Mathematics for the Life Sciences:
Calculus, Modeling, Probability, and Dynamical Systems
R Scripts and Instructions

My first experience of computers was in 1971 as a 10th grader in an advanced algebra class. We had a remote terminal connected to the main frame computer housed in the district office. We wrote programs in Basic (the text version, not Visual Basic), ran them immediately, and had them printed on yellow paper tapes because software storage methods did not yet exist. The power conferred by the capability of making a computer do calculations was immediately obvious. I had no doubt that within a few years computer programming would become as standard a topic in school as mathematics or English. Fast forward to 2013, where Estonia has become the first country to mandate instruction in computer programming for all children. Many people will be surprised when Estonia becomes the dominant force in world science and technology in twenty years, but not me.

In the 42 years since I started programming, learning to write computer programs has been a clear path to success, not just for the entrepreneurs who created Apple and Microsoft, but also to the hordes of programmers who work for these companies and others. There are a number of reasons for this success. One is that computers are useful, but only when they are running good programs. Few of us can write a word processing or spreadsheet program ourselves. Another is that computer programming is good mental training, and this reason applies to everyone.

The skills learned in computer programming have so suffused my thinking that I cannot picture myself without them. I make this claim in spite of the fact that by modern standards I am not considered to have real programming skills. Real programmers today do object-oriented programming, which is necessary for any program that is launched, interacts with a user through sporadic mouse clicks, and terminates only when so directed by the user. This type of programming is difficult, but it is not necessary for mathematics. Mathematics programs are completely set up in advance, run on their own with no user input, and terminate automatically. The simpler function-oriented paradigm is all that is needed for this kind of programming. Where object-oriented programs have come to be called *applications*, function-oriented programs are often called *scripts*. The name is apt, as function-oriented programs dictate calculations that are followed literally in a prescribed order. The calculations can be made conditional on intermediate results, but the computer still follows a predetermined path.

No special talents are required to learn to write scripts. It is very similar to giving a set of instructions to an assistant, but with one key difference: computers are incredibly stupid. They can only understand communication that follows a highly proscribed format, and any deviation from that format on the part of the programmer leads to error in execution of the script. Function-oriented programming consists of two unrelated sets of skills–the skill of communicating with computers in accordance with rigid syntax requirements and the skill of breaking a complicated procedure down into tiny pieces that can be executed in some sequence. Both of these skills can be learned through practice, and both make us better thinkers in all settings. Learning to write scripts will help you harness the power of

mathematics beyond what you can do by hand, and it will also help you harness the power of your own mind beyond what it can do now.

For the skill of communication with computers, I recommend what I call the "Rosetta stone" method. Start with simple scripts written by a skilled function-oriented programmer. Run them line by line so that you understand what each line does, picking up the syntax requirements as you go. If, for example, you run the script

```
x <- 2
y <- 3*x
y
```

in R, you see the following output:

```
> x <- 2
> y <- 3*x
> y
[1] 6
```

A number of conclusions about R syntax follow from this example.

1. The construction "name <– formula" assigns the numerical value of the formula to the name, as clearly the program has obtained $x = 2$ and $y = 6$.

2. Multiplication always requires the "*" symbol, else the script would have had "y <– 3x".

3. All commands in the script are echoed in the output. (Syntax errors provoke R output that interrupts the echo so that you can tell which line detected the error. The error could actually be a formula error in an earlier line rather than a true error in syntax in the flagged line. One method for locating the actual error is to salt the script with extra statements that output intermediate calculation results.)

4. Assignment commands are normally echoed without showing the results, but we can obtain results at any point in a script using a command that merely names the variable whose value is desired.

This guide accompanies the folder of computer scripts found on the web site for the book. A few of these scripts are complete, but most of them are templates that require completion by the user. Completed scripts can be adapted for use with similar problems.

Getting Started in R

When you open the R application, you get a full-screen window that has some open space and a smaller window that is the R console. You can enter commands in the R console and have them executed immediately. Type

```
x <- 2
```

and hit the Enter key. You will simply get a new command prompt. Now enter

```
x
```

and you will get the answer 2, indicating that your first line did indeed assign the value 2 to the variable name x.

The problem with doing computations in the console is that you can't save them. Instead, it is better to create script files that contain a set of commands. From the file menu, click on New Script to bring up a script window. Move and resize the two windows so that there is no overlap. You want to be able to see the full contents of both windows regardless of which one is active.

Type

```
x <- 2
y <- 3*x
y
```

in the new script file. To run this script, type <Ctrl>-ar. This sequence highlights the entire text and then executes all of the commands.

One of the most difficult programming skills to learn is the skill of debugging. Remove the "*" symbol from your script and run it again. Note the error message that appears in the console window. The error messages appear at the first point where a script command failed. This may not be the line where the actual error occurred. Even when it is the right line, the message may not clearly identify the error. In this case, the error message `unexpected symbol` is misleading; the real problem is that there is a symbol missing. Unfortunately, R doesn't know what the correct formula should have been. It only knows that a letter should not follow a number. The unexpected symbol is the x.

One drawback of R is that the official documentation is not always very helpful. When I need to look something up, I do a Google search. For example, I needed to know how to get a dashed line for the program 01 below, so I searched "line styles in R" and got a nice document on the web site of Washington University of St. Louis.

