Chapter 15

Modeling Community Population Dynamics with the Open-Source Language R

Robin Green and Wenying Shou

Abstract

The ability to explain biological phenomena with mathematics and to generate predictions from mathematical models is critical for understanding and controlling natural systems. Concurrently, the rise in open-source software has greatly increased the ease at which researchers can implement their own mathematical models. With a reasonably sound understanding of mathematics and programming skills, a researcher can quickly and easily use such tools for their own work. The purpose of this chapter is to expose the reader to one such tool, the open-source programming language R, and to demonstrate its practical application to studying population dynamics. We use the Lotka–Volterra predator–prey dynamics as an example.

Key words Modeling, R, Lotka–Volterra, Population dynamics, Predator–prey relationship

1 Introduction

Mathematics is integral to the study of biological systems. From the direct application of the Malthusian growth model [1] to abstraction from Fibonacci number series, mathematical models can help researchers explain natural phenomena quantitatively and generate new hypotheses better than with only experimental observations. After translating a biological problem into a set of mathematical equations, solutions can be sought and visualized.

Perhaps one of the most popular tools for such analysis is the open-source language and computing environment R (r-project.org). First developed by Ross Ihaka and Robert Gentleman at the University of Auckland in 1993, R is part of the Free Software Foundation's GNU Project, a massive collaborative effort meant to develop high quality open-source software (http://www.fsf.org/). R offers users a plethora of standard statistical and computational tools, extensive collections of predefined functions, and a well-maintained and documented support system. In addition to the preexisting functionalities, R also allows users to define their own functions and algorithms. Syntax (jargon for rules and structure of the programming language) of R is also relatively easy to understand.

Here we will demonstrate how R can be used to express and analyze mathematical models of population dynamics. We describe the Lotka–Volterra equations for representing population dynamics between predator and prey. We then present a step-by-step guide to getting set up to use the R environment, and an easy-to-follow implementation of the above model in R. By the end of this chapter, the reader will have a basic understanding of how to implement and numerically solve a mathematical model based on differential equations, visualize the solutions, and explore different permutations to formulate new hypotheses.

Disclaimer: The reader should note that this chapter is not intended to give a full background or tutorial on R. For a more comprehensive introduction to R, please *see* ref. [2]. It should also be noted that R, at its current stage, may have a slower performance than other languages for specific types of problems.

2 Background: The Lotka–Volterra Equations

A fundamental phenomenon in population ecology is predation, the feeding of one organism (the predator) on another (the prey). In 1926, the biophysicist Alfred Lotka proposed a mathematical model [3] to represent this relationship. The Italian mathematician Vito Volterra explored this relationship independently of Lotka [4]. This has led to the proposal of the *Lotka–Volterra equations*:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = Ax - Bxy. \tag{1}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = Cxy - Dy. \tag{2}$$

where

- $x(t)$ is population density of the prey at time t.
- $y(t)$ is the population density of the predator at time t.
- A is net birth rate (natural birth rate subtracting natural death rate, in the unit of per time unit) of the prey population in the absence of predator.
- B is the rate at which prey are killed due to the presence of predator (in the unit of per time unit per predator density).
- C is the birth rate of the predator population due to the presence of prey (in the unit of per time unit per prey density).
- D is the death rate of the predator population in the absence of prey.

- dx/dt and dy/dt are the rates of change of x and y, respectively.

Intuitively, the reader can think of the two differential equations as:

The rate at which the prey population changes is the birth rate of the prey minus the rate of consumption of the prey by the predator

and

The rate at which the predator population changes is the birth rate of the predator (which is dependent on the amount of prey present) minus the death rate of the predator.

Thus these rates are dependent on the densities of both the predator and the prey populations, in addition to parameters which are static in this model.

It is important to note that this model does make assumptions that might not necessarily be true:

- There is an ample source of food for the prey at all times.
- The predator population only feeds on the prey population (no other source of food) and feeds continuously.
- There is infinite space to hold both predator and prey populations.
- The rate of change of the population is proportional to its density.
- The interactions between predator and prey are determined by the product of the density of the two populations, much like in the collision of two reactants in concerted bimolecular reactions. There are no spatial refuges for prey.

From Eqs. 1 and 2 at time $t' = t + \mathrm{d}t$, where dt approaches 0,

$$x\left(t'\right) = x\left(t\right) + \left(\frac{\mathrm{d}x}{\mathrm{d}t} \times \mathrm{d}t\right) = x\left(t\right) + \left(Ax - Bxy\right) \times \mathrm{d}t. \tag{3}$$

$$y\left(t'\right) = y\left(t\right) + \left(\frac{\mathrm{d}y}{\mathrm{d}t} \times \mathrm{d}t\right) = y\left(t\right) + \left(Cxy - Dy\right) \times \mathrm{d}t. \tag{4}$$

Once the initial ($t = 0$) values for $x(t)$ and $y(t)$ are known, the prey and predator populations densities $x(t)$ and $y(t)$ can be computed for any t.

For pedagogical purposes, we first present an R implementation based on Euler (first-order) approximations in Eqs. 3 and 4 to estimate population dynamics corresponding to Eqs. 1 and 2. We will then present a more practical and efficient implementation using the *deSolver* package to solve Eqs. 1 and 2. For a more complete overview, please *see* ref. 5.

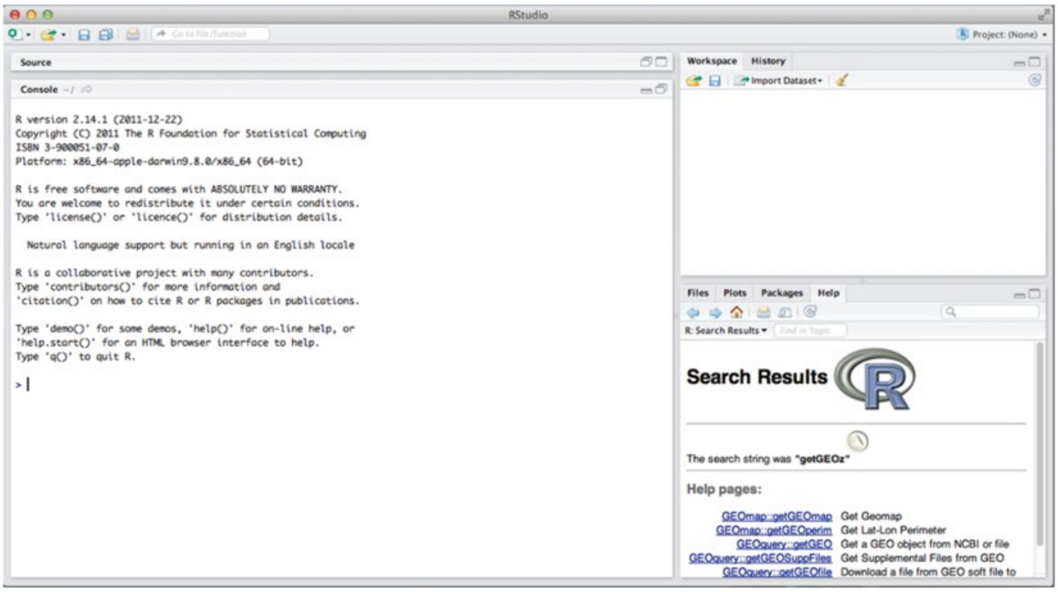
3 Getting Started with R

For this chapter, the authors ran all simulations in RStudio (www.rstudio.com), an open-source environment for R computing. While certainly not the only environment available, RStudio is simple and provides an integrated environment for basic computation, writing scripts, and visualizing data in addition to up-to-date documentation on various aspects of the language.

