R Tutorial 1

Introduction to Computational Science:
Modeling and Simulation for the Sciences, 2nd Edition

Angela B. Shiflet and George W. Shiflet
Wofford College

R materials by Stephen Davies, University of Mary Washington
stephen@umw.edu

Introduction

R is one of the most powerful languages in the world for computational science. It is used by thousands of scientists, researchers, statisticians, and mathematicians across the globe, and also by corporations such as Google, Microsoft, the Mozilla foundation, the New York Times, and Facebook. It combines the power and flexibility of a full-fledged programming language with an exhaustive battery of statistical analysis functions, object-oriented support, and eye-popping, multi-colored, customizable graphics.

R is also *open source!* This means two important things: (1) R is, and always will be, absolutely **free**, and (2) it is supported by a great body of collaborating developers, who are continually improving R and adding to its repertoire of features. To find out more about how you can download, install, use, and contribute, to R, see http://www.r-project.org.

Getting started

Make sure that the R application is open, and that you have access to the R Console window. For the following material, at a **prompt** of >, type each example; and evaluate the statement in the Console window. To evaluate a command, press ENTER. In this document (but not in R), input is in red, and the resulting output is in blue.

We start by evaluating 12-factorial (also written "12!"), which is the product of the positive integers from 1 through 12. We can use R's built-in *factorial*() function[1] to accomplish this by typing "factorial(12)." After pressing ENTER or RETURN for the first computation, R returns the result, as the following prompt (>), input (in red), and answer (in blue) indicate:

```
> factorial(12)
[1] 479001600
```

Notice that the number "1" appears in brackets before the answer. You can ignore this fact for now; for the curious, this is because in R, all values are *vector* instead of *scalar*

1 An R "function" can be thought of similarly to a mathematical function: it gives an output for certain input(s). When referring to R functions, we typically include an empty pair of parentheses immediately after the function name, as in sin() or cos(), to make clear that it is a function rather than a simple variable.

quantities, and so what *factorial*(12) actually produces is a vector with one element (namely, 479001600). For 25!, the following output appears in exponential notation with e+25 indicating multiplication by 10^{25}, so that the value is 1.5511×10^{25}:

```
> factorial(25)
[1] 1.551121e+25
```

Quick Review Question 1 Evaluate 100-factorial by typing the appropriate code at the R console.

R Programs

Many simple operations can be carried out directly at the R console, but often it makes more sense to write an R program (sometimes also called a "script") that contains multiple different commands organized to carry out some more complex operations. You can open a new script in Windows or Mac by using the "***New***" command from the "***File***" menu. (In Linux, you will simply use your favorite text editor, making sure to save the file in the R working directory.)

The first part of an R program is often an explanation of what the program does, for instance:

```
# hurricaneSimulation.R
# Simulate a category-5 tropical storm to 100-km accuracy
# Filbert Lewis
# Jan.10, 2015
```

The lines above are referred to as **comments**. Frequently, and especially when writing R programs (see below), we need a "comment," or explanation of material, which we do not want R to execute. We begin a comment line with a **# sign** (pronounced "pound" or "hash" or "tic-tac"), as above. R will ignore everything on that line, considering it fit for only humans to read.

To save a file, we use the familiar "***File***" menu, entering a file name to which R appends the extension **.R**. ***Save often***, particularly before you print, do a long calculation, or display a graph.

To execute an entire R program, use the ***source*** command from the R console, like this:

```
> source("filename.R");
```

For example, if you saved a program under the name "hurricaneSimulation," you would run it like this:

```
> source("hurricaneSimulation.R");
```

Sometimes, it's helpful to have R print out the evaluation of every command in the .R program file as it runs. To do this, we can include the text ",print.eval=TRUE" after the name of the file, like this:

```
> source("hurricaneSimulation.R",print.eval=TRUE);
```

Note that R program files normally don't have spaces in them, since this can cause problems. It's a good idea to capitalize the first letter of each successive word, as in the hurricaneSimulation example, above. (This is called "camel case" for obvious reasons.)

If, when you type the *source()* command, you get an error message saying that R could not find the file (sometimes this error message says something like "cannot open the connection"), one of three things is probably wrong: (a) you mistyped the filename, or (b) you forgot to type the ".R" extension as part of the source command, or (c) the folder/directory that R is using is different from the folder/directory that contains the file. To diagnose the latter, type the command "*getwd()*" in the R console (or under the ***Misc*** menu, select ***Get Working Directory***), and press ENTER. This command will print out the name of the folder/directory that R is looking in. If it does not match the name of the folder/directory you saved your file in, then this is the problem. You have two choices: either tell R to change to the correct folder/directory, or else move your file to the folder/directory R is using. The latter is probably simpler, although if you do this you will need to remember to save all of your files into that new folder/directory instead of your original one. To do the first option, under the *Misc* menu, select ***Change Working Directory…***, maneuver to the desired directory, and select *Open*.