General Note on R Scripts for This Book

The Rosetta Stone method of computer programming instruction has two components. The first component is that new instructions need to be explained so that students can understand them. Each time I introduce something new in R, I include the full statement in the program and the full explanation in this guide. The second component is that students must write new program code that adapts instructions to new purposes. Each time a calculation is needed that uses R syntax already discussed, I leave a set of three blanks (___) for the reader to fill in. Doing this requires knowledge of the context as well as R syntax. For example, there is a formula to calculate a quantity `m` in script 2-3. The reader who is working through Section 2.3 should know the correct formula.

It is good programming practice to test lines of code as you write them. This requires you to execute small blacks of instructions rather than the full script. In R, you do this by highlighting the desired lines and then typing <Ctrl>-r. The shortcut <Ctrl>-ar discussed

earlier runs the entire code without the need to highlight specific lines. (Think of "r" as representing "run" and "a" as representing "all".)

Part I
Calculus and Modeling

The first three scripts present instructions for plotting functions and linear and nonlinear curve fitting. The instructions in these scripts are used in scripts for Parts 2 and 3 without further description.

1-1 Plotting in R

The first script introduces much of the syntax required to produce graphs in R as well as some other standard R syntax. In the commentary, I also discuss general programming style.

```
### This program reproduces Figure 1.1.4.
```

Any command that begins with the # character is a comment. It is a good idea to use comments to document a script so that you can figure out what it does a year after writing it. I use comments with ## to break scripts into sections.

```
## DATA
```

I use the data section to define quantities that I envision changing for additional runs of a script. The viewing window of a graph might need to be changed, or I might want to change parameter values. Putting these definitions together in a small section makes it easier to find them and allows me to ignore the working parts of a script when I am only doing additional runs with slight modifications.

```
# k values
k <- c(0.5,1,2)
k
```

The command `c` is used to create a list of values with a single name. If needed, R will interpret this list as a row or column vector, but most often it is simply a numbered list that can be accessed element by element.

```
## INITIALIZATION
```

I use the initialization section for instructions that set up the main computations in the program. In this case, the only initialization needed is to set up the list of `t` values for the plots. In most programs, this section does not need to be changed from one run to another.

```
# make list of t values
t <- 0.02*(0:100)
t
```

There are a number of ways to make lists of numbers in R. If we want a list of 100 equally-spaced values, we certainly don't want to have type them all into a `c` command. One nice way is to define the values as the product of a spacing and a range. This command creates a set of values with spacing 0.02 and running from 0 to 2. Note that the construction `m:n`, where `m` and `n` are integers, creates a list of consecutive integers from `m` to `n`.

```
## COMPUTATION
```

I use the computation section for the main calculations in a script. These are the same in every run. As our scripts get more complicated, it will become a big advantage to have a large part of the script that we need not look at when doing new runs.

```
# compute half-lives
T <- log(2)/k
T
```

Note that R uses `log` for the natural logarithm rather than the more usual "ln". This command shows why lists of values are convenient in R. One calculation produces the half-life for each of the values of k.

```
## OUTPUT
```

R scripts generate echoes of all instructions, so any calculation reported in the early part of a run has scrolled out of view by the end. In most scripts, I collect all statements that produce some useful output at the end in an output section. I generally put plots before numerical results so that the numerical results come after all command echoes. Because plot parameters sometimes need to be changed for different runs, the output section often needs to be modified for different runs of a program.

```
# plot the first function
plot(t, exp(-k[1]*t), type="l", xlim=trange, ylim=yrange,
    xlab="t", ylab="y", lty="dotdash", lwd=2, xaxs="i", yaxs="i")
```

The plot command contains only two required arguments: the lists of horizontal and vertical coordinates of the points. These can simply be names, such as t, or calculations. Here, the function to be plotted is $y = e^{-0.5t}$, which means that we want the first value of the list `k`. Note that the first value is accessed using square brackets. The remaining arguments of the plot command are optional and can be in any order.

- The type parameter specifies that we want a connect-the-dots curve rather than the specific points. If this command is omitted, the points are plotted instead.
- The `xlim` and `ylim` parameters specify the viewing window. If omitted, R has a built-in algorithm that chooses the viewing window.

- The `lty` parameter determines the line style. The standard choices are `solid`, `dashed`, `dotted`, `dotdash`, `longdash`, and `twodash`. The default is solid.
- The `lwd` parameter specifies the line width. The default line width of 1 is rather thin, so I generally use a line width of 2.
- The `xlab` and `ylab` parameters prescribe the axis labels.
- The `xaxs` and `yaxs` instructions specify the relationship between the plot window and the axis limits. The default is for the plot window to extend a little bit beyond the axis limits. All of the graphs in the book have the plot window the same as the axis limits, which we get with the specification of `i`. This does not always look best in R, so the user should try plots with and without these specifications.

```
# plot the remaining functions (types 1 and 2 are solid and dashed)
for (i in 2:3)
    lines(t, exp(-k[i]*t), lty=i-1, lwd=2)
```

The plot command in R erases any previous plots. Hence, we need a different command to add a curve to an existing plot, and that is what the lines command does. Note that the optional arguments should only include those items that are specific to each curve and not those that are properties of the graph as a whole. Note that the lines command is executed twice, once with `i=2` and once with `i=3`. Repeated instructions can be embedded inside a loop indexed by a parameter so that the command is executed differently each time. In the lines command here, the index `i` appears twice. We get two different curves by using different elements of the `k` list and we get two different line styles by calculating a value for `lty`. The numbers refer to the list of line styles given above; in particular, `1` is solid and `2` is dashed.

```
# plot the dotted lines
lines(c(0,T[1]), c(0.5,0.5), lty="dotted")
for (i in 1:3)
    lines(c(T[i],T[i]), c(0,0.5), lty="dotted")
```

The last set of instructions plot all the dotted lines in the figure. There are several ways to plot straight lines, but the easiest to remember is to use the standard lines command with the list of x coordinates of the end points followed by the list of y coordinates of the end points.