To install RStudio, please visit www.rstudio.com/ide/download/desktop and download the package most suitable for your operating system (the authors recommend you select the version under "Recommended For Your System" at the top of the page). Once the package is downloaded, click on the file and follow the on-screen instructions to install all files in the proper directories. Depending on where you chose to install RStudio, the graphical user interface (GUI) icon should appear in that directory. Click on it and you should see a screen similar to the one below (*see* Fig. 1):

The first thing to do is to familiarize yourself with the environment. The "Console" window is where you can perform simple computations, call scripts and functions, and create variables for later use. For example, in the "Console" window, type the following commands (">" automatically appears in the Console for a new command) (*see* Fig. 1):

```
>a<-2
>b<-50
```



```
R version 2.14.1 (2011-12-22)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

The search string was **"getGEOz"**

Help pages:

GEOmap::getGEOmap	Get Geomap
GEOmap::getGEOperim	Get Lat-Lon Perimeter
GEOquery::getGEO	Get a GEO object from NCBI or file
GEOquery::getGEOSuppFiles	Get Supplemental Files from GEO
GEOquery::getGEOfile	Download a file from GEO soft file to

Fig. 1 New RStudio session

This is called "defining a variable." Basically, you are telling the current environment that the character *a* now holds the value (in R, this value is called a "numeric") 2, and the character *b* holds the value '50. Next, type the following command (note the quotation marks):

```
>word<-'airplane'
```

The assignment of variable names is not limited to single letters or symbols (something we will exploit later in our code). In this case, you are telling the environment that *word* holds the string of characters (in R, this is known simply as a "character") that make up "airplane".

These variables that you have created can be manipulated. Type the following commands:

```
> a+b
[1] 52
> a-b
[ 1] -48
> a*b
[1] 100
> b/a
[1] 25
>
```

Your screen should show the values 52, –48, 100, and 25. As you can see, simple arithmetic operations can be performed with the newly created variables.

Next, the user should become familiar with their working directory. In your console, type the following command to get the current working directory:

```
> getwd()
```

You should see something equivalent to the following:

```
> getwd()
[ 1] "/Users/user1"
```

This location can be changed with the following command:

```
> setwd('path/to/desired/directory' )
```

To save your work to the current working directory, type the following command:

```
> save.image()
```

This will create a file called ".RData" (this will be a hidden file in most directories). To load the ".RData" file for future use, type the following command:

```
> load('.RData' )
```

Alternatively, RData can be files that are explicitly named during the save process, which also makes them visible in directories:

```
>save.image('LotkaVolterraExample.RData')
>load('LotkaVolterraExample.RData')
```

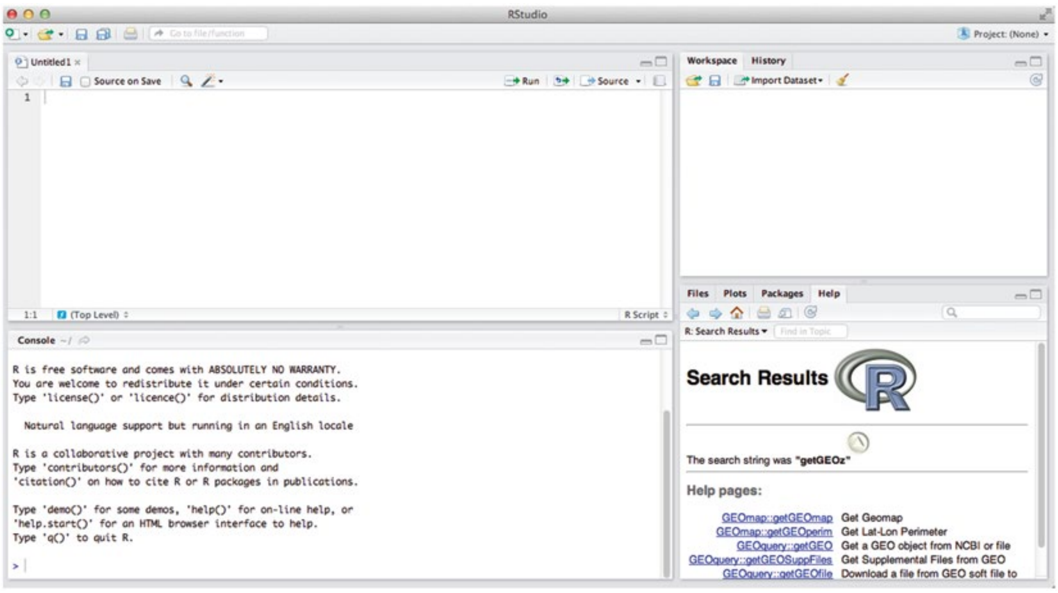
Use the up arrow key to return to the previous command, and repeat this process to access earlier commands. Now that you are familiar with the basics, you are ready to begin implementing the Lotka–Volterra equations.

4 Implementation

To get started, click on the "File" button in the top left-hand corner of the screen and select "File->New->R-Script". This should be a drop down screen in RStudio that looks something like the following (Fig. 2):

In this new window, type the following (Fig. 3):

This is the framework that will contain the *function* that you will write. Inputs to a function can be put inside ( ). Next, set up your environment to easily save and run your work. To "tell" the R environment to use the code you have written, you may *"source"* your script by selecting "Source on Save" from the top left corner of your window, as shown below (Fig. 4):



```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

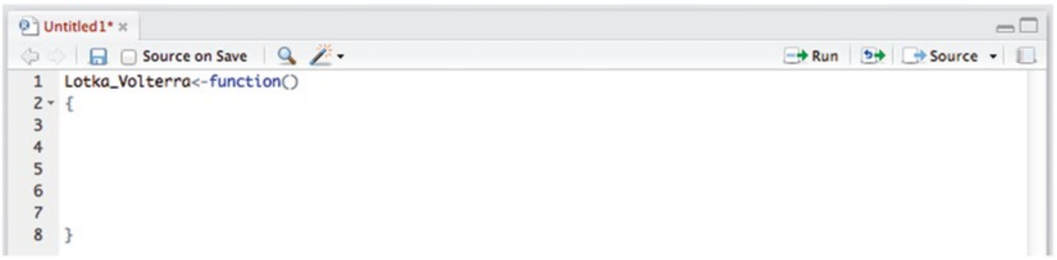
Search Results

The search string was **"getGEOz"**

Help pages:

GEOmap::getGEOmap	Get Geomap
GEOmap::getGEOperim	Get Lat-Lon Perimeter
GEOquery::getGEO	Get a GEO object from NCBI or file
GEOquery::getGEOSuppFiles	Get Supplemental Files from GEO
GEOquery::getGEOfile	Download a file from GEO soft file to

Fig. 2 New RScript window

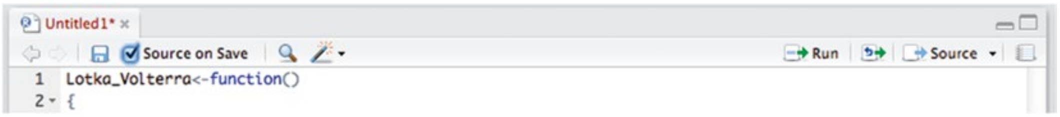


```
Lotka_Volterra<-function()
{



}
```

Fig. 3 New Lotka–Volterra function



```
Lotka_Volterra<-function()
{
```

Fig. 4 Source on save command

Sourcing can be thought of as a way of making the R environment reevaluate the file/script in question. By sourcing the file/script, you are telling the R environment to execute the file/script, which can either result in running a program or in this case, updating a function. Next, click on the floppy disc icon to save your script as, for example, LV_Example. Since you are saving your script for the first time, name the file as you wish (it will be saved as a .R file). This file will be saved in your current working directory. When you save your file, on your Console screen you should see something similar to the following output:

```
> source('~/LV_Example.R')
```

This means your function is now "ready" to be called in your environment. Add the following print statement to your function in the scripting window and save/source your script (Fig. 5):