Quick Review Question 2

a. Copy the *RCTTutorial1.R* file to your R programs folder from the course materials you downloaded and unzipped. (Note that the file should be copied into your working directory. A common source of error is to save a file in a *different* location than where R is looking for it, so be careful.)

b. Open the file in R by using the "***Open...***" command from the "***File***" menu. (In Linux, you will simply use your favorite text editor to edit the file.)

c. Skim the contents of this file. Note that it has a section of comments demarking each Quick Review Question. From the R console, copy your code from Quick Review Question 1 (using the "***Edit***" menu) and paste it into the appropriate place in the file (namely, immediately below the comment line that begins "Quick Review Question 1".) This part of the file should now look like this:

```
# Quick Review Question 1   Evaluate 100-factorial.
factorial(100)
```

Note carefully that the "factorial(100)" line *does not* have a ">" sign before it: the ">" sign is simply a prompt that *R* prints to the console before we type an interactive command. Note also that the line *does not* have a "#" sign before it either, since it is not a comment: we actually want R to execute that command.

d. In the file, find the section for Quick Review Question 2, and add comment lines as it describes.

e. Save your file.

f. Execute the file *RCTTutorial1.R* using the `source` command, with the "`,print.eval=TRUE`" included, as described above. Make sure the answer to the factorial question appears on the R console (and nothing else.) If R cannot find your file, refer to the last paragraph of the "R Programs" section.

g. **Throughout the rest of this tutorial, record all your answers to the Quick Response Questions in *RCTTutorial1.R*.**

Numbers and Arithmetic Operations

Scalar multiplication in R is indicated by an **asterisk** (*) (also called a "star" or "splat.") Thus, to multiply 5 times 18, we write `5*18`, not 5(18) or 5x18. The addition, subtraction, and division operators are `+`, `-`, and `/`, respectively. The operator **`%%`** (called "modulus") will give the remainder after dividing two integers. (For instance, `11%%3` gives `2` because 11 divided by 3 is three, with a remainder of two.) An expression can be raised to a power using a **caret** (^) (also called a "hat") as follows:

```
> 3 * (5/8 - 1.25)^2
```

Quick Review Question 3 In the appropriate place in your *RCTTutorial1.R* file, type commands that will add the fractions (not decimal numbers) one-half and three-fourths. In the R console, execute your file (using `source`, with "`,print.eval=TRUE`") as you did before. Note that R prints a decimal expansion instead of the fractions. (As with all Quick Review Questions, paste your code that added one-half and three-fourths to the appropriate place in the *RCTTutorial1.R* file, right below the comment lines that begin "Quick Review Question 3."

R has numerous built-in functions, such as `sin()`, and built-in constants, such as `pi` representing π. A function usually has one or more inputs, also called "arguments" or "parameters." To use a function, you type the name of the function, followed by parentheses around the argument(s). (Multiple arguments should be separated by commas.) For instance, to compute 5 sin(π/3):

```
> 5 * sin(pi/3)
```

(Again, the asterisk for multiplication is mandatory.)

Quick Review Question 4 ***log10*(*x*)** is the common logarithm of *x*, or logarithm to the base 10. Evaluate the common logarithm of 23.4, and by copying and pasting, record your answer in *RCTTutorial1.R*.

Quick Review Question 5 ***log*(*x*)** is the natural logarithm of *x*, usually written as ln(*x*) in mathematical notation. Evaluate the sine of the natural logarithm of 23.4, and by copying and pasting, record the code to do so in *RCTTutorial1.R*.

Quick Review Question 6 e^x is ***exp*(*x*)** in R. Evaluate the number e^2, and by copying and pasting, record your code in *RCTTutorial1.R*.

Variables and Assignments

We can employ **variables** to store values for future use. Variable names must begin with a letter, and be composed of letters, digits, periods, and underscores. Variable names are case-sensitive in R, meaning that variables named "length," "Length," and "LENGTH" will all refer to *different* variables. As mentioned above, the use of camelCase is recommended for variable names comprised of multiple words.

We can **assign** a value of an expression to a variable this way:

variableName = expression

This sets the value of the variable called *variableName* to whatever *expression* evaluates to. For example, the following assignment sets the variable called *lengthOfBridge* to 15:

```
> lengthOfBridge = 15
```

R calculates the value of the expression on the right, such as 15, and then assigns the value to the variable on the left, such as *lengthOfBridge*. If we subsequently refer to *lengthOfBridge*, R replaces *lengthOfBridge* with its value (15, or whatever it might have been set to subsequently.)