2-3 Linear Least Squares

This script applies the least squares formulas to obtain a best fit line for a set of data. The data can be entered directly in the program using the `c` command as in program 01; however, there are cases where the data set is large enough that it is easier to import it from a file. The easiest way to do that is to put the data in columns in a spreadsheet and then save the file as comma-delimited (csv). The file DetroitLake0.csv contains two columns of data with

no headers. The first column is the year, with 1930 representing the winter of 1929-1930. The second column is the number of days of consecutive ice cover seen at Detroit Lake, Minnesota during that winter. The slope of the straight line is in some sense an indication of global warming, although we should be careful not to interpret this result seriously. More thorough studies of global warming are indicated in Problems 2.3.9 and 2.7.10.

```
# Define the original data for a linear model Y=mX.
data <- read.csv("DetroitLake0.csv",header=FALSE)
x <- data[,1]
```

The first of these instructions defines a data frame in R. Data frames are matrices augmented with headers. We can extract the pure numerical values from the data frame using data handling instructions in R. The first line creates a variable x using all of the numbers in the first column of the data. If we only wanted the first ten numbers, we could use `data[1:10,1]`.

```
# Modify the data so that the mean is at (X,Y)=(0,0).
xbar = mean(x)
X = x-xbar
```

The mean function computes the arithmetic average or mean of a list of numbers. Note that it is legal in R to subtract a single number `xbar` from a list `x`, the result being a list in which the subtraction has occurred for each element.

```
# Compute the best slope and residual sum of squares.
sumX2 <- sum(X^2)
RSSavg <- RSS/length(X)
```

The sum and length functions extract properties of a list of numbers, similar to the mean function. We could have calculated the mean directly with `xbar = sum(x)/length(x)`. There are often trade-offs between using more elementary components that have to be assembled versus using pre-assembled but specialized components. At times, I will combine R components rather than add more syntax to the list of things the reader should know. In this case, the built-in mean function is easy to remember and will have multiple uses.

```
# Plot the data.
plot (___, ___, xlab="year", ylab="ice duration")

# Plot the linear regression line.
abline (b, m, lwd=2)
```

Fewer optional instructions are needed in this plot command, primarily because we are using it to plot data rather than a curve, which corresponds to the defaults in R. The abline instruction is a convenient way to add a straight line to a plot when it is easier to identify the lline by slope and intercept than by end point coordinates.

```
# Display the slope, residual sum of squares (RSS), and average RSS
```

The reader should know how to display results, as this was done in the first elementary example in which we calculated and displayed a variable $y = 3x$.

2-4 Semilinear Least Squares

This script computes the parameters for models of the form $y = qf(x, p)$. It can be applied to any model of this form, which means that the user must provide the model form as part of the program. In making the necessary modification, one could change the names of the parameters q and p wherever they occur in the program, but it makes more sense for programming to simply change the names of the parameters in the model. The example in the text uses the model $y = Ax^p$, so we can define $f(x, p) = x^p$ and simply associate q with A.

```
# Specify the data.
x <- ____
y <- ____
```

The data can be specified as lists of values using the `c` instruction or it can be read from a csv file. In the example from the text, we use the *P. speedius* data from Table 2.1.1.

```
# Define the function f.  The parameter must be the first argument.
f <- function(p,x) x^p
```

Simple functions can be defined in R using a single line of code. This particular line establishes a function $f(x, p)$ calculated with the formula x^p. Note the special use of the reserved word "function". In ordinary mathematical usage, it is common to say that $y = x^2$ defines a function $y(x)$. This interpretation is not always correct outside of a mathematical context. In computer programming, we should say that the statement defines a value y in terms of a value of x. The statement `y <- x^2` cannot calculate the square of any number other than the one currently stored as x. The word "function" in computer programming specifically refers to a line of code that produces an output from any given input, regardless of the name the input had been given. Thus, the code given here will calculate $f(p, x)$ for any pair of numbers. For example, suppose the script has previously assigned values of 2 and 3 to names `y` and `z`, respectively. Then `f(y,z)` will produce the result 9, just the same as `f(2,3)`.

```
# Provide minimum and maximum values for p.
minp <- 0
maxp <- 1
```

The program finds the optimal p using a built-in function that requires an interval that brackets the optimal value. In this case, a plot of the data clearly shows that the graph is increasing, which means $p > 0$ and concave down, which means $p < 1$.

```
# Provide a list of x values for the plot.
xplot <- seq(0,5,by=0.1)
```