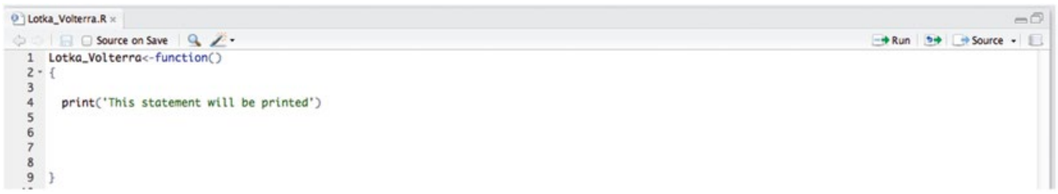
Now "call" your function by typing the following:

```
> Lotka_Volterra()
```

You should see the following on your screen:

```
[1] "This statement will be printed"
```

Note that we added the print command in between the two curly braces ("{ }") of the function. Anything written in between these curly braces will only be executed when the function is called. If the same print command was instead placed outside the braces, then the command would be executed every time the file is sourced.



```
Lotka_Volterra<-function()
{

  print('This statement will be printed')




}
```

Fig. 5 Adding a print command

Next, we will describe how to pass variables to your function. Add the following commands to your function:

```
Lotka_Volterra<-function(x_start)
{
   print('This statement will be printed')
   print(x_start)
}
```

Now, call your function like before. You will notice an error message:

```
> Lotka_Volterra ()
[1] "This statement will be printed"
Error in print(x_start): argument "x_start" is
missing, with no default
```

Because no value was passed to the function, yet the function required a value to be assigned to "x_start", we got an error message. This can be solved in two ways. The first is to explicitly pass a value to the function:

```
> Lotka_Volterra(8)
[1] "This statement will be printed
[1] 8
> Lotka_Volterra(x_start=8)
[1] "This statement will be printed"
[1] 8
```

Or, alternatively, you can define a *default value* for "x_start" in your RScript function, shown below:

```
Lotka_Volterra<-function(x_start=10)
{
```

This means that if no value is passed to the function, "x_start" will automatically be assigned to 10. However, explicitly passing a value to the function overrides this:

```
> Lotka_Volterra()
[1] "This statement will be printed"
[1] 10
> Lotka_Volterra(8)
```

```
[1] "This statement will be printed"
[1] 8
> Lotka_Volterra(x_start=8)
[1] "This statement will be printed"
[1] 8
```

It may be easier to define default parameters for the purpose of this program; otherwise calling the function each time will require you to specify every parameter in the console, which can be onerous.

Modify your function as follows (be sure to delete the two previous print statements you were using within your function, as they are no longer needed):

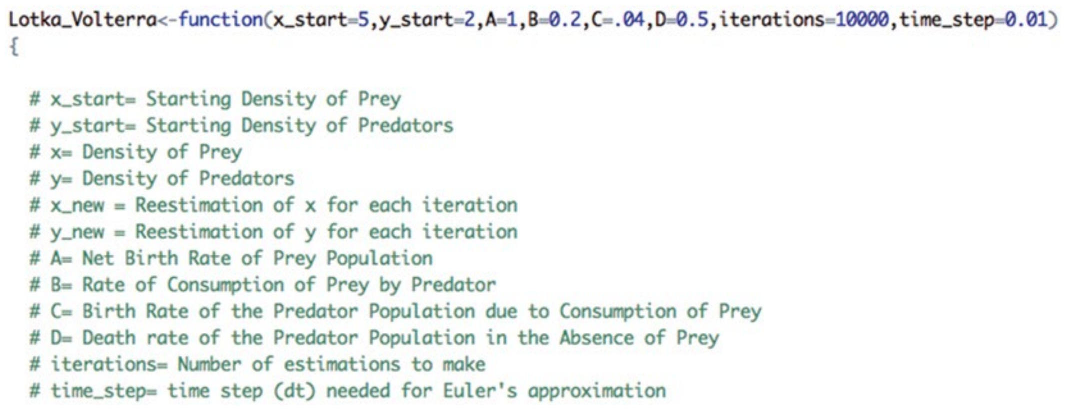
```
Lotka_Volterra<-function(x_start=5,y_start=2,A=
1,B=0.2,C=0.04,D=0.5,iterations=10000,t
ime_step=0.01)
```

These will be the default parameters for running the simulations. It may be useful to keep a record of what each of these parameters represents when writing the code. This can be accomplished by *commenting* our code. Comments are not "read" by the program when running, so they only serve as documentation for the programmer (or someone else reading the code in the future). Commenting in R can be accomplished by adding "#" at the beginning of a line in our script. We can use comments to add notes about our code, as shown below (Fig. 6):

Next, add the following commands to your code:

```
x<-x_start
y<-y_start

dt<-time_step
graph_frame<-c(0,x,y)
```



```
Lotka_Volterra<-function(x_start=5,y_start=2,A=1,B=0.2,C=.04,D=0.5,iterations=10000,time_step=0.01)
{

  # x_start= Starting Density of Prey
  # y_start= Starting Density of Predators
  # x= Density of Prey
  # y= Density of Predators
  # x_new = Reestimation of x for each iteration
  # y_new = Reestimation of y for each iteration
  # A= Net Birth Rate of Prey Population
  # B= Rate of Consumption of Prey by Predator
  # C= Birth Rate of the Predator Population due to Consumption of Prey
  # D= Death rate of the Predator Population in the Absence of Prey
  # iterations= Number of estimations to make
  # time_step= time step (dt) needed for Euler's approximation
```

Fig. 6 Documenting the code

The first two commands simply assign the values of "x_start" and "y_start" to the new variables "x" and "y", respectively. The third command sets the time step for simulation "d*t*" to the value stored in "time_step." The last command creates a *vector* called "graph_frame." The vector is one of the most important data types in R. Intuitively, it can be thought of as a collection of variables held together in a searchable container (similar to lists in Python or arrays in C++). Essentially, we have created a 1×3 table of values, which will be expanded to store the results of our code. The first column will hold information on time (which starts at 0 and is in an arbitrary unit), the second and third columns a will respectively hold the population densities of prey and predator corresponding to the time point in the first column.

Next, add the following to your code, just below the above commands:

```
for( i in 1:iterations)
{



}
```

This is known as defining a *loop*. A loop is a section of code that is repeated until some criterion tells the loop to stop or a command is explicitly given to break out of the loop. In our case, we have qualified that our loop should continue until the "for" statement is no longer true. In our "for" statement, we are implicitly defining a vector containing all the numbers from one to *iterations*. Intuitively, this can be thought of as saying:

Repeat the loop until the value of *i*, which iterates through a vector of all the number of iterations, is equal to the number specified in the variable *iterations* (in this case, has reached the end of the vector).

It is important to note that the value of *i* will increase by one as it traverses the vector of values from one to the number of iterations. To test this, add the following print command to your loop:

```
for( i in 1:iterations)
{
   print(i)

}
```

If you source your code and call the function, you'll notice that, new lines of numbers ranging from 1 to 10,000 appear on your console, demonstrating that "*i*" is being increasing while the code stays within the loop. Now you have the entire framework in place to begin modeling with the Lotka–Volterra equations.

Start by adding the code required to update the population densities of predator and prey at each iteration:

```
for( i in 1:iterations)
{

    dx<-(A*x-B*x*y)*dt
    dy<-(C*y*x-D*y)*dt

    x_new<-x+dx
    y_new<-y+dy

    print(i)

}
```

Within each iteration, dx and dy calculate changes in population densities of x and y according to Eqs. 3 and 4, respectively. New values of x and y after time dt are evaluated and stored in variables "*x_new*" and "*y_new*", respectively.

