8.1 – R Computational Toolbox Tutorial 3

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

1 Introduction

We recommend that you work through this tutorial with a copy of R, answering all Quick Review Questions as you go.

The prerequisite to this tutorial are "R Computational Toolbox Tutorial 1" and "R Computational Toolbox Tutorial 2." Before proceeding, follow those tutorials in their entirety, including installing R on your system and answering all Quick Review Questions as you go. Tutorial 3 prepares you to use R for material in this and subsequent chapters. It introduces the following functions and concepts: vector operations, transpose; additional graphics options, such as for equal axes, plotting area, line width, and line style; fitting; rule substitutions; reading data files; and logarithms.

2 Vector operations

Vectors are essential to R, and we can perform operations on entire vectors. To perform addition, subtraction, multiplication, division, or exponentiation of a scalar (number) by every element in a vector, we use the usual operator of +, -, *, /, or ^, respectively. For example, consider the following vector:

```
> vec = seq(4, 5.2, .3)
> vec
[1] 4.0 4.3 4.6 4.9 5.2
```

The following input statements (in red) perform the same operation on every element of vec, returning the appropriate output array (in blue) without changing the value of vec:

```
> vec + 10
[1] 14.0 14.3 14.6 14.9 15.2
> 3 + vec
[1] 7.0 7.3 7.6 7.9 8.2
> vec - 5
[1] -1.0 -0.7 -0.4 -0.1 0.2
> vec * 10
[1] 40 43 46 49 52
> vec / 10
[1] 0.40 0.43 0.46 0.49 0.52
> vec ^ 3
```

```
[1] 64.000 79.507 97.336 117.649 140.608
> 2 ^ vec
[1] 16.00000 19.69831 24.25147 29.85706 36.75835
```

To change the value of vec, we must assign the result of an operation to vec, such as follows:

```
> vec = vec * 10
> vec
[1] 40 43 46 49 52
```

These techniques apply not only to 1-dimensional vectors, but also to multi-dimensional matrices and arrays. For example:

```
> mat = matrix(c(3,5,2,4,4,1),nrow=3)
> mat
     [,1] [,2]
[1,]    3    4
[2,]    5    4
[3,]    2    1
> mat + 7
     [,1] [,2]
[1,]   10   11
[2,]   12   11
[3,]    9    8
```

As the following Quick Review Question illustrates, we can also apply other functions, such as the square root function (*sqrt()*), that operate on a single number to every element of a vector.

Quick Review Question 1 Create a new script / program file. In opening comments, put "R CT Tutorial 3 Answers" and your name. Save the file under the name *RCTTutorial3.R*. In the file, preface this and all subsequent Quick Review Questions with a comment that has "QRQ" and the question number, such as follows:

```
# QRQ 1
```

a. With one assignment statement, make *qrq* be a 2-by-4 matrix of all zeros.

b. With one assignment statement, make the first row of *qrq* be the sequence of positive integers 1, 3, 5, 7.

c. Return the product of 3 by every element of *qrq* without changing *qrq*.

d. Return the square root of every element of *qrq* without changing *qrq*.

e. Add 2 to every element of *qrq*, changing the value of *qrq* to hold those increased numbers.

If two arrays[1] have the same dimensions, we can perform operations that combine corresponding elements, using the typical operators +, -, *, /, and ^. If you've had linear algebra before, **note that the * operator performs an element-by-element multiplication, not matrix multiplication.** The matrix multiplication operator is **%*%** (all three symbols typed in a row).

The following statements illustrate these operations using *vec* and another five-element vector:

```
> vec = c(40,43,46,49,52)
> vec2 = c(3,-1,0,8,1)
> vec * vec2
[1] 120 -43 0 392 52
> vec2 / vec
[1] 0.07500000 -0.02325581 0.00000000 0.16326531 0.01923077
> vec ^ vec2
[1] 6.400000e+04 2.325581e-02 1.000000e+00 3.323293e+13 5.200000e+01
```

The last result includes entries in scientific notation, since the numbers were so large (49^8 is a large number indeed). When R prints a number like "5.3e+6", this means "5.3 × 10^6". (The "e" stands for "exponent" and does not indicate the constant e ≈ 2.71828.) So the fourth entry in the above vector is the value 3.323293 × 10^{13}.