The variable `xplot` is so named to distinguish it from the variable `x`, which contains the x values of the data. In script 02, we were able to plot the best fit line simple by using the parameters, but this can only be done for a line. Here, we want to plot the function f with particular choices for p and q; as in script 01, we can do this only by preparing a list of values for numerous points. The `seq` instruction provides an alternative means of defining a list of equally spaced values.

```
# Calculate quantities that do not depend on p.
sumy2 <- ___
numpoints <- ___
```

The algorithm for semilinear least squares involves some quantities that do not depend on the parameter p. These could be put in either the initialization or computation sections. The reader should know how to calculate these from earlier scripts.

```
## FUNCTIONS

# Define the function F(p) that determines RSS when q is optimal.
F <- function(p) ___-(___)^2/sum((f(p,x))^2)
```

This instruction defines a function, so it is the same kind of programming statement as the one that defined the function $f(p, x)$. I put this one in a separate functions section because it is part of the fixed code for the script, whereas the other is part of the data to be specified by the user. I have found the standard practice of isolating code that needs to be changed between runs in the data and output sections. Note that the function F uses all of the x values from the data, the input value p, and the model f to produce a residual sum of squares corresponding to the optimal choice of q. The formula for the residual sum of squares is derived in Section 2.4 of the text.

```
# Find the optimum value of p and the residual sum of squares.
result <- optimize(f=F, lower=minp, upper=maxp, maximum=FALSE)
pstar <- result$minimum
RSS <- result$objective
```

The name `optimize` refers to a built-in function that produces more than a single quantity of output. In such cases, R combines the outputs into a data structure called a *frame*. This is something R programmers must understand. The first line of code runs the optimize function, using the user-defined function `F` as the input function that `optimize` knows as `f` and the user-defined quantities `minp` and `maxp` as the lower and upper bounds. It could be used to find either the maximum or minimum, depending on what is indicated for the parameter `maximum`. What makes a multiple-valued function different from a normal function in R is that you have to know the right syntax for unpacking the data structure. The code puts the output of the `optimize` function into a variable called `result`, but this quantity is a frame rather than a number. What we actually want in this case are the components of the frame that contain the minimizer and the corresponding function value. These are accessed using the `$` symbol as in the subsequent lines of code. To use the `optimize` function, we need to know the names of the desired results (`minimum` and `objective`) as well as the input arguments (`f`, `lower`, `upper`, and `maximum`). Presumably we would have used `pstar <- result$maximum` if we had set `maximum` to `TRUE`, and it also seems that R does not have a restriction preventing input names from matching names for the output data.

```
# Find the corresponding value of q and the average residual sum of squares
qstar <- ___
```

```
RSSavg <- ___

# Compute y values for the plot.
yplot <- ___
```

The reader should be able to fill in these instructions using prior knowledge about R programming and semilinear least squares.

```
# Determine the axis bounds.
xmax <- max(x,xplot)
ymax <- max(y,yplot)

# Plot the data.
plot (___, ___, xlim=c(0,xmax), ylim=___, xlab=___, ylab=___)

# Plot the model.
lines (___, ___, lwd=___)
```

The plot commands are essentially the same as in earlier scripts, but here we have some code to determine the axis bounds. If the axis bounds are left unspecified, R will choose them to match the data used in the original plot command. If the points used in the subsequent lines command extend beyond the boundaries of the axis limits, it will be too bad. The code used here avoids this problem by finding the largest values of `x` and `y` in the combined sets of data and then hard coding the axis bounds to go from 0 to the appropriate maximum.

Part II
Probability

The scripts in this section present descriptive statistics and probability distributions. The reader should be familiar with the instructions used in the Part I scripts.

3-1 Descriptive Statistics

This script collects the ice duration values from DetroitLake0.csv, computes the mean,variance, and standard deviation, and produces a histogram of the data. Its only real purpose is to document the basic instructions for descriptive statistics. Nothing more need be said, as the documentation is self-contained.

3-X Probability Distributions

Like the previous script, this one is primarily here to document the basic instructions for probability distributions. The script does include a small amount of code that produces a plot containing a probability density function and an associated histogram. Combining the two plots requires some special effort, also documented in the script.

```
# Define the set of x values for the histogram boundaries.
dx <- 1
x1 <- seq(0-0.5*dx,10+0.5*dx,by=dx)
```

We have chosen a bin width of 1 with average x values from 0 to 10. The boundaries of these bins must then run from -0.5 to 10.5.

```
# Use the cumulative distribution function to compute the frequencies.
f1 <- pexp(x1[-1],rate=0.2) -pexp(x1[-length(x1)],rate=0.2)
```

We want ten values for the histogram, each with the accumulated probability corresponding to that interval. Thus, the first histogram value should be $E(0.5) - E(-0.5)$ and the last should be $E(10.5) - E(9.5)$. This is accomplished in a single instruction by creating a list $f1$ in which the list with elements $E(-0.5)$ through $E(9.5)$ is subtracted from the list with elements $E(0.5)$ through $E(10.5)$. Recall that `x1[-1]` refers to the list `x1` without its first element.

```
# Use the barplot instruction to plot the histogram.
barplot(f1,space=0,names.arg=0:10,ylim=c(0,0.2))
```

A description of `barplot` occurs earlier in the script.

```
# Define the set of x values for the probability density function plot.
#  Normally a plot of 1000 points is overkill.  In this case, we need
#  a very large number because the pdf is discontinuous at x=0.
x <- seq(0-0.5*dx,10+0.5*dx,by=0.01*dx)

# Compute the set of y values for the pdf plot.
f <- dexp(x,rate=0.2)
```

