6.1 – R Computational Toolbox Tutorial 2

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

Angela B. Shiflet and George W. Shiflet
Wofford College
© 2014 by Princeton University Press

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

1 Introduction

The prerequisite to this tutorial is "R Computational Toolbox Tutorial 1." Before proceeding, follow that tutorial in its entirety, including installing R for your system, and answering all Quick Review Questions as you go.

2 Vectors

As you recall from R Computational Toolbox Tutorial 1, all variables in R are *vector* variables, meaning they have the capability to store more than one number at a time. One way we learned to create a vector was by using the *c()* function and listing its elements:

```
> dailyStockPrice = c(55.25, 56.5, 56, 57.75, 58.25)
> dailyStockPrice
[1] 55.25 56.50 56.00 57.75 58.25
```

Again, the "[1]" indicates that the element immediately to the right is element #1 of the vector. If we had made it longer:

```
> dailyStockPrice = c(55.25, 56.5, 56, 57.75, 58.25, 59, 59, 58.75, 55, 57,
51.25, 53, 48.75, 49.25, 50.25, 51.5)
> dailyStockPrice
[1] 55.25 56.50 56.00 57.75 58.25 59.00 59.00 58.75 55.00 57.00 51.25 53.00
[13] 48.75 49.25 50.25 51.50
```

the contents would have flowed onto a second line. The "[13]" indicates that the 48.75 is the 13th element.

We can also create vectors using the *seq()* function, by specifying a starting point, ending point, and increment amount:

```
> tideLevel = seq(15, 30, 1.5)
> tideLevel
[1] 15.0 16.5 18.0 19.5 21.0 22.5 24.0 25.5 27.0 28.5 30.0
```

Also remember that the colon operator (:) can be used as a shorthand for *seq()*, if the increment amount is 1:

```
> lotNumbers = 56:70
```

```
> lotNumbers
[1] 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
```

Vectors of any length can further be combined with *c()*. Consider:

```
> incomingTideLevel = seq(15, 30, 1.5)
> outgoingTideLevel = seq(30, 15, -1.5)
> tideLevel = c(incomingTideLevel, outgoingTideLevel, incomingTideLevel)
> tideLevel
[1] 15.0 16.5 18.0 19.5 21.0 22.5 24.0 25.5 27.0 28.5 30.0 30.0 28.5 27.0 25.5
[16] 24.0 22.5 21.0 19.5 18.0 16.5 15.0 15.0 16.5 18.0 19.5 21.0 22.5 24.0 25.5
[31] 27.0 28.5 30.0
```

Another useful function is *rep()*, which **rep**eats one value some number of times:

```
> adultTicketCosts = rep(15, 4)
> adultTicketCosts
[1] 15 15 15 15
```

Vectors created with *rep()* can of course be combined like any other:

```
> adultTicketCosts = rep(15, 4)
> kidsTicketCosts = rep(10, 7)
> seniorTicketCosts = rep(14, 1)
> ticketCosts = c(adultTicketCosts, kidsTicketCosts, seniorTicketCosts)
> ticketCosts
[1] 15 15 15 15 10 10 10 10 10 10 10 14
```

Interestingly, to extract elements out of a vector, you use another vector! The vector you provide contains *the index or indices of the element(s) you want to select*. For instance, to get element number 3 out of the ticketCosts vector, you type:

```
> ticketCosts[3]
[1] 15
```

To get elements 3 through 9, you do:

```
> ticketCosts[3:9]
[1] 15 15 10 10 10 10 10
```

And to get elements 2, 12, and 4 through 7, in that order, you do:

```
> ticketCosts[c(2,12,4:7)]
[1] 15 14 15 10 10 10
```

You can see what we did: first, create a vector containing 2, 12, 4, 5, 6, 7 by combining a 2, a 12, and the sequence from 4-7. Then, put that vector in the square brackets of *ticketCosts* to extract exactly those elements in that order. Part of R's great power comes from how easy it is to manipulate and combine vectors in this way.

Remember, too, that vectors lend themselves to *for* loops, since a variable can be used as the index into the vector:

```
for (i in 3:6) {
```

```
    cat("Ticket #",i," costs $", ticketCosts[i], ".\n", sep="")
}
Ticket #3 costs $15.
Ticket #4 costs $15.
Ticket #5 costs $10.
Ticket #6 costs $10.
```

Quick Review Question 1 Create a new script / program file. Save the file under the name *RCTTutorial2.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
```

Now, use R functions to create vector variables for each of the following, copying your code into the *RCTTutorial2.R* file:

a. The numbers 47, 35, 22, and 10.

b. The numbers 1 through 12.

c. The numbers 4, 8, 12, 16, …, 84.

d. Eleven 3's followed by eleven 4's followed by twelve 5's.

e. 7, 6, 5, …, 1, 19, 2, 4, 6, 8, …, 30.

3 Matrices

A matrix is essentially a 2-dimensional vector. Each element in it has two indices rather than one, specifying an *x* and *y* coordinate.

You create matrices in R by first creating a vector with the data for the matrix, then calling the *matrix*() function and telling it with ***nrow*** how many rows the matrix will have. For instance:

```
> t = c(57, 61, 63, 64, 88, 89, 88, 86, 70, 81, 76, 76)
> temperatures = matrix(t,nrow=3)
> temperatures
     [,1] [,2] [,3] [,4]
[1,]   57   64   88   81
[2,]   61   88   86   76
[3,]   63   89   70   76
```

We've created a vector of raw data called *t*, then used it to create a matrix with 3 rows and 4 columns. Obviously, we have to tell R how many rows there are, otherwise it wouldn't know whether we wanted to create a 3 × 4 matrix (as we did), or a 4 × 3 matrix, 2 × 6 matrix, or even a 1 × 12 matrix. Equally obvious, we don't have to specify the number of columns, since if R knows how many elements there are, and how many rows, the number of columns is fixed. (If we'd rather specify the number of columns than the number of rows, we can use the *ncol* parameter to *matrix*() instead.)

To extract an element from the matrix, we supply both row and column numbers:

```
> temperatures[2,3]
[1] 86
```

```
> temperatures[3,1]
[1] 63
```

If we want *all* the elements of a particular row, we just leave out the column number:

```
> temperatures[3,]
[1] 63 89 70 76
```

If this were a table of, say, the high temperatures in three different U.S. cities over the past four days, this would give us the temperatures for the third city. Note that the answer R gave us has a "[1]" to the left of the numbers, not a "[3]". This is because once the row is extracted, it is an ordinary stand-alone vector, and has no row number information any longer.

Similarly, we can leave out the row number:

```
> temperatures[,2]
[1] 64 88 89
```

and get the high temperature for all the cities on day 2. Again, note that the result is an ordinary 1-dimensional vector, which is therefore displayed horizontally (not vertically).

You can probably guess that R lets us combine vectors of indices:

```
> temperatures[2:3,c(4,1,3)]
     [,1] [,2] [,3]
[1,]   76   61   86
[2,]   76   63   70
```

This gives us the bottom two rows, and columns 4, 1, and 3 (in that order). This particular operation may or may not be useful, but it does show the flexibility of R's matrices. Note once more that the rows and columns of the resulting answer are labeled starting from 1, since the answer is a matrix in its own right.

Incidentally, just as the *length*() function told us the length of a vector, so the *nrow*() and *ncol*() functions give us the dimensions of a matrix:

```
> nrow(temperatures)
3
> ncol(temperatures)
4
```

Many times we will use matrices to represent the contents of a virtual map, and use for loops to iterate through them. To progressively move through a matrix requires a ***nested for* loop**, which means a *for* loop inside another *for* loop. This is required because for *each* row, you need to move through each of the columns. Here's what it looks like in code:

```
for (r in 1:nrow(temperatures)) {
    for (c in 1:ncol(temperatures)) {
        cat("The high temp in city #", r, " on day #", c, " was ",
            temperatures[r,c], ".\n", sep="")
    }
}
The high temp in city #1 on day #1 was 57.
The high temp in city #1 on day #2 was 64.
The high temp in city #1 on day #3 was 88.
The high temp in city #1 on day #4 was 81.
```

```
The high temp in city #2 on day #1 was 61.
The high temp in city #2 on day #2 was 88.
The high temp in city #2 on day #3 was 86.
The high temp in city #2 on day #4 was 76.
The high temp in city #3 on day #1 was 63.
The high temp in city #3 on day #2 was 89.
The high temp in city #3 on day #3 was 70.
The high temp in city #3 on day #4 was 76.
```

Quick Review Question 2 Given the temperatures matrix, as defined above, write R expressions to return each of the following:

a. The element at row 3, column 3.

b. The element at row 2, column 1.

c. The entire 3rd row.

d. The entire 2nd column.

e. Columns 2 through 4 of rows 1 and 3.

4 Multidimensional Arrays

It may occur to you that what we did in one dimension (vectors) and two dimensions (matrices) could be extended to three, four, or even more dimensions. R can indeed let us define a structure of elements with any number of dimensions: it's called an *array*.

To create one, we use the array() function, and specify the dimensions with the dim parameter. dim is *a vector of the sizes for each dimension of the array*. For instance, suppose we wanted to model the air pressure at various places in a wind tunnel. This chamber is clearly three-dimensional, so we need a 3D array, with coordinates for length (*x* axis), width (*y* axis), and height (*z* axis). Let's say our chamber is 5 meters long, 5 meters wide, and 3 meters high, and we want to measure the air pressure at one-meter intervals. We can create an array to hold the data like this:

```
> airPressure = array(dim=c(5,5,3))
```

Perhaps at the beginning of our experiment, the air pressure is consistent across the length and width of the chamber, and it is slightly higher at lower heights. We'll say that it's 1 bar (a "bar" is a standard measure of atmospheric pressure) near the bottom of the chamber, .99 bars in the middle, and .98 bars at the top. Here's how we could initialize our three-dimensional array:

```
for (i in 1:5) {
    for (j in 1:5) {
        airPressure[i,j,1] = 1
        airPressure[i,j,2] = .99
        airPressure[i,j,3] = .98
    }
}
> airPressure
, , 1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    1    1    1    1    1
```