Many functions in R can combine the values in a vector. For example, the function *sum*() returns the sum of all the elements in a vector:

```
> vec = c(40,43,46,49,52)
> sum(vec)
[1] 230
```

Other similar functions include *max*(), *min*(), *prod*(), *mean*(), and *sd*() (for "standard deviation"). Experiment with these to see their effects. All of these functions work on multi-dimensional arrays as well as vectors, too:

```
> mat = matrix(c(5,3,8,4,3,6),nrow=3)
> mat
      [,1] [,2]
[1,]     5    4
[2,]     3    3
[3,]     8    6
> sum(mat)
[1] 29
> max(mat)
[1] 8
> mean(mat)
[1] 4.833333
```

Quick Review Question 2

a. With one assignment statement, make *qrq2* a 2-by-4 array of all zeros.

b. With one assignment statement, make the first column of *qrq2* contain ones.

1 Or vectors (recall that a vector is just a one-dimensional array).

c. Make the first row, third element of *qrq2* be 5.

d. Display *qrq*.

e. Without changing either array, obtain the sum of *qrq* and *qrq2*.

f. Return an array where corresponding elements of *qrq* and *qrq2* are multiplied.

g. Define a new user-defined function *sqr*() to square a parameter. The function should work for numeric and vector arguments.

h. Using *sqr* from Part g, return a vector with every element of *qrq* squared.

i. Give a command to return the sum of the squares of the elements of *qrq*.

3 Combining and Transposing Arrays

Sometimes we have separate vectors of numbers that we need to combine. An example would be if we had a vector of *x*-coordinates and a vector of *y*-coordinates, and we wanted to combine them to get a matrix of ordered pairs, where each row represented one ordered pair (*x*,*y*). Suppose for an hour a scientist measures amounts (in milligrams) of residues from a chemical reaction every 12 minutes, or 0.2 hours. The following command assigns to tlst the list of times:

```
> tlst = seq(0,1,.2)
> tlst
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

The following rlst is a list of residue measurements:

```
> rlst = c(0,.05,.16,.23,.55,1)
```

We can use the *cbind*() function (for "column bind") to bind these two vectors togethers as columns of a matrix:

```
> combinedlst = cbind(tlst,rlst)
> combinedlst
     tlst rlst
[1,]  0.0 0.00
[2,]  0.2 0.05
[3,]  0.4 0.16
[4,]  0.6 0.23
[5,]  0.8 0.55
[6,]  1.0 1.00
```

We now have one time/residue ordered pair per row.

Another useful operation is to take the **transpose** of a matrix. A matrix's transpose is just the same matrix with the rows and columns interchanged: what were the rows become the columns, and vice versa. In R, the *t*() function performs the **transpose**:

```
> t(combinedlst)
```

```
        [,1] [,2] [,3] [,4] [,5] [,6]
tlst       0 0.20 0.40 0.60 0.80    1
rlst       0 0.05 0.16 0.23 0.55    1
```

Note that R preserves the names of the original rows, which is handy for interpreting data once you've combined it.

Quick Review Question 3 Write a statement to generate a list xlst of x-values, which are positive integers from 1 through 9. Using one assignment statement, have *glst* store the corresponding values of $3\sqrt{x}$. Write commands to assign to pairslst the array of ordered pairs with one ordered pair per row.

4 Reading from files

Files can store huge amounts of data and simplify input. Links to data files for various projects appear on the textbook's website. For example, Module 8.3 on "Empirical Models" uses the file *DanWoodEM.dat*, which appears in Table 1, and several much larger files, all with rectangular arrays of tab-delimited data.

```
1.309                                       2.138
1.471                                       3.421
1.49                                        3.597
1.565                                       4.34
1.611                                       4.882
1.68                                        5.66
```

Table 1 *DanWoodEM.dat*

The R function *read.table*() can read such a file of numbers. One form of the command is as follows:

```
data = read.table("Filename")
```

The file name appears in double quotes. Normally it has an extension such as *.dat* or *.txt* or *.tsv*. For files that have comma-separated (instead of tab-separated) data, the extension is normally *.csv*. This is a format that Microsoft Excel can save a spreadsheet in, for instance.

When we run the command above, we get back a **data frame** called "data". (Note that "data" was an arbitrary name — we could have called the data frame anything.) A data frame is kind of like a matrix, except that its columns are labeled:

```
> danwood = read.table("DanWoodEM.dat")
> danwood
     V1    V2
1 1.309 2.138
2 1.471 3.421
3 1.490 3.597
4 1.565 4.340
5 1.611 4.882
```

```
6 1.680 5.660
```

Printing the value of danwood shows the rows labeled 1 through 6, and the columns labeled "V1" and "V2" (for "variable 1" and "variable 2.")[2]

The data file must be in the same folder/directory where your R script file is, or else you must specify the full path name of the file.

5 Fit

In Module 8.3 on "Empirical Models," we investigate discovering functions that capture the trend of data. To do so, we employ the R function *lm*() (which stands for "linear model.") The form of a call to *lm*() is as follows:

```
model = lm("formula", data=dataframe)
```

The "formula" is a way of telling R the general form of the function we want it to fit to the data. (Again, "model" is an arbitrary name that we give to the result of calling *lm*().) Suppose we want to obtain the best-fit line — i.e., the straight line which best approximates the data. We would type:

```
> bestline = lm(V2 ~ V1, data=danwood)
```

The formula "V2 ~ V1" says "create a model of the V2 variable based on the V1 variable." The "data=danwood" part says "the data frame I want you to use is in the variable called danwood."

The coefficients of this actual line can be obtained by typing *bestline$coefficients* (this is a dollar-sign between the name of the variable and the word "coefficients"):

```
> bestline$coefficients
(Intercept)          V1
 -10.426961    9.489346
```

The y-intercept is -10.426961, and the coefficient of the *V1* variable (i.e., the slope) is 9.489346. Therefore, the best fit line has equation $y = 9.489346x - 10.426961$.

Let's plot the line, and compare it with the actual data points. First, plot the data points themselves:

```
> plot(danwood)
```

This results in the plot in Figure 1. Now, to add the best-fit line to the plot, we do the following:

2 Often a comma-separated-value (.csv) file will have the names of the variables on the first line. The R command read.csv("*Filename.csv*", header=TRUE) can be used to read such a file, and the columns of the resulting data frame will then be named properly.

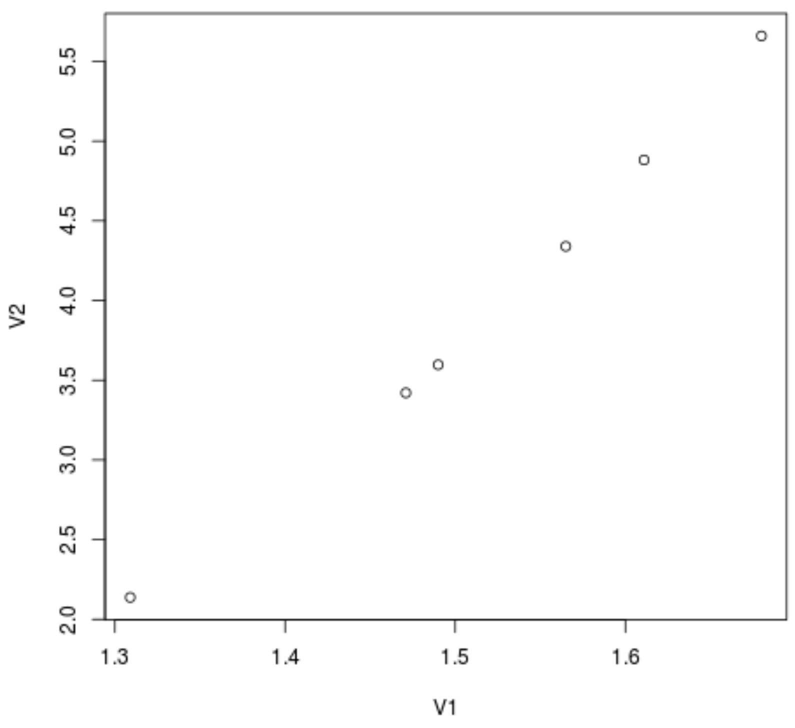

Figure 1 The points in the *DanWoodEM.dat* file.