We now want to add a plot of the probability density function to the barplot, so we want the same range of x values.

```
# Add the pdf plot to the barplot.  Note that the x values need
#  to be shifted by half of the bin width so that they line up
#  correctly with the barplot.
lines(x+0.5*dx,f,type="l",lwd=1.5,col="blue3")
```

This instruction seems to make perfectly good sense except that x coordinates are specified as `x+0.5*dx` rather than `x`. This has nothing to do with the mathematics, but is an artifact of the way R plots bar graphs. The vertical axis for a bar plot is positioned differently on the screen that a standard vertical axis. Since the bar plot was done first, all we can do is to shift the graph for the probability density function to line up. If we use the seemingly correct `x`, the y values for the centers of the bars will be plotted at the left end instead.

4-1 Random Sampling

One important use of computers in biology is to set up simulations for stochastic processes (processes that produce randomly-distributed results). These require random samples to be drawn from a probability distribution. The book uses random samples from probability distributions for pedagogical experiments in Sections 4.2 and 4.3.

R has excellent built-in capability for drawing random numbers from probability distributions and manipulating the resulting data. Some of the key instructions are introduced in this script. We also introduce the first of several control structures, the counting loop.

Note that this script requires instructions that define data values, produce a histogram from a list of probabilities, and display results of output variables. The reader should be able to write these instructions from prior knowledge. One point is worth noting: omitting the `ylim` parameter from the `barplot` instruction allows R to use its default routine for determining the vertical axis limits. This is usually a good thing, and you can always prescribe the parameter value later if you don't like the way the plot looks initially.

```
# Collect a sample of size N from a binomial distribution b_{n,p}.
X <- rbinom(N,size=n,prob=p)
```

The instructions for drawing random numbers from a probability distribution use the prefix "r" with the same suffix as that used for other instructions with the same distribution, and the parameters are mostly the same as well.

```
# Set up the list of frequencies.
maxX <- max(X)
counts <- rep(0,maxX+1)
```

The `rep` instruction given here creates a list of length `maxX+1`, with all values preset to 0. The actual preset value is irrelevant, as we will subsequently assign the correct values. It is generally good practice in R to create data structures of the desired size before filling in the values. The previous instruction is needed to establish the correct size for the list. We need to record the frequencies of all outcomes in the interval $[0, x_m]$, where x_m is the largest outcome obtained from the simulation; hence the list needs to have $x_m + 1$ elements.

```
# Count the occurrences of each outcome.
for (k in 0:maxX)
    counts[k+1] <- length(which(X==k))
```

This composite instruction is best examined one part at a time. The function `which(X==0)` determines which entries in the list `X` are the number 0. We don't actually care what order the outcomes are drawn; we only want to know how many 0s there are, which is the length of the list obtained from the `which` function. Hence, the command

```
counts[1] <- length(which(X==0))
```

finds the list entries that have value 0, counts how many there are, and records that number in the first position of the list `counts`, replacing the initial value of 0. Our goal is to count the number of occurrences of each of the outcomes and store these in the appropriate position in the list. Given any possible outcome `k`, we want to store the total count in position `k+1`. This can be accomplished in one step by embedding the instruction

```
counts[k+1] <- length(which(X==k))
```

in a loop that executes the instructions for all desired values of `k`. The general structure of a counting loop is

```
for (var in set)
    {
    statement
    statement
    ...
    }
```

where `for` and `in` are part of the R syntax and "var", "set", and "statement" are used to represent whatever variable, set, and statement(s) are desired. The curly braces are optional when there is only one statement, as in the current example. The layout is also optional. I find that it makes scripts easier to read if the curly braces are typed in separate lines and indented, with the statements in the loop indented as well.

4-2 Normality Tests

This script should be almost entirely reproducible by anyone who has worked through the first six scripts and Section 4.2 of the book. To encourage more independence in the programming student, I have begun with this script to use ___ ___ to indicate places where the programmer needs to add two or more instructions. Only two instructions in this script are new.

```
# Draw n values from a uniform distribution.
X <- runif(n,min=0,max=1)
X <- sort(X)
```

The reader familiar with the previous two scripts could probably guess that `runif(___)` is the instruction for obtaining values from a uniform distribution. As the instruction shows, the programmer needs to indicate how many values are desired and supply the minimum and maximum outcomes for the distribution. The `sort` function rearranges the elements in the input list in ascending order. The sorting is necessary for producing the EDF plot and calculating the Cramer-von Mises statistic. The output is worth noting. Most runs of the script produce a value for the Cramer-von Mises statistic that is not inconsistent with a normal distribution; however, the clear pattern in the EDF plot suggests that the distribution is not normal. A larger sample of data from a non-normal distribution will generally yield a larger value of the statistic.

We could have used a different name for the sorted list, but it is standard programming practice to reuse names when the old values are no longer needed. Note that the goal could also be accomplished by nesting the two instructions in a single line of code:

```
X <- sort(runif(n,min=0,max=1))
```

The single line version is easy enough to read in this case, but we should be cautious about combining too many instructions or making single formulas that are too complicated. Programs only work correctly if there are no errors, and readability, rather than brevity, is the key to identifying errors.