Note that the statement "lengthOfBridge = 15" is *not* an **assertion** that the value of the variable *is* 15, but rather a **command** that instructs R to *make* the value be 15. For this reason, the use of the equals sign in this syntax is somewhat misleading, and some R programmers prefer to substitute the key sequence "**<-**" instead, as in:

```
> lengthOfBridge <- 15
```

(These two statements are equivalent.)

Quick Review Question 7

a. In the console window, assign 4.8 to the variable time and execute the statement (press ENTER.)

b. Type *time* at the console and execute. Note how the value was stored and recalled.

c. Type *time* + 3 at the console and execute.

d. Type *time* at the console and execute. Note that the execution of step c did not change the value of time. This is because R was not told to change time to hold a new value, but simply to compute the value of *time* + 3 (and then do nothing with it.)

e. Now type *time* = *time* + 3 at the console and execute.

f. Type *time* at the console and execute. Note that the variable's value has now been updated as desired.

g. Evaluate $time^3$.

h. Paste all the code for these steps into *RCTTutorial1.R*.

If we use a variable before it has been assigned a value, R returns an error message, such as "Error: object ' silly ' not found".

We can **clear** a variable (i.e., erase its value) by using the "*rm*" (for "remove") command. The argument is the name of the variable we want to clear. For instance, if we have a variable x whose value is 17.5, and we no longer want there to *be* a variable x at all, we can simply type:

```
> rm(x)
```

At any time, we can see what variables are defined via the "*ls*" (for "list") command:

```
> ls()
```

A common technique in R programming is to use a variable to cumulatively sum a quantity. In these situations, we need the old value of the variable to compute the new value of the same variable, as in:

```
> numberOfBirths = numberOfBirths + birthsThisYear
```

A variation of this technique is to "count" some continuously incrementing quantity over time. For instance:

```
> day = day + 1
```

Again, we stress that these statements do not *declare* that a variable is equal to itself plus (say) one, but rather *instruct* R to set the variable to a new, higher, value.

Quick Review Question 8 Write a statement to assign 34 to variable *time* and then to add 0.5 to time, changing its value. Paste the code into *RCTTutorial1.R*.

In R, a variable can hold a whole sequence of numbers, rather than just a single number. A sequence of numbers is called a **vector**. One way to create a vector is by **c**ombining several individual numbers into a vector variable using the "*c*()" function. As an example, if we measured the heights (in meters) of various trees in a forest, we could create a variable called heights that could hold all of the values this way:

```
> heights = c(21.6, 22.5, 19.8, 20.5)
```

The heights variable now holds four different numbers, one for the height of each tree we measured. Typing the name of the variable shows all of its elements:

```
> heights
[1] 21.6 22.5 19.8 20.5
```

To extract the value of a *single* tree's height, we use the **square bracket notation** to specify the **index** of the value we want:

```
> heights[1]
[1] 21.6
> heights[3]
[1] 19.8
```

There are other ways of quickly creating sequences of numbers. The function "*seq*()" creates a vector with numbers in a **seq**uence, given a starting and ending number:

```
> years = seq(1994,2011)
> years
[1] 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008
[16] 2009 2010 2011
```

(Note that there are so many values in this vector that the elements "wrap around" to a second line. The number in brackets at the start of each line ([1], [16]) give the index of the element at the start of that line, for easy reference.)

Also, the **colon notation** can be used as a more compact way of defining these vectors. The expression "1994:2003" is a shorthand for "all the integers (whole numbers) between 1994:2003."

```
> years = 1994:2003
> years
[1] 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003
```

Returning to the *seq* function, we can also add a third argument to specify the **increment** (or "**step**") between values, if we want the values to increase by something other than 1. For instance:

```
> years = seq(1992,2012,4)
> years
[1] 1992 1996 2000 2004 2008 2012
```

Variables in R are not always numeric; they can also contain text data, known as "character strings." For instance, one could declare a variable this way:

```
> myName = "Stephen"
```

Note the use of double-quotes to specify character strings. If we had omitted the quotation marks in the previous example, R would have given an error message because it would have been looking for a *variable* named "Stephen" to assign to the variable myName, instead of treating "Stephen" as a value.

Quick Review Question 9 Use the *c*() function to create vector variables for each of the following:

a. A vector called *ages* with values 19, 21, 21, and 20.

b. A vector called names with values "Ruth," "Callixte," and "Talishia." (Note: this should be a vector of *three* values, the first of which is "Ruth" and the last of which is “Talishia.” It should *not* be a vector of just one value, whose value is “Ruth, Callixte, Talishia.”)

c. Paste your code for these items into the appropriate place in *RCTTutorial1.R*.