```
> xvals = seq(1.3, 1.7, .01)
> yvals = bestline$coefficients %*% rbind(1,xvals)
> lines(xvals, yvals, col="blue")
```

This adds the best-fit line (in blue), giving the plot in Figure 2. The second line of code, above, is a bit tricky. It's using a little bit of linear algebra to accomplish the task. Here's how: recall that bestline$coefficients is a vector of the coefficients for the best-fit line. In other words, it's the vector (-10.426961, 9.489346). Now to find the *y* value for each *x* value, we need to multiply *x* by 9.489346, and then add -10.426961. If we make a matrix that has all 1's in the first row, and the *x* values in the second row, then performing a matrix multiplication (using the %*% operator) between the coefficients and this matrix will yield us the *y* value for each point.

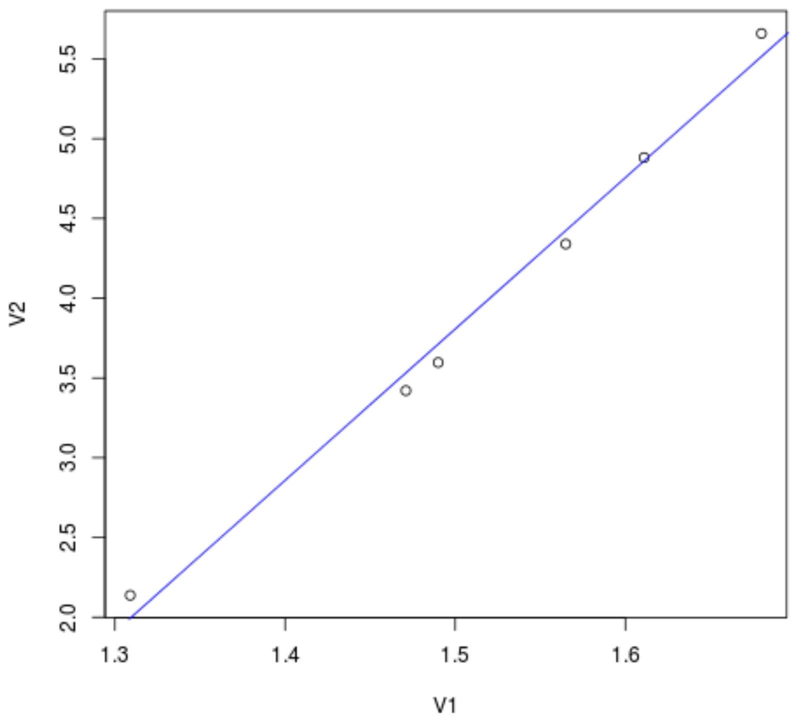

Figure 2 The best-fit line for the DanWoodEM points.

Let’s take this one step further, and find the best-fit quadratic function (i.e., the best-fit function with an x^2 term in it, instead of just an x term.) We type:

```
> bestquad = lm(V2 ~ V1 + I(V1 ^ 2), data=danwood)
```

The formula “V2 ~ I(V1 ^ 2)” says “create a model of the *V2* variable based on (a) the *V1* variable itself, and (b) the *V1* variable squared.” The *I*() function is an inconvenient necessity to prevent R from simplifying the formula before it gets a chance to make the model.

The coefficients of this quadratic can again be obtained:

```
> bestquad$coefficients
(Intercept)          V1     I(V1^ 2)
   6.301899  -13.084786    7.564851
```

The y-intercept is now -6.301899, the coefficient of the (linear) *V1* variable is -13.084786, and the *V1*-squared term is 7.564851. Therefore, the best fit quadratic to this set of points has equation $y = 7.564851x^2 - 13.084786x + 6.301899$.

To add it to the plot (in green), we type:

```
> y2vals = bestquad$coefficients %*% rbind(1,xvals,xvals^ 2)
> lines(xvals, y2vals, col="green")
```

giving the plot in Figure 3. As you can see, the best-fit line is pretty close to intersecting all the points, but the best-fit quadratic is even better.

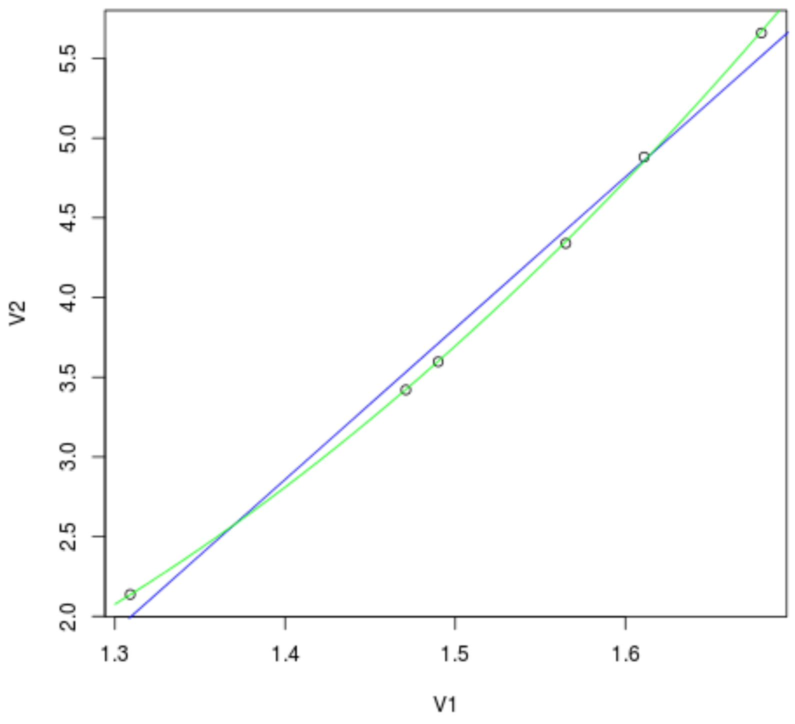

Figure 3: The best-fit line and quadratic polynomial for the DanWoodEM points.