4-3 Distributions of Sample Means

This script collects a large number of values from the distribution of sample means with underlying distribution $E_{0.5}$, determining the key statistics of the distribution and producing the plot of Figure 4.3.2b.

```
# Draw N values from the distribution of sample means and sort.
X <- rep(___)
for (___)
    X[k] <- mean(___)
X <- sort(X)
```

The blanks in this section of code can be filled in by reference to earlier scripts. Note that the random variable `X` is defined as a list of size `N` with all values initially set to 0. The actual values in the list are computed one at a time in the loop. Each value is obtained by collecting a list of `n` values from the appropriate probability distribution and computing the mean. The values are then sorted because the script computes the Cramer-von Mises statistic.

```
# Prepare the histogram.
over <- length(which(X>7))
N <- N-over
hist(X[1:N],freq=FALSE,breaks=___,col="yellow")
```

There are two subtleties in the histogram instruction used here. First, the desired plot only includes means up to 7, but some of the samples may have a larger mean. These samples must be culled from the data. The variable `over` is used to record the number of outsized values, and this number is subtracted from the total to identify those values that are to be included in the histogram. The argument `X[1:N]` in `hist` limits the histogram to the values that are not outsized. Second, the parameter setting `freq=FALSE` is used to make the vertical coordinate in the graph be the relative frequency per bin width rather than the actual frequency. Note that the axis limits differ by a factor of 4 from Figure 4.3.2b, which uses relative frequency rather than relative frequency per bin width.

Part III
Dynamical Systems

The scripts in this section present the instructions needed to run simulations for discrete and continuous dynamical systems. These scripts assume familiarity with the scripts of Part I, but not the scripts in Part II.

5-1 Discrete Equations

This script runs a simulation for a single discrete dynamic variable and plots the results. It will be superseded by script 5-2.

```
# Define the model.
g <- function(x) x+R*x*(1-x/K)
```

As in script 2-4, we can define a simple function using a single line of code. As we have defined the function, the names `R` and `K` are *global* in scope, while the name `x` is *local*. Global names have values defined outside the body of the function; in this case, they are defined in the data section. Local names are only known internally to the function. Thus, the name `x` has been used arbitrarily for the input variable; its value is passed from the main program by any instruction that calls the function `g` and is independent of any global variable called `x`.

```
# Define t and create data structure for N.
t <- 0:T
N <- rep(N0,T+1)
```

The `rep` instruction is used to create a data structure. This instance sets up a list of `T+1` items, all initially set at `N0`. This initial value is only correct for the first entry in the list, which represents N_0, but it will be overwritten in all other list positions.

```
# Compute N values.
for (i in 1:T)
    N[i+1] <- g(N[i])
```

Instead of separately computing `N[2]`, `N[3]`, and all the rest, all `N` values are computed with a single instruction embedded in a counting loop. The general structure of a counting loop is

```
for (var in set)
    {
    statement
    statement
    ...
    }
```

where `for` and `in` are part of the R syntax and "var", "set", and "statement" are used to represent whatever variable, set, and statement(s) are desired. The counting loop produces different results in each run through the loop because the formulas depend on the value of the index variable. The curly braces are optional when there is only one statement, as in the current example. The layout is also optional. I find that it makes scripts easier to read if the curly braces are typed in separate lines and indented, with the statements in the loop indented as well.

```
# Determine top value.
ymax <- ___

# Plot the results.
plot(___)
```

See script 2-4 for discussion of how to set the top value for plot axes.

5-2 Discrete Equations

This script improves on script 5-1 by adding optional code for a cobweb plot.

```
# Set display parameters.
cobweb <- 1                # 1 if yes and 0 if no
Nmax <- K*(1+1/R)          # maximum coordinate on N axes
pts <- 50                  # number of points used to plot g(N)
```

The upper limit on N for a cobweb plot should be set manually or computed; in this case, it is chosen to be the N value for which $g(N) = 0$.

```
# Set up the plot window.
par(mfrow=c(1,1+cobweb))
```

This bizarre-looking instruction sets up a plot window with 1 row and `1+cobweb` columns; hence, we will get the usual plot window in `cobweb` is 0 and a window with two columns in `cobweb` is 1. In the latter case, subsequent plot commands will alternate between the left and right halves of the plot window. Thus, the second plot command will produce a new plot without erasing the previous one.

```
# Produce the cobweb plot.
if (cobweb==1)
    {
    # Plot the function g.
    x <- ___
    y <- ___
    plot(___ ___, xlim=___, ylim=___, xlab=___, ylab="",
        col="blue", lwd=3)
    # Plot the line y=N.
```

```
    lines(___ ___, lwd=2)
    # Plot the first vertical cobweb line.
    lines(c(N[1],N[1]), c(0,N[2]), col="green4")
    # Plot the remaining cobweb lines.
    for (i in 2:T)
        {
        lines(c(N[i-1],N[i]),c(N[i],N[i]), col="red3")
        lines(c(N[i],N[i]),c(N[i],N[i+1]), col="green4")
        }
    }
```

This section of code contains the cobweb plot instructions embedded in a conditional structure that dictates execution only if the cobweb plot is indicated by the value of `cobweb`. The instructions themselves are broken into four groups. The first group computes points for the plot of $g(N)$ and plots the curve through those points, while the second group plots the line $y = N$. The third group plots the vertical line that locates N_1. The remaining group plots pairs of horizontal and vertical lines to locate the points N_2 through N_T.

```
# Plot the results.
plot(___ ___)
```

The final line plots the results of the discrete simulation. This plot will appear on the right if `cobweb` is 1 and in a single window otherwise.