Quick Review Question 10 Use the *seq*() function to create vector variables for each of the following:

a. A vector called *itemNumbers* with all the whole numbers between (and including) 10000 and 10005.

b. A vector called *pipeFittings* with all the *even* numbers between (and including) 32 and 48.

c. A vector called *timeChecks* that contains numbers from 0 to 4, in increments of .25 (in other words, the vector should have the values 0, 0.25, 0.5, 0.75, 1, 1.25, etc.)

d. Now use the colon notation to recreate the *itemNumbers* vector (from step a) without the use of the *seq*() function.

e. Paste your code for these items into the appropriate place in *RCTTutorial1.R*.

User-defined functions

Frequently, we wish to define our own functions that we can use again and again. As with variable names, by convention, we begin the name of such a **user-defined function** with a lowercase letter.
Suppose we wish to define the function $sqr(x) = x^2$ in R. At the console, we can type:

```
> sqr = function(x) x^2
```

This tells R that a new function variable called *sqr* is to be set equal to a function of a variable x , and that the value of *sqr*(*x*) should be x^2.

We can “call” (or “invoke”) the function on a particular value just like we did for the built-in functions *factorial*(), *c*(), and *seq*():

```
> sqr(14)
[1] 196
```

User-defined functions longer than this are typically written in an R program file as part of a larger program, rather than typed into the console. However, the syntax is the same.

Quick Review Question 11

a. Define a function $quick(x) = 3\sin(x - 1) + 2$. (Be careful to remember how to express multiplication in R. Typing “3sin(x-1)+2” will not work.)

b. Evaluate the function at x = 5.

c. Paste the code for both the function and the evaluation into *RCTTutorial1.R*.

Online documentation

R has an extensive help system built in to the console that is very easy to use. To find out information about a particular function (whose name you know), you can type "**?**" followed by the name of the function. For instance,

```
> ?sin
```

brings up a page of information about sine, as well as other related trigonometric functions. (Linux users should press "q" (for "quit") to return to the R console.) Alternatively, from the ***Help*** menu, we can select ***R Help***, and then an option, such as ***Search Engine & Keywords***.

If you need help on a topic but aren't sure what the exact name of the R function is, you can also *search* the help system for a word or phrase. To do this, type "**??**" (*two* question marks) followed by your search string. You will be presented with a list of all the R help topics that contain that string. For instance, suppose you want to perform a statistical test of proportions (this is a statistical test to compare whether the proportions in two groups are significantly different from each other; for instance, whether the proportion of 15-year-olds who regularly play videogames differs from the proportion of 18-year-olds who do.) You aren't sure of the name of the R function, so you guess "proportion." Typing "?proportion," however, gives a "no documentation" message. So you decide to search the help system by typing:

```
> ??proportion
```

Scanning the results, you see that stats::prop.test is a function that will perform a "Test of Equal or Given Proportions." This tells you that prop.test() is the function you want to use. (Note that the "stats::" prefix is a **package name**. Packages in R are related sets of functions and data files that are grouped together. You don't need to type the name of the package to use a function, but you can.)

To get specific help about how to use the *prop.test*() function, you could then type:

```
> ?prop.test
```

(with just one question mark.)

Quick Review Question 12 At the R console, access the help page for the built-in function *log10*. Scroll through this help page, then return to the R console. Paste the code to do so *as a comment line* in *RCTTutorial1.R*.

Quick Review Question 13 Suppose you do not know the R command to perform a standard deviation. Typing "?standard" and "?deviation" do not yield any results. Use the help system to *search* for the word "deviation" and see if you can discover the name of the function. Record your findings in *RCTTutorial1.R*.

Displaying

Sometimes, particularly when doing error checking, we wish to display intermediate results. To do so, we can employ the *cat* function. ("cat" stands for "con**cat**enate.") *cat*'s arguments are a list of character strings that should be concatenated (or "stuck") together. In order to get cat to print a **newline** (carriage return / line feed) after it prints a result, the two-character sequence "**\n**" must be added. For instance, the following command prints a friendly hello:

```
> cat("Why hello there",myName,"!!\n")
Why hello there Stephen !!
```

Note that this command provided *three* character strings to stick together in the output: (1) "Why hello there," (2) the value of myName (which is "Stephen" or whatever that variable's current value is), and (3) a string with two exclamation marks and the newline sequence.

Numeric values, of course, can also be printed:

```
> gpa = 3.5
> year = 2011
> cat("Your GPA in",year,"was",gpa,".\n")
```

Your GPA in 2011 was 3.5.

Quick Review Question 14 Write a statement to assign the value 3 to *t*. Then, use the cat function to display "Velocity is", the result of the computation -9.8*t*, and "m/sec." The output for the command should be as follows:

```
Velocity is -29.4 m/sec.
```

Paste your code for this into *RCTTutorial1.R*.