We then add the following lines in the loop after the definition of *x_new* and *y_new* inside the loop:

```
new_data<-c(i*dt,x_new,y_new)

graph_frame<-rbind(graph_frame,new_data)
```

The first line stores three pieces of information (time elapsed $i \times \mathrm{d}t$, and the values of "*x_new*" and "*y_new*" at time $i \times \mathrm{d}t$) in a vector called *new_data*. The next line adds this *new_data* to our *graph_frame* as a new row through a command called *rbind* (i.e. row-bind), expanding an original $N \times 3$ table to an $(N+1) \times 3$ data vector (which can also be referred to as a *matrix*).

Finally, update the values of "*x*" and "*y*" for the next iteration:

```
x<-x_new
y<-y_new
```

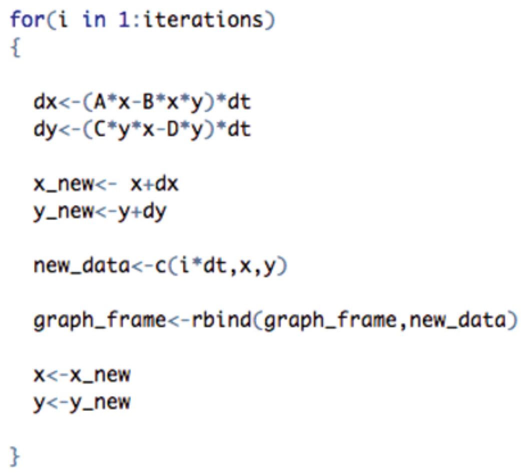
Delete the "print(i)" at the end of the code and your loop should loop like the one below (Fig. 7):

Thinking about the code within the loop logically, we can summarize the sequence of events as follows:

For each iteration/given unit of time:

- estimate changes in the values of x and y from the last iteration based on the Lotka-Volterra equation
- apply these changes to x and y to get new estimates
- add these new estimates and their corresponding time stamp to our overall vector
- update the values of x and y with new estimates

The result of this loop is an $N \times 3$ table/matrix of points, with each of the N rows containing a time point, and an estimate of the numbers of prey and predator for that time point.



```
for(i in 1:iterations)
{

  dx<-(A*x-B*x*y)*dt
  dy<-(C*y*x-D*y)*dt

  x_new<- x+dx
  y_new<-y+dy

  new_data<-c(i*dt,x,y)

  graph_frame<-rbind(graph_frame,new_data)

  x<-x_new
  y<-y_new

}
```

Fig. 7 Final loop code

These data can also be visualized easily in RStudio by using the "matplot" function to plot columns in a given vector against one another. Add the following to your code outside of the loop after all iterations:

```
matplot(x=graph_frame[ ,1],y=graph_frame
[,c(2,3)],pch=20)
```

Notice how we have selected which columns to use in the matplot function. Multidimensional vectors are indexed row by column. To select everything in the first column (for the *x*-axis), we leave the first entry in brackets empty and select "1" for the second entry. To select both the second and third columns to be plotted against the first column, we use a vector containing the columns we want, in this case "2" and "3". The "pch" command is simply specifying how the data should be represented, with different values specifying different shapes. Here, *pch* = 20 represents solid circles.

Now call your function. Something similar to the below should appear in the bottom right of your screen (Fig. 8):

This plot can be made more readable with modifications to the matplot function. For example, each axis can be labeled and a title can be added as follows:

```
matplot(x=graph_frame[,1],y=graph_frame[,c(2,3)
    ],pch=20,xlab='Time',ylab='Population
  Density',main='Lotka-Volterra Simulation')
```

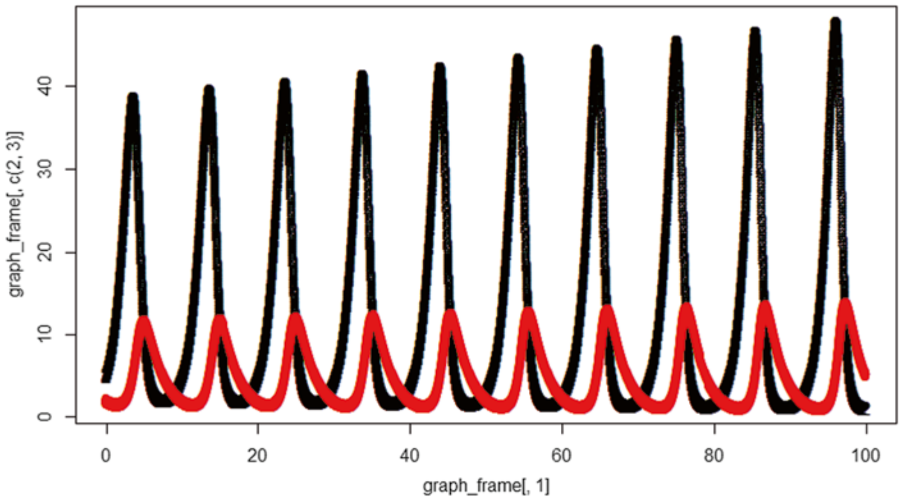


Fig. 8 Initial simulation result

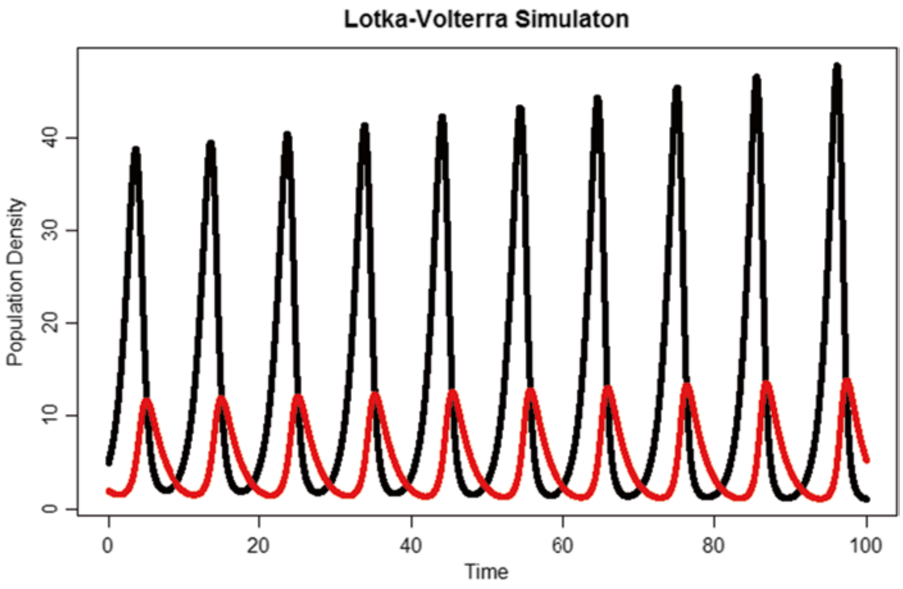


Fig. 9 Initial simulation result

This should give you a plot like this (*see* Fig. 9):

The colors of the plots can be changed as well by specifying with the "col" option in matplot (*see* Fig. 10):

```
matplot(x=graph_frame[,1],y=graph_frame[,c(2,3)],
     pch=20,xlab='Time',ylab='Population
   Density',main='Lotka-Volterra Simulation',
         col=c('darkgreen','darkblue'))
```