5-3 Differential Equations

This script runs a simulation for a single discrete dynamic variable and plots the results.

```
# Specify total time and number of time steps
tmax <- 20
steps <- 100
```

Numerical methods for differential equations work by computing exact solutions of discrete approximations to the differential equations. The total time and number of steps combine together to determine the step size, which controls the error in the discrete approximation as well the spacing between points in the plot. Of course the number of steps should generally be increased if the total time is increased. Some trial and error can discover how many steps are necessary. If the plot for 200 steps is indistinguishable from that for 100 steps, then 100 steps is sufficient.

```
# The RK4 function computes one step of the solution of a
#   discrete approximation to an initial value problem
#   x'=f(x,t), x(0)=x0.
#   The inputs are the starting value, time, and time step.
```

```
RK4 <- function(x0,t,dt)
    {
    k1 <- f(x0,          t     )
    k2 <- f(x0+k1*dt/2, t+dt/2)
    k3 <- f(x0+k2*dt/2, t+dt/2)
    k4 <- f(x0+k3*dt,   t+dt  )
    x <- x0+(k1+2*k2+2*k3+k4)*dt/6
    return(x)
    }
```

We have already seen the `function` instruction used to create single line functions. The `RK4` function requires multiple lines of code. Those lines must be placed inside a set of curly braces, similar to sets of instructions that go inside a `for` loop. The last line inside the curly braces must be a `return` statement, which indicates the name of the variable whose value is the output of the function. This particular function implements the most popular simple method for discretizing a differential equation. The user does not need to know how the method works; the only crucial thing to understand is that the function has been written with `f` as the name of the function on the right side of the differential equation. This means that the function defined for that purpose in the data section must be named `f`.

```
# Compute the solution.
for (i in 1:steps)
    x[i+1] <- RK4(x[i],t[i],t[i+1]-t[i])
```

User-defined functions are called in exactly the same manner as functions that are predefined. In structural terms, `RK4` is identical to the predefined R functions that compute probability distributions. It differs from the `optimize` function used in script 2-4 only in that the latter outputs a data frame rather than a single variable. Note that `RK4` computes one step in the approximation; its inputs are the current values of the variable `x` and time `t` and the length of the time step for the next interval.

6-1 Discrete Linear Systems (scalar notation)

This script runs a simulation for a pair of discrete dynamic variable and plots the results. It will be superseded by script 6-3.

```
# Set model parameters.
——— ———

# Set initial condition and total time.
——— ———
```

It is good programming practice to test scripts on problems for which you already know the answer. Equation (6.1.4) in the book provides a model that is thoroughly analyzed in subsequent examples.

```
# Set the top value for the y axis.  Use 0 to accept the default.
ymax <- 0
```

The R default axis limits do not always work well with this script, and we need a sophisticated way of working around the difficulty. It may be necessary to choose the maximum y value, but we can't know what is a good choice without seeing the plot. The output part of the script contains a statement that establishes a default upper bound for the plot. Setting `ymax` to 0 accepts this default; if need be, you can improve the plot by choosing a specific value.

```
# Define t and create data structures.
___ ___

# Compute population values.
for (i in 1:T)
    {
    ___
    ___
    }
```

These lines of code are easily adapted from script 5-1.

```
# Compute growth rates.
x <- 2:T
y1 <- J[x]/J[x-1]
y2 <- ___
```

The ratios of successive population values are meaningful in discrete linear systems, and these are the quantities that should be used for the plot. Note that the list of ratios begins with J_2/J_1 and ends with J_T/J_{T-1}.

```
# Set maximum y value for plot.
if (ymax==0)
    ymax <- 2.5*y1[T-1]
```

The conditional structure is used here so that the calculation occurs only when the earlier instruction `ymax <- 0` was used to indicate that the default should be used. The default simply makes sure that the long-term growth rate is visible in the plot. In many cases, the default choice for the axis limit is larger than is necessary.

```
# Plot growth rates.
plot(___ ___,col="blue3")
points(___ ___,col="red3")
```

We have previously used `lines` to add curves to existing plots. Sets of points are added with `points`. The arguments are the same, except of course that we would not specify a line style. There are optional arguments that can be used to change the style of the points.

6-3 Discrete Linear Systems

This script improves on script 6-1 by generalizing to systems of any size and adding code to find the long-term stable growth rate and population ratios.

```
# Set model parameters.
r1 <- c(  0, 104, 160)
r2 <- c(.01,   0,   0)
r3 <- c(  0,  .3,   0)
M <- rbind(r1,r2,r3)

# Set initial condition and total time.
x0 <- ___
```

A discrete linear system is defined by the matrix M of coefficients. An efficient was to assemble a matrix from data in R is to define each of the rows and then bind them together into the desired matrix. The specific example used here is Example 6.1.5 from the book. The extra spaces in the definitions of the rows are not needed, but they make it easier to identify the model. Code in the data section should be as human-readable as possible. Note that the initial condition is a list of n values, one for each component of the population.

```
#   x is the vector of current populations.
#   X is a matrix with the population vectors as columns.
#   N is the vector of total populations.
#   L is the ratio L[j]=N[j+1]/N[j]
x <- ___
X <- matrix(0,nrow=length(x0),ncol=T+1)
X[,1] <- ___
N <- rep(___,___)
L <- rep(0,___)
```