Looping

It is often advantageous to be able to execute a segment of code a number of times. For example, to obtain the velocity for each integer time ranging from 1 to 1000 seconds, it would be inconvenient for the user to have to execute one thousand statements assigning a time and computing the corresponding velocity. Some method of automating the procedure is far more preferable. A segment of code that is executed repeatedly is called a **loop**.

Two types of loops exist in R. When we know exactly how many times to execute the loop, the **for loop** is a good choice. One form of the command is as follows:

```
for (i in min:max) {
    expr
}
```

Recall that the notation *min:max* is a shorthand for *seq(min,max)*, meaning that it represents a vector containing the *entire* sequence of numbers from *min* to *max*, including both end points. In this loop, then, the **index** or **loop variable** is *i*, and *i* takes on integer values from this sequence. For *each* value of *i*, the computer executes the body of the loop, which is *expr*, the statements between the curly braces.

For example, suppose, as the basis for a more involved program, we wish to increment a variable called *dist* (for "distance") by 2.25 seven times. We initialize the variable to 0. Within a for loop that executes 7 times, we calculate the sum of *dist* and 2.25, and assign the result of the expression to *dist*, giving the variable an updated value. Because the for loop does not return a value and we are not printing intermediate values of dist, we display the final value of dist after the loop with the *cat*() function:

```
dist = 0
for (i in 1:7) {
    dist = dist + 2.25
}
cat(dist,"\n")
15.75
```

Quick Review Question 15 Write a segment of code to assign 1 to a variable *d*. In a loop that executes 10 times, change the value of *d* to be double what it was before the previous iteration. After the loop, type *cat*(*d*,"\n") to display *d*'s final value. Before executing the loop, determine the final value on your own so you can check your work. Then test your code, and paste it into the appropriate place in *RCTTutorial1.R*.

In the next loop, we increment *dist* by 2.25, compute time as $\frac{24.5 - \sqrt{600.25 - 19.6 dist}}{9.8}$, and then print out the distance and time for each time step.

```
dist = 0
for (i in 1:7) {
    dist = dist + 2.25
    t = (24.5 - sqrt(600.25 - 19.6 * dist))/9.8
    cat("For distance",dist,", time = ",t,"\n")
}
For distance 2.25 , time = 0.0935885
For distance 4.5 , time = 0.1909672
For distance 6.75 , time = 0.2926376
For distance 9 , time = 0.3992227
For distance 11.25 , time = 0.5115127
For distance 13.5 , time = 0.6305354
For distance 15.75 , time = 0.7576699
```

(A couple of notes on printing output: (1) notice that an extra space appears between the distance and the comma (",") in the above output. This is because by default, the "*cat*()" function pads its concatenated outputs with spaces. If you want to suppress these extraneous spaces, you can add the argument "sep=""" to the *cat*() command. (2) The time values are printed with many decimal places. To round to (say) two decimal places, you can use the *round*() function, passing the value you want rounded as the first argument, and the number of decimal places as the second argument. This modified program, and its output, appears below:)

```
dist = 0
for (i in 1:7) {
    dist = dist + 2.25
    t = (24.5 - sqrt(600.25 - 19.6 * dist))/9.8
    cat("For distance ",dist,", time = ",round(t,2),"\n",sep="")
}
For distance 2.25, time = 0.09
For distance 4.5, time = 0.19
For distance 6.75, time = 0.29
For distance 9, time = 0.4
For distance 11.25, time = 0.51
For distance 13.5, time = 0.63
For distance 15.75, time = 0.76
```

The previous example did not use the loop index in the loop's body. However, the following example of *cat*() and *for* with an index *i* displays both *i* and *i*-factorial (*i*!) with spaces between the values for *i*, going from 1 through 9:

```
for (i in 1:9) {
    cat(i," ",factorial(i),"\n")
}
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
```

To start the display with the value of 0!, which is 1, we indicate a beginning value of 0, as follows:

```
for (i in 0:9) {
cat(i," ",factorial(i),"\n")
}
```

Quick Review Question 16 For this question, complete another version of the code segment above that displays distance and time. In this version, do not initialize dist. Employ a loop with an index i that takes on integer values from 1 through 7. Within the loop, the value of dist is computed as 2.25*i*.

The code below is provided as a starting point in *RCTTutorial1.R*. After replacing each xxxxxxxx with the proper code, execute the program, and compare the results with the similar segment above.

```
for xxxxxxxxxx
    dist = xxxxxxxxxx
    t = (24.5 - sqrt(600.25 - 19.6 * dist))/9.8
    cat("For distance ",dist,", time = ",round(t,2)," seconds.\n")
xxxxxxxxxx
```

Quick Review Question 17

- **a.** Define the function qrq17 to be ln(3x + 2). Recall that log is the R function for the natural logarithm.
- **b.** Write a loop that prints the value of *k* and *qrq17*(*k*) for *k* taking on integer values from 1 through 8.
- **c.** Put all of this code into *RCTTutorial1.R*.