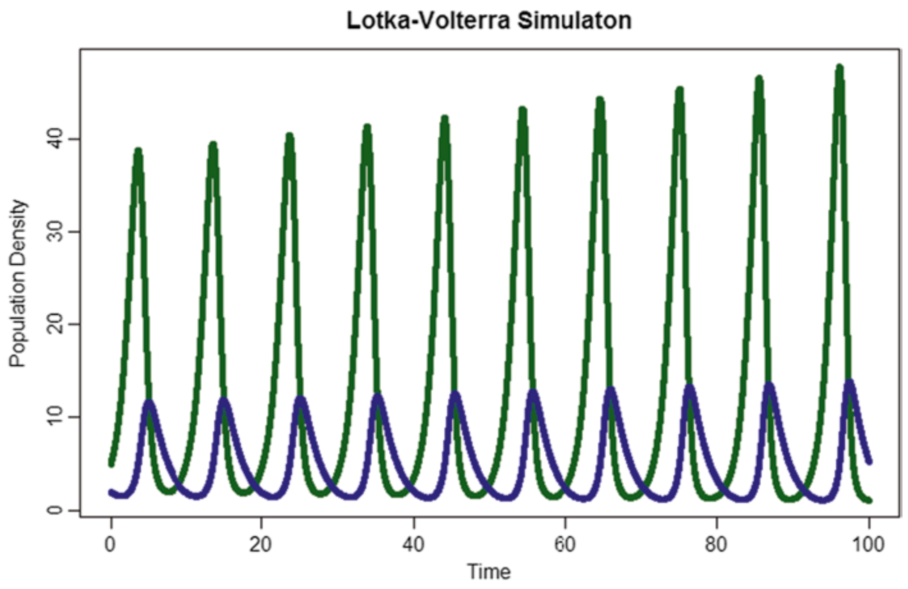


Fig. 10 Results with new colors

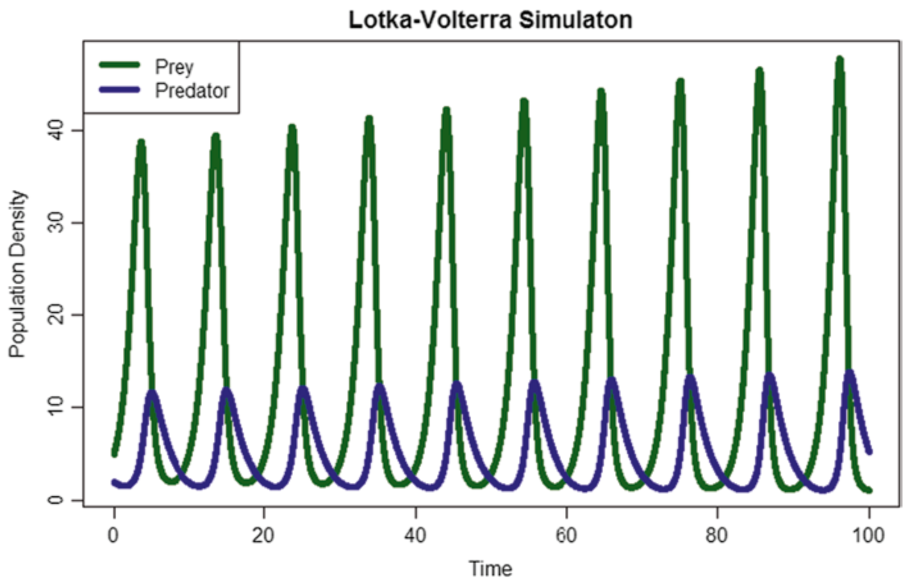


Fig. 11 Result with legend

Finally, a legend can be added to the plot for convenience (note that this is separate from the matplot function) (*see* Fig. 11):

```
legend(x='topleft',legend=c('Prey','Predator'),
    col=c('darkgreen','darkblue'),lwd=5)
```

At this point, you can delete the any early print commands, as they are not needed for further work with the program. Note: For certain calculations, loops may not be the most time-efficient method. Unless storage for the values generated in a loop are pre-allocated, calculations typically take less time using slightly more complex built-in functions in R like the apply function, which applies a given calculation over a table of values (also known as matrix calculation). For more information see the R help (type '?") for 'for', 'vector', 'matrix' and 'apply'.

5 Analysis of the Lotka–Volterra Model

Perhaps one of the most striking characteristics of the plot is its oscillatory nature. At periodic intervals, the prey and predator populations will dramatically rise and fall but out-of-phase with each other. Notice the rapid increase in the prey population density just prior to a rapid rise in the predator population density, followed by a sudden drop in the prey population density which in turn causes a drop in the predator population density. Biologically, this pattern can be explained:

For some given period of time:

- a low predator population density (which is the result of a low prey population density i.e. lack of food supply) allows the prey population to expand through reproduction with little predation
- the sudden increase in prey population density now results in an increased food supply for the predators, which allows the predators to consume and reproduce more, resulting in a larger predator population density and lower prey population density
- when the prey population density decreases, the predators no longer have a food supply to sustain their population, so they begin to die and decrease in population density

This cycle repeats infinitely in this model.

An important trend to note is the increasing peak heights of both the predator and prey populations, which turns out to be an artifact of the estimation in the code. The Euler approximation is a somewhat crude method for solving differential equations and as such will result in some error associated with any solution. To improve the accuracy of these estimations, the time step for each iteration could be decreased, which results in diminishment of this artifact. However, to calculate the solution for the same amount of time, the number of iterations must be increased (since time is calculated by multiplying the time step by number of iterations).

6 Incorporating Alternative Assumptions to the Lotka–Volterra Model

As mentioned above, the Lotka-Volterra equations are based on assumptions about the nature of the predator–prey interactions and the environment in which they occur. Arguments could be made about the validity (or lack thereof) of such assumptions. Therefore, it is useful to explore different possible models based alternative assumptions. For example, let's assume the following:

- Rather than having access to an infinite amount of space and resource, there is a carrying capacity, K, for the prey population.
- The predator population does not necessarily infinitely feed on the prey population. Rather, if there was an abundance of prey available, the predator population would become sated prior to eating all the prey.

To incorporate these assumptions into the model, we can modify our two previous differential equations:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = Ax\left(1 - \frac{x}{K}\right) - y \times Bf\left(x\right). \tag{5}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = y \times Cf\left(x\right) - Dy. \tag{6}$$

where

- K is carrying capacity of the environment for the prey population. Notice how when x is small, the growth rate of prey approaches its maximum value A and as x approaches K (the total number of prey gets closer to its carrying capacity), the population will grow at a near-zero rate (indicative of competition within the prey population for food).
- $Bf(x)$ is the prey-consumption rate per predator density, which is represented by,

$$Bf\left(x\right) = \frac{Bx}{\alpha + x}. \tag{7}$$

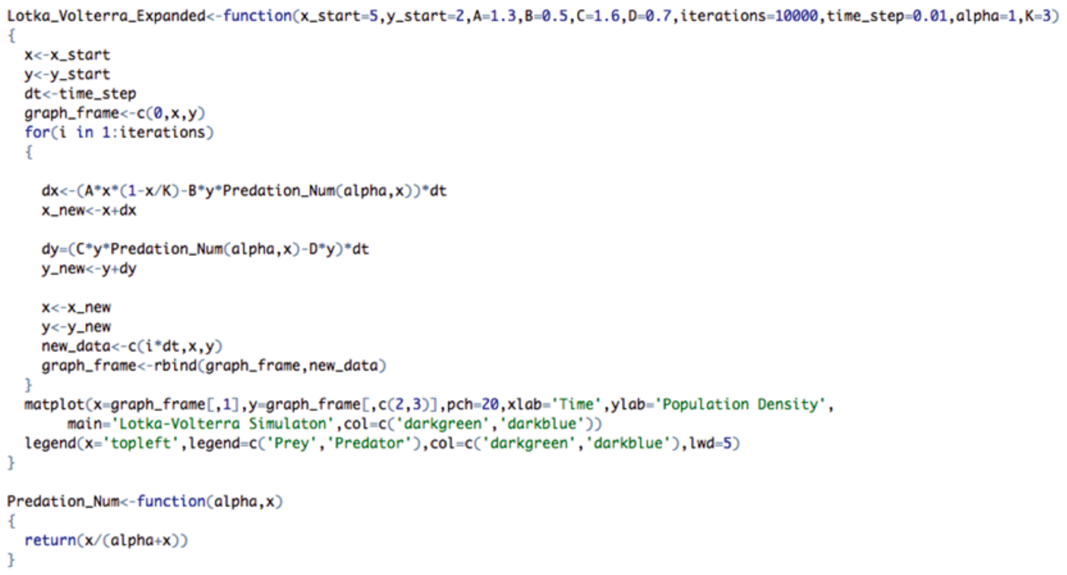
where B is the maximum prey-consumption rate per predator density (per time unit per predator density), α is the prey population density at which half maximal prey-consumption rate per predator density is achieved. Notice when x is large, $Bf(x)$ saturates at B; when x is small, $Bf(x)$ increases almost linearly with x.

