phase plane actually match. Both graphs predict the same maximum population values for rabbits (18 thousand) and foxes (13 thousand).

Numerical methods are useful for the visualization of predator-prey interactions. The general predator-prey model seen in Section 1 can be applied to other species as well. For example, we can study lions versus zebras, or pandas versus eucalyptus trees. Once we assign values and set up a specific predator-prey model as seen in Section 2, the system of differential equations poses an exciting challenge. If numerical methods are implemented effectively, we can gain intriguing insights into predator-prey dynamics.

6 Sources

1. https://drive.google.com/drive/u/1/folders/1tH8SdEOckeRmL7SUzcr8jNInVJaDBqc0
2. https://rpubs.com/Jeet1994/Prey-predator-model