The other main type of loop that R supports is the **while loop**, which we will cover in future modules.

Plotting

We employ the "*plot*()" command for graphing two-dimensional functions. First, we establish a sequence of values for the independent variable, such as *t*. Recall that we employed sequences earlier as vector variables and in loop iterations. For example, "0:9" indicates the sequence 0, 1, ..., 9. For a smooth display of a graph, however, we need to plot additional points. To indicate a step size of 0.1, we could call the function like seq(0,9,0.1). The following statement assigns such a sequence to *t*:

```
> t = seq(-1,2,.1)
> t
[1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1 0.0 0.1 0.2 0.3 0.4
[16] 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
[31] 2.0
```

Now that we have an independent variable, we can use the *plot*() function. The basic form of the command gives the independent variable (such as *x* and the function (such as *g*()) to graph, as follows:

plot(x, $g(x)$)

For example, the following command graphs *cos*(t) with t varying from -1 to 2 by 0.1:

```
> t = seq(-1,2,.1)
> plot(t,cos(t))
```

R displays the resulting graphics in a figure window.

Quick Review Question 18 Graph $e^{\sin(x)}$ from -3 to 3. Experiment with different step increments, to see how it affects the graph display. Paste the code to generate this plot with 0.1 increments into *RCTTutorial1.R*.

We can indicate additional options, and R revises the display to reflect our changes. For example, ***xlab*** and ***ylab*** options generate axes labels, which should appear in all scientific graphics. The following command will annotate the graph with sensible labels:

```
> plot(t,cos(t),xlab="t",ylab="cos(t)")
```

Quick Review Question 19 Adjust your answer to the previous Quick Review Question to label the *x* and *y* axes appropriately. Paste the modified code into *RCTTutorial1.R*.

We can plot several functions on the same graph by using the *points*() function immediately after our initial *plot*(). The arguments are the same:

```
> plot(t,cos(t))
> points(t,.5*t+.4)
```

The second command in this sequence adds points to the plot according to the function $y(t) = .5t + .4$. We could change the shape (say, to the letter 'X') and color (say, to purple) of these new points by including those arguments to *points*() as well:

```
> plot(t,cos(t))
> points(t,.5*t+.4,pch='x',col='purple')
```

pch (plot character), ***col*** (color), *xlab*, and *ylab* (labels) are just a few of the many different parameters you can give to R graphics functions to format your output just the way you want it.

Differentiation (Optional)

If we have a function $f(x)$, we can numerically compute its derivative in R using the *D* function. *D* takes two arguments: an "expression" (i.e., the definition of the function as a symbolic expression) and the independent variable, enclosed in double-quotes. This function returns an object which can be evaluated using the "*eval*()" function for specific values of the independent variable.

For example, consider the following function for the height of a ball thrown into the air:

$$y = -4.9t^2 + 15t + 11$$

First we create the function itself:

```
> y = function(t) -4.9*t^2+15*t+11
```

Then, to differentiate it, we need to specify an ***expression*** (instead of a function):

```
> yexp = expression(-4.9*t^2+15*t+11)
```

and then differentiate it using the *D* function. The first argument is the expression to differentiate, and the second is a character string giving the independent variable:

```
> dy = D(yexp,"t")
> dy
15 - 4.9 * (2 * t)
```

Note that this is the symbolic derivative of the original *y* function.

Now, we create a vector of *t* values (specifying the starting and ending values, and the increment (time step)):

```
> t = seq(0,4,.1)
```

And now we can plot the original function *y* in one color, and its derivative (using *eval*) in another:

```
> plot(t,y(t),col="green")
> points(t,eval(dy),col="blue")
```