Our code can be modified, as shown below, to incorporate these changes (Fig. 12):

Note that in the above implementation we have updated some of the initial parameters, added the K and α parameters, and are now calling a new function *Predation_Num* outside our loop.

Running the above code gives the following result (Fig. 13):

Notice that the prey population oscillations never exceed the initial population density. Because the prey population is limited, the predator prey is also limited in how large it can grow. We could hypothesize that increasing the carrying capacity of the environment for the prey population could mitigate this effect (Fig. 14):



```
Lotka_Volterra_Expanded<-function(x_start=5,y_start=2,A=1.3,B=0.5,C=1.6,D=0.7,iterations=10000,time_step=0.01,alpha=1,K=3)
{
  x<-x_start
  y<-y_start
  dt<-time_step
  graph_frame<-c(0,x,y)
  for(i in 1:iterations)
  {

    dx<-(A*x*(1-x/K)-B*y*Predation_Num(alpha,x))*dt
    x_new<-x+dx

    dy=(C*y*Predation_Num(alpha,x)-D*y)*dt
    y_new<-y+dy

    x<-x_new
    y<-y_new
    new_data<-c(i*dt,x,y)
    graph_frame<-rbind(graph_frame,new_data)
  }
  matplot(x=graph_frame[,1],y=graph_frame[,c(2,3)],pch=20,xlab='Time',ylab='Population Density',
       main='Lotka-Volterra Simulaton',col=c('darkgreen','darkblue'))
  legend(x='topleft',legend=c('Prey','Predator'),col=c('darkgreen','darkblue'),lwd=5)
}

Predation_Num<-function(alpha,x)
{
  return(x/(alpha+x))
}
```

Fig. 12 Alternative Lotka–Volterra code

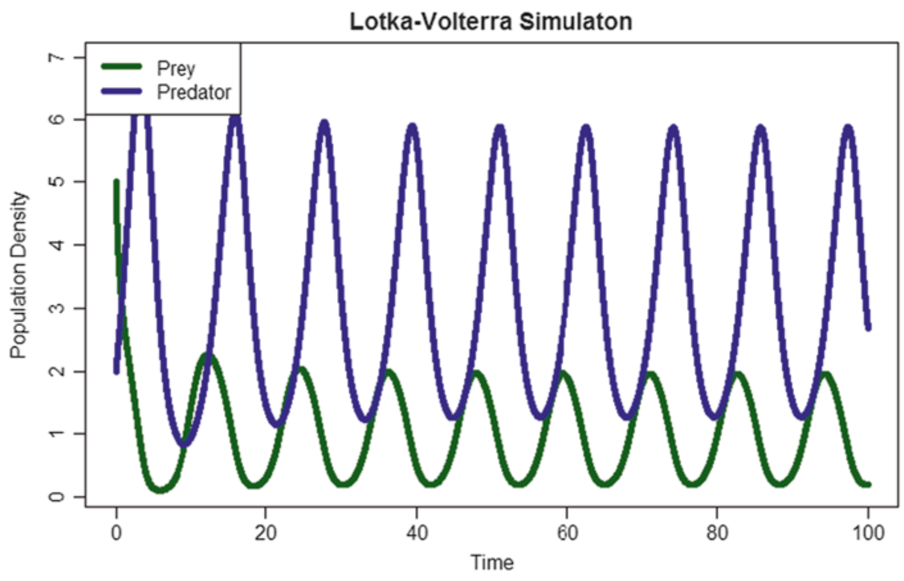


Fig. 13 Alternative Lotka–Volterra model result

```
Lotka_Volterra_Expanded<-function(x_start=5,
y_start=2,A=1.3,B=0.5,C=1.6,D=0.7,iterations=
      10000,time_step=0.01,alpha=1, K=10)
```

Because the prey population is allowed to grow to a higher carrying capacity limit, the maxima of both populations become larger.

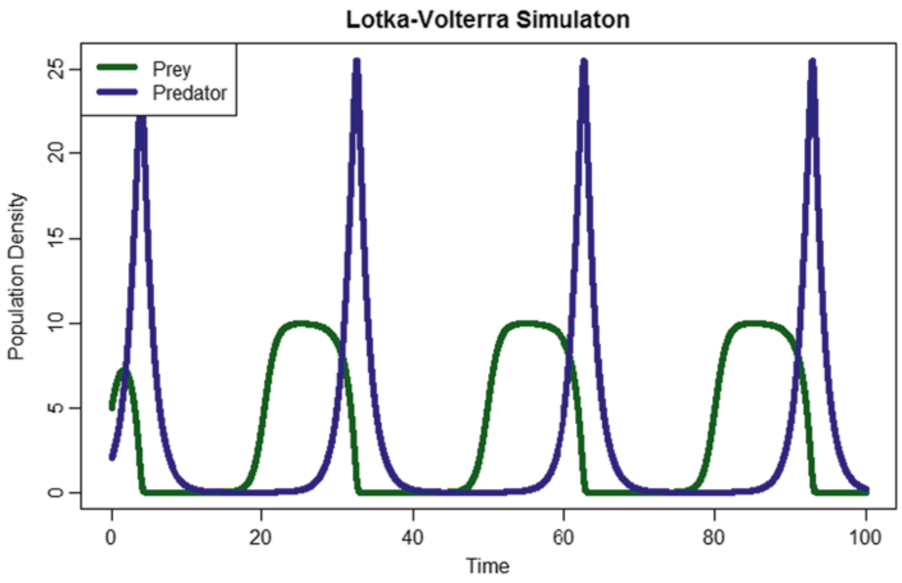


Fig. 14 Altered carrying capacity result

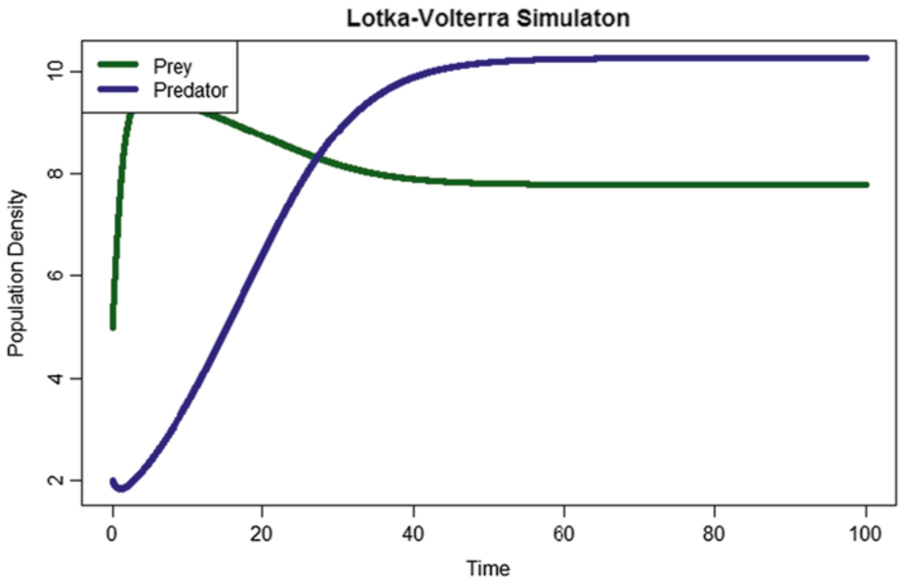


Fig. 15 Altered alpha value result

If we assume a larger α value (i.e., low predator affinity for prey from low capturing efficiency), a different pattern is likely to emerge. This can be tested as shown below (Fig. 15):

```
  Lotka_Volterra_Expanded<-function(x_start=5,
y_start=2,A=1.3,B=0.5,C=1.6,D=0.7,iterations=
    10000,time_step=0.01,alpha=10, K=10)
```