We want to save a vector of population values for each time step, which requires a more sophisticated data structure than any of the previous scripts. As defined here, `x` is the variable that gets updated with current populations at each time step. These population values are also recorded in a matrix `X`, which saves the initial population vector as column 1 and the population at time t as column $t + 1$. Two instructions are used to create the data structure, with initial values all 0, and then record the list of initial populations in the first column. The list `N` is used to keep track of the total population at each time step, so it must be initialized as a list with a component for each time, including 0. Be careful to initialize it with the correct value. The list `L` is the list of total population growth rates, which should converge to the long-term stable growth rate λ_1. Since growth rates are ratios of successive populations, we do not have a value for time 0; hence the list has only `T` entries and no meaningful initial value.

```
# Compute the long-term behavior.
result <- eigen(M)
```

```
evals <- result$val
evecs <- result$vec
lambda <- evals[1]
proportions <- evecs[,1]/sum(evecs[,1])
```

`eigen` is a built-in R function that produces a data frame of output values. These are computed in the first instruction and then unpacked in the next two. The function automatically sorts the eigenvalues by magnitude, so the long-term growth rate is always the first eigenvalue. The long-term proportions must be computed from the corresponding eigenvector; dividing by the sum of the components of that vector ensures that the total is 1. Alternatively, we could divide by `evecs[1,1]` to obtain a list of ratios of stage populations to the population of the last stage.

```
# Compute population values and update data structures.
for (i in 1:T)
    {
    x <- M %*% x
    X[,i+1] <- x
    N[i+1] <- ___
    L[i] <- ___
    }
```

Matrix multiplication in R uses the symbol `%*%` rather than `*`. The following instruction saves the new population vector in the appropriate column of `X`. The final two instructions record the total population and growth rate in the next position of the list; recall that `N[1]` is needed to record the initial population, whereas `L[1]` is used for the ratio of population at time 1 to that at time 0.

```
    ymax <- max(L)
```

Since only total populations are used for the plot, there is no risk of an infinite ratio of successive populations; hence, the largest ratio is a reasonable choice for the maximum axis limit. If the initial conditions are far from the stable stage structure, there could be very large values at the beginning, which would lead to a graph with axis limits too large. In this case, the parameter `ymax` should be chosen to make a meaningful graph rather than using the default.

```
# Plot growth rates.
plot(t[-1],___ ___)
```

Note that there is no ratio for time 0; hence, the list of times that correspond to values of `L` begins with time 1 rather than time 0. The list `t[-1]` is the list of all times except the first one.

7-X Continuous Systems

This script generalizes script 5-3 for use with continuous dynamical systems.

```
## MODEL

# xx is the vector of current solution values
# tt is the current time
# yy is the vector of derivatives

# solution components are extracted from xx
# derivatives yy are computed from components and
#    parameters

f <- function(xx,tt)
        {
        X <- xx[1]
        Y <- xx[2]
        Z <- xx[3]
        yy <- rep(0,length(xx))
        yy[1] <- ___
        yy[2] <- ___
        yy[3] <- ___
        return(yy)
        }
```

This section replaces the single-line function in script 5-3. It is easy to get confused between the names of scalar components of a system and the name used for the collection of these components. I use `xx` and `yy` in this function rather than `x` and `y` because we might have scalar variables with the latter names. The function `f` must compute the derivatives for all of the differential equations and return the results as a list. While we could use the scalar components as inputs, it is good programming practice to make these a list as well; this means that the instructions that call the function `f` do not need to be modified to account for different names or numbers of scalar components. The drawback of passing all variable values in through a list is that the individual variables names are lost. For easy readability, I like to include instructions that unpack the variable list `x` into its components, which will match the notation for the specific model. The right hand sides of the differential equations are then calculated individually, with the formula for the derivative of the jth variable defined to be `yy[`j`]`. The derivative formula list is then returned as the value of the function.

```
# Create a data structure for the solution.
x <- ___
results <- ___
results[,1] <- ___
```

The data structures are similar to that used in script 6-3.

```
# The RK4 function computes one step of the solution of a
#   discrete approximation to an initial value problem
#   x'=f(x,t), x(0)=x0, where x and f could be vectors.
#   The inputs are the starting values, time, and time step.

RK4 <- function(x0,t,dt)
```

The RK4 function used here is exactly the same as that of script 5-3. No changes are needed to use the function when x is a list rather than a single variable.

```
# Compute solution values and update data structure.
for (i in 1:steps)
    {
    x <- ___
    ___
    }
```

This section of code is similar to that of script 5-3 in computation and 6-3 in structure, with results playing the role played by X in script 6-3.

```
# Set maximum y value for plot.
___ ___
```

The maximum value for the plot should be the largest value in the matrix of results.

```
# Unpack results matrix.
X <- results[1,]
___ ___
```

The results should be unpacked in the same fashion as xx is unpacked in the function f. This makes it easier to set up the plot instructions. Note that the details of the instructions in this output section are model-dependent; hence, the output section of this script needs to be modified each time the script is adapted to a new model.

8-1 Discrete Systems

This script does not exist. Create it using elements of scripts 6-2 and 7-X.