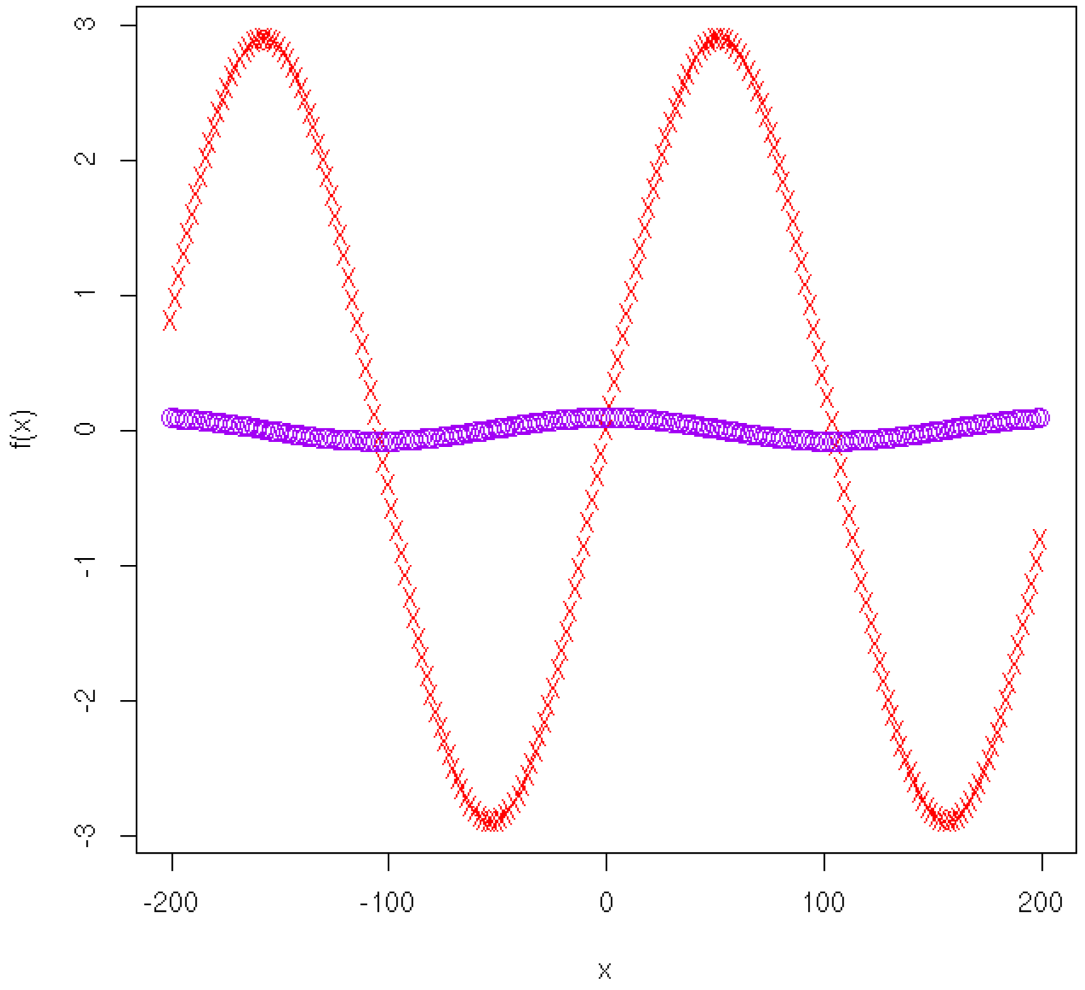
The green dots represent the *y* function evaluated at each *t* point, and the blue dots represent its derivative at each *t* point. Observe that the green dots show the trajectory of the thrown ball over time, while the blue dots show its velocity (rate of change). The blue dots continually decrease (indicating that the ball is being pulled down by the force of gravity) and they cross the *x*-axis (i.e., the derivative is zero) precisely where the green dots reach their maximum (i.e., where the ball ascends to its highest height, and is about to fall again.)

Quick Review Question 20 Give R code to do the following:

- **a.** Create a function $y = 2.9 \sin(0.03x)$.
- **b.** Create an expression *fexp* $= y = 2.9 \sin(0.03x)$. so that it can be differentiated.
- **c.** Differentiate the expression using the *D* function (don't forget the second argument.) Assign that differentiated expression to a variable *df*.
- **d.** Create a vector called *x* that contains values from -200 to 200 by twos.
- **e.** Plot the function *f* vs the *x* vector. Use parameters *pch* and *col* to draw the plot points with red X's.
- **f.** Add to the plot the derivative *df* vs the *x* vector. Use parameters *pch* and *col* to draw the plot points with purple O's. (Don't forget you'll need to use the eval function.)

If you did the above steps correctly, your plot should look like Figure 1. Paste this code into *RCTTutorial1.R*.

Figure 1 Output for Quick Review Question 20.



Integration (Optional)

If we have a function $f(x)$, we can numerically integrate it in R using the *integrate* function. *integrate* takes three arguments: a function, the starting x value (i.e., lower boundary) of the integration, and the ending x value (i.e., upper boundary) of the integration.

For instance, to find the value of $\int_0^5 (-t^2 + 10t + 24)\, dt$, we type:

```
> f = function(t) -t^2+10*t+24
> integrate(f,0,5)
203.3333 with absolute error < 2.3e-12
```

The message about the "absolute error" tells us how accurate the numerical integration was (in this case, it is accurate to nearly 12 decimal places.)

Quick Review Question 21 Compute the definite integral of $sin^2(x)$ from 0 to 2π. Paste this code into *RCTTutorial1.R*.