Here we notice that the populations seem to reach a "steady state". This can also be demonstrated mathematically by setting the rate of change for each population equal to zero in Eqs. 5 and 6.

$$0 = Ax\left(1 - \frac{x}{K}\right) - y \times Bf\left(x\right). \tag{8}$$

$$0 = y \times Cf\left(x\right) - Dy. \tag{9}$$

From Eqs. 7 and 9, we obtain

$$f\left(x\right) = \frac{D}{C} = \frac{x}{a + x}. \tag{10}$$

Thus, x*, the steady state level of prey x is

$$x^* = \frac{D\alpha}{C - D} \sim 7.8. \tag{11}$$

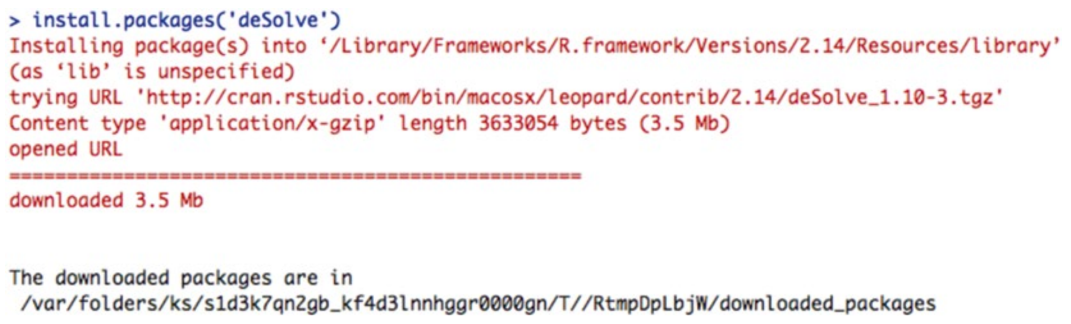
Combine Eqs. 8 and 11, y*, the steady state level of predator y is

$$y^* = \frac{Ax^*\left(1 - \frac{x^*}{K}\right)}{Bf\left(x^*\right)} = \frac{Ax^*\left(1 - \frac{x^*}{K}\right)}{B\frac{x^*}{a + x^*}} = \frac{A}{B}\left(a + x^*\right)\left(1 - \frac{x^*}{K}\right) \sim 10.2. \tag{12}$$

As shown above, when the rates of change for both populations are set to zero, the steady state population densities for both predator and prey correspond to the values shown in the above graph. At the steady state, the birth of prey equals consumption of prey, while the birth of predator due to consumption of prey equals the natural death of predator.

7 Using ODE Solver Libraries for Population Modeling

The above examples were pedagogical demonstrations of how R can be used to simulate population dynamics based on approximating differential equations with difference equations via Euler's method. There are other mathematical techniques for solving differential equations that are more accurate than Euler's method. Many of these techniques are incorporated into the R environment through *packages*, which can be thought of as an "add-on" to the current environment meant to serve a specific purpose. For example, one of the most popular packages for solving differential equations in R is the *deSolve* package [6]. This package is useful when solving *initial value problems*, which are differential equations where the initial values of the state variables (e.g., population densities of prey and predator) are given.



```
> install.packages('deSolve')
Installing package(s) into '/Library/Frameworks/R.framework/Versions/2.14/Resources/library'
(as 'lib' is unspecified)
trying URL 'http://cran.rstudio.com/bin/macosx/leopard/contrib/2.14/deSolve_1.10-3.tgz'
Content type 'application/x-gzip' length 3633054 bytes (3.5 Mb)
opened URL
==================================================
downloaded 3.5 Mb


The downloaded packages are in
 /var/folders/ks/s1d3k7qn2gb_kf4d3lnnhggr0000gn/T//RtmpDpLbjW/downloaded_packages
```

Fig. 16 Install package command output

We will illustrate how to use this package to solve the Lotka–Volterra equations (1) and (2). The first step is to install the *deSolve* package. This can be achieved using the *install.packages* command:

```
> install.packages('deSolve')
```

You should see something similar to Fig. 16 on your console.

Next, open a new R Script for editing and add the following to it (Fig. 17):

To make the code more understandable, let's break it down section by section. First, look at the following piece of code:

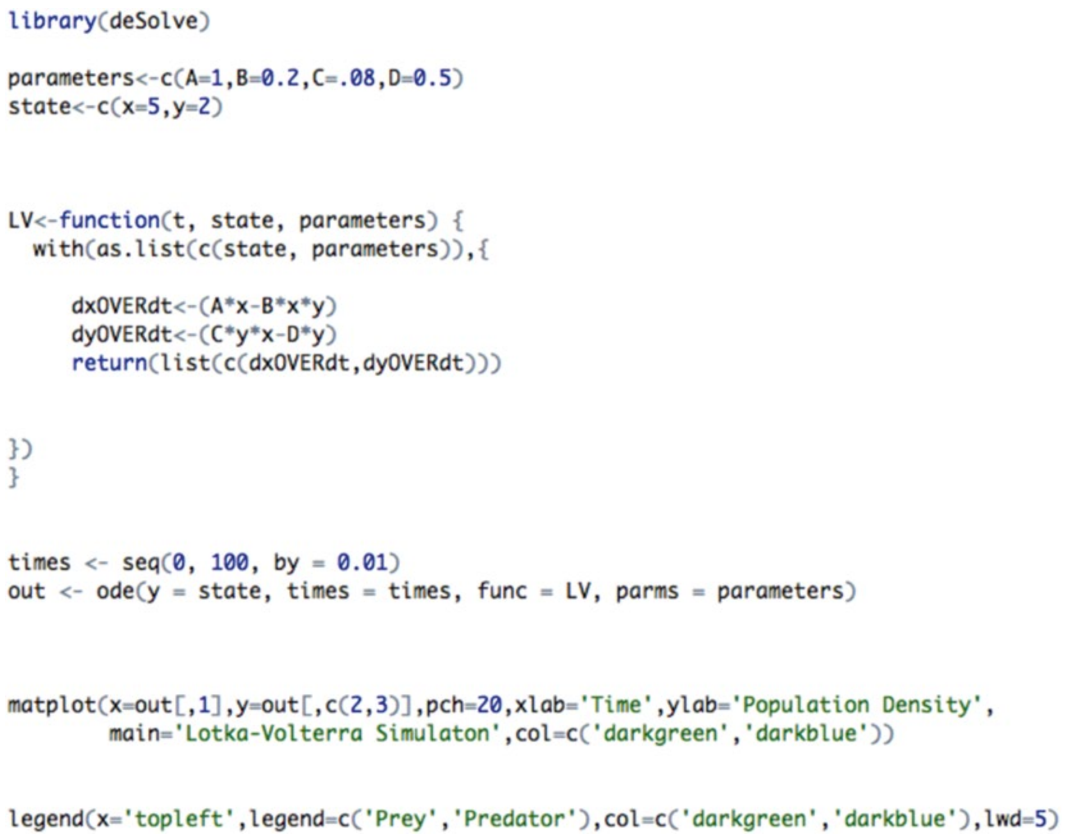
```
library(deSolve)

parameters<-c(A=1,B=0.2,C=0.08,D=0.5)
state<-c(x=5,y=2)
```