```
[3,]    1    1    1    1    1
[4,]    1    1    1    1    1
[5,]    1    1    1    1    1
, , 2
     [,1] [,2] [,3] [,4] [,5]
[1,] 0.99 0.99 0.99 0.99 0.99
[2,] 0.99 0.99 0.99 0.99 0.99
[3,] 0.99 0.99 0.99 0.99 0.99
[4,] 0.99 0.99 0.99 0.99 0.99
[5,] 0.99 0.99 0.99 0.99 0.99
, , 3
     [,1] [,2] [,3] [,4] [,5]
[1,] 0.98 0.98 0.98 0.98 0.98
[2,] 0.98 0.98 0.98 0.98 0.98
[3,] 0.98 0.98 0.98 0.98 0.98
[4,] 0.98 0.98 0.98 0.98 0.98
[5,] 0.98 0.98 0.98 0.98 0.98
```

The output looks a little strange at first, until you recognize the pattern. Each cell is identified by three coordinates now, and since the screen is obviously two-dimensional, R lists the contents of each height location separately. The first 5 × 5 group of cells is labeled "`, , 1`" at the top, which means "here are all the rows and columns for height value 1." The resulting group can be read just like a (2-dimensional) matrix, for that is what it is. Then the cells at heights 2 and 3 follow. Notice that inside the body of our nested for loop, we set the values for all three cells at row *i* and column *j* by varying the third index from 1 to 3.

Multidimensional arrays work just the same as matrices: we can extract a cell or group of cells by specifying indices, or vectors of indices. If we wanted columns 4 and 1 (in that order) of rows 2 through 5 for all the heights, we would write:

```
airPressure[c(4,1),2:5,]
```

leaving the third index blank since we want all the heights.

Quick Review Question 3 Given the *airPressure* array, as defined above, write R expressions to return each of the following:

a. The element at row 3, column 3 at height 3.

b. The element at row 4, column 2 at height 1.

c. The entire 3rd row for all columns and heights.

d. All rows and columns for the lowest height.

e. All heights for row 4, columns 2 through 5.

5 Plotting points

To plot points in an array, we use the *plot()* function and pass the *x* and *y* coordinates of the points. R is very flexible about how this can be done: we can pass a vector of *x* coordinates and a vector of *y* coordinates, or an $n \times 2$ matrix with *x*-*y* coordinate pairs, or use several other techniques.

The following segment assigns a vector of *x* coordinates to the variable *xCoords*, a vector of

corresponding *y* coordinates to *yCoords*, and then displays the points as small circles with the appropriate axes labels:

```
> xCoords = c(-3,2,-1,4,0)
> yCoords = c(-3,2,-1,4,0)
> plot(xCoords, yCoords)
```

This produces the plot in Figure 1.

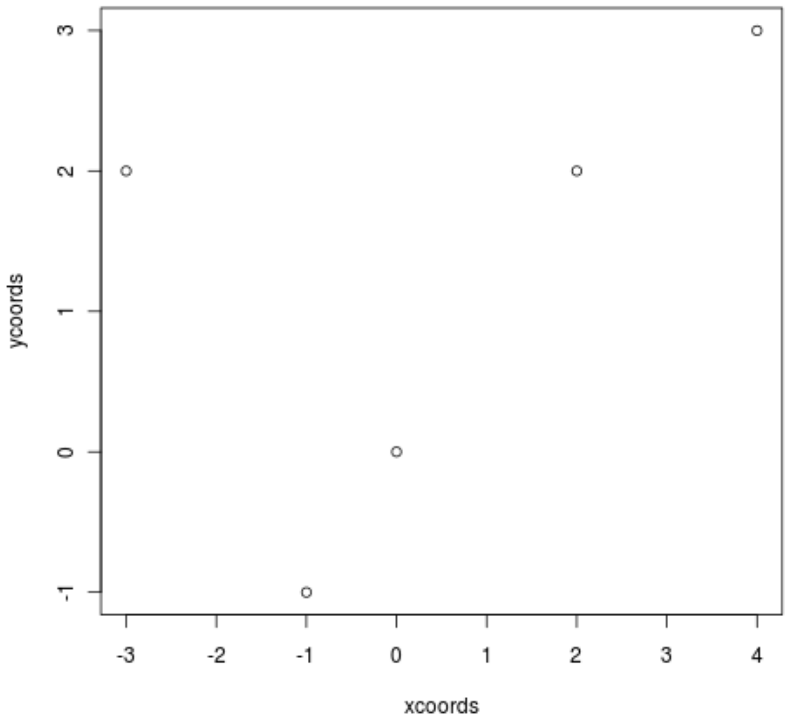

Figure 1 Plot of points as circles

R also lets us change the plot character, color, and many other graphical parameters, as we saw in the previous tutorial. For instance, we can use blue plus signs for the points, set the axes so the origin is at the center, title the plot and label the axes with the following:

```
> plot(xCoords,yCoords,pch="+",col="blue",main="A plot of points",
    xlab="the x variable",ylab="the y variable",xlim=c(-5,5),ylim=c(-5,5))
```

This produces the plot in Figure 2.