Checking your work

At this point, you should be able to execute your *RCTTutorial1.R* program and get output similar to the following:

```
> source("RCTTutorial1.R",print.eval=TRUE)
[1] 9.332622e+157
[1] 1.25
[1] 1.369216
[1] -0.01114314
[1] 7.389056
[1] 4.8
[1] 7.8
[1] 4.8
[1] 7.8
[1] 474.552
[1] -0.2704075
Velocity is -29.4 m/sec.
1024
For distance 2.25, time = 0.09 seconds.
For distance 4.5, time = 0.19 seconds.
For distance 6.75, time = 0.29 seconds.
For distance 9, time = 0.4 seconds.
For distance 11.25, time = 0.51 seconds.
For distance 13.5, time = 0.63 seconds.
For distance 15.75, time = 0.76 seconds.
k=1, qrq17(k)=1.609438
k=2, qrq17(k)=2.079442
k=3, qrq17(k)=2.397895
k=4, qrq17(k)=2.639057
k=5, qrq17(k)=2.833213
k=6, qrq17(k)=2.995732
k=7, qrq17(k)=3.135494
k=8, qrq17(k)=3.258097
3.141593 with absolute error < 2.3e-09
```

with a plot similiar to Figure 1 appearing.[2]

Additional Features

In order to save typing, R allows you to scroll backwards through the previous commands you have entered. You do this by pressing the **up arrow**, ↑, once per command. Each time you press it, you go to the previous command you entered. Using the left (←) and right (→) arrows, you can edit one of these lines in order to enter a modified command. Pressing ENTER (even when the cursor is not at the end of the line) will actually carry out the command.

2 Note that every time you call the plot() function, the contents of the new plot completely replace any previous plot. You can instead cause new plots to appear in new windows by calling x11() (Linux/Mac) or windows() (Windows) immediately before calling plot() a second or subsequent time.

Quick Review Question 22 Using the arrow keys, go back to the *factorial*(25) command you entered much earlier in this tutorial, change it to evaluate 24-factorial, and execute the command.

R also will **tab complete** the names of functions, parameters, and other system objects so that you don't have to type long names in their entirety (or guess their spelling). This is a very useful feature that can save you time and make you more productive as you use R. As an example, *as.integer*() is an R function that will convert textual data to numeric data. (For instance, *as.integer*("4500") will convert the text string "4500" into the integer 4,500.) Instead of typing "as.integer" in its entirety, you can type "as.i" and then press the TAB key. The remaining letters in "as.integer" will automatically appear.

As another example, suppose you want to run a proportion test (such as in the videogame example earlier in this tutorial.) Recall that the R function to perform this test is called *prop.test*. If you type "prop" and then press TAB, R will complete the next two characters ".t" because those come next. However, there are also other R functions that begin with the letters "prop.t" and so R only completes as much as it knows. If you now press TAB *twice*, R will show you all of these functions, so that you can decide which one you wish to type.

Now suppose you wish to set the "alternative" parameter to this function, in order to specify an alternative hypothesis. Once you have "prop.test(" typed in its entirety, you can type "al" followed by TAB. The name of this entire parameter will appear without you having to type it.

It may seem like a small thing to save a few characters of typing. However, the ability to quickly press TAB and have R complete your intention, mistake-free, can be very powerful when used consistently. It also saves you from having to remember the exact spelling and phrasing of every command and parameter.

Quick Review Question 23

a. Let R tab-complete the name of the "aggregate" function for you. ("aggregate" is a built-in R function for splitting data into subsets and giving a summary of each subset.) At the R console, type "agg" followed by TAB. Notice that the entire word "aggregate" appeared.

b. Now type a parenthesis to begin the parameter list "(". You want to specify a value for the "formula" parameter. Type "for" and press TAB. Notice that nothing happened — this is because R can't guess what you want to type, since more than one parameter begins with the letters "for". Press TAB a second time. Now you see a list of all the parameters that begin with the letters "for". You see "formula" in the list, so type "m" and press TAB once more. Now the whole word "formula" appears.

Quick Review Question 24 At the R console, set a variable called *touchdownPercentage* to the value .15. Now suppose you want to increase the value of *touchdownPercentage* by .1. Using tab-completion, execute the R command

```
        touchdownPercentage = touchdownPercentage + .1
```

(Hint: type "touch" and press TAB each time you need to type the variable name.)

We can select text and **cut**, **copy**, and **paste** it using items from the "***Edit***" menu or the shortcuts indicated on that menu.

To quit R, close the application as you normally close others on your operating system (perhaps by clicking an "X" icon, for example.) You will be prompted as to whether you wish to save your current workspace. A "workspace" is a collection of R objects that you have created while you've worked in R. (You can see a list of these objects by typing *ls()* at the R console.) For now, when you quit, you won't need to worry about saving your workspace. However, when you begin to write real programs, it may be advantageous to save your workspace between sessions. This way, when you return to R, all of your work will be restored just as if you'd never quit.