The *library* command is a way of telling the R environment to load the functionalities from the installed *deSolve* package for use in this script. Note that in this case we are storing the parameters of the Lotka–Volterra equations in a vector called *parameters*. The initial values of state variables *x* and *y* (prey and predator population densities, respectively) are stored in a vector called *state*. Next, we declare the actual function used for calculations:

```
LV<-function(t,state,parameters){
   with(as.list(c(state,parameters)),{
      dxOVERdt<-(A*x-B*x*y)
      dyOVERdt<-(C*y*x-D*y)
      return(list(c(dxOVERdt,dyOVERdt)))
})
}
```

Notice that d*xOVER*d*t* and d*yOVER*d*t* correspond to the d*x*/d*t* and d*y*/d*t* from Eqs. 1 and 2, respectively. The *with(as.list(c(…))* command is used to allow to access the values stored in the *parameters* (*A, B, C,* and *D*) and *state* (*x* and *y*) by their names



```
library(deSolve)

parameters<-c(A=1,B=0.2,C=.08,D=0.5)
state<-c(x=5,y=2)



LV<-function(t, state, parameters) {
  with(as.list(c(state, parameters)),{

     dxOVERdt<-(A*x-B*x*y)
     dyOVERdt<-(C*y*x-D*y)
     return(list(c(dxOVERdt,dyOVERdt)))


})
}


times <- seq(0, 100, by = 0.01)
out <- ode(y = state, times = times, func = LV, parms = parameters)



matplot(x=out[,1],y=out[,c(2,3)],pch=20,xlab='Time',ylab='Population Density',
        main='Lotka-Volterra Simulaton',col=c('darkgreen','darkblue'))


legend(x='topleft',legend=c('Prey','Predator'),col=c('darkgreen','darkblue'),lwd=5)
```

Fig. 17 ODE solver implementation

(this is a syntax step that must be taken for easily using the *deSolve* package). It is also important to note that we are returning a vector of the d*xOVER*d*t* and d*yOVER*d*t* values (the order is also important—rates of change for state variables must be returned in the same order they were listed in the *state* vector). This function will be applied iteratively by the *deSolve* package for model calculations.

```
times<-seq(0, 100,by = 0.01)
out<-ode(y=state, times=times, func=LV,
parms=parameters)
```

The first command uses the *seq* function, which is used to create a time sequence data object. Essentially, this data object will be used to tell the *deSolve* package to sample from $t=0$ to $t=100$ every 0.01 time steps (just like our code written previously). The second command uses the *ode* (ordinary differential equations) function from the *deSolve* package and stores the output in the matrix *out*. The *ode* is the default function for standard initial value problems, but there are many more functionalities in the *deSolve* package (for more information, type

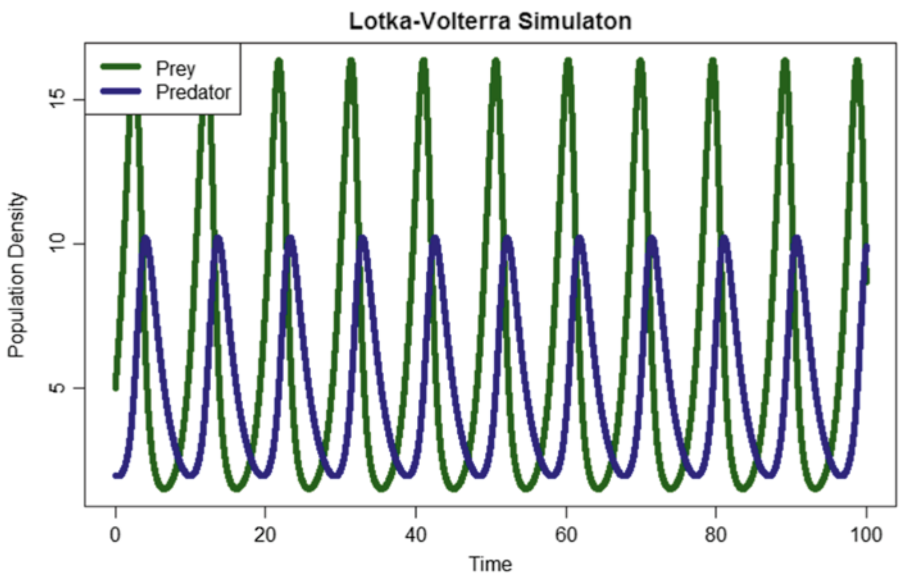


Fig. 18 Results using deSolve package

"?deSolve" in the Console). By default, *ode* uses an interface to an ODE solver written by Linda R. Petzold and Alan C. Hindermarsh [7]. However, there are other approaches, such as the Runge-Kutta method, that can be used by the *ode* function with the *method* parameter (for more information, type "?ode" into the Console).

Finally, because *out* is a type of matrix, it can be indexed and plotted like previous examples:

```
matplot(x=out[,1],y=out[,c(2,3)],pch=20,xlab='
Time',ylab='Population Density', main='Lotka-
Volterra  Simulation',col=c('darkgreen','darkb
lue'))

legend(x='topleft',legend=c('Prey','Predator'),
col=c('darkgreen','darkblue'),lwd=5)
```

Note that in this case, we haven't stored the bulk of the code in a function. Rather, every time the script is sourced, the entire code will run. Sourcing this file should give Fig. 18:

Notice how the peak heights of the predator and prey populations do not seem to increase, despite having a time step of 0.01. This can be attributed to the more accurate estimation in the *ode* function compared to the Euler's method.

Note: If, at any time, debugging is required, there are useful functionalities built into R that allows the user to halt execution of a function or script and examine the values stored in various parameters. One such function, browser(), can be placed at a given point in a script/function/loop and creates an interactive environment for the user to examine values. For more information, type "?browser" into your command console.

8 Conclusion

The purpose of this chapter was to demonstrate the power of using simple and open-source software like R to examine population dynamics represented by a mathematical model. With a mathematical model, hypotheses can be easily formulated and tested. The ability to represent abstract concepts in an intuitive manner can greatly facilitate the understanding of novel concepts and phenomena, leading to new insights in biological sciences.

Acknowledgements

The authors would like to thank members of the Shou lab (Björn F.C. Kafsack, David Skelding, Babak Momeni and Adam Waite), Sarah Holte, Jerry Davison, and Alex Hu for their critical feedback and insightful comments. Work in the W.S. group is supported by the W. M. Keck Foundation and the National Institutes of Health (Grant 1 DP2OD006498-01). R.G. is an NSF predoctoral fellow.

References

1. Malthus T (1798) An essay on the principle of population: an essay on the principle of population, as it affects the future improvement of society with remarks on the speculations of Mr. Godwin M, Condorcet and other writers. Electronic Scholarly Publishing, London, http://www.esp.org/books/malthus/population/malthus.pdf
2. Venables WN, Smith DM (2013) The R Core team. An introduction to R-Notes on R: a programming environment for data analysis and graphics version 3.0.1 (2013-05-16). http://www.cran.r-project.org/doc/manuals/R-intro.pdf
3. Lotka AJ (1925) Elements of physical biology. Williams & Wilkins, Baltimore, MD
4. Volterra V (1926) Variations and fluctuations of the number of individuals in animal species living together. J Cons Perm Int Ent Mer 3:3–51, Reprinted in R.N. Chapman, Animal Ecology, New York, 1931
5. Soetaert K, Petzoldt T, Setzer RW (2010) Solving differential equations in R. R J 2(2): 5–15
6. Soetaert K, Petzoldt T, Setzer S (2010) Solving differential equations in R: package deSolve. J Stat Softw 33(9):1–25
7. Hindmarsh A (1983) ODEPACK, a systematized collection of ODE solver (Stepleman R, et al. ed). IMACS Trans Sci C Comput 1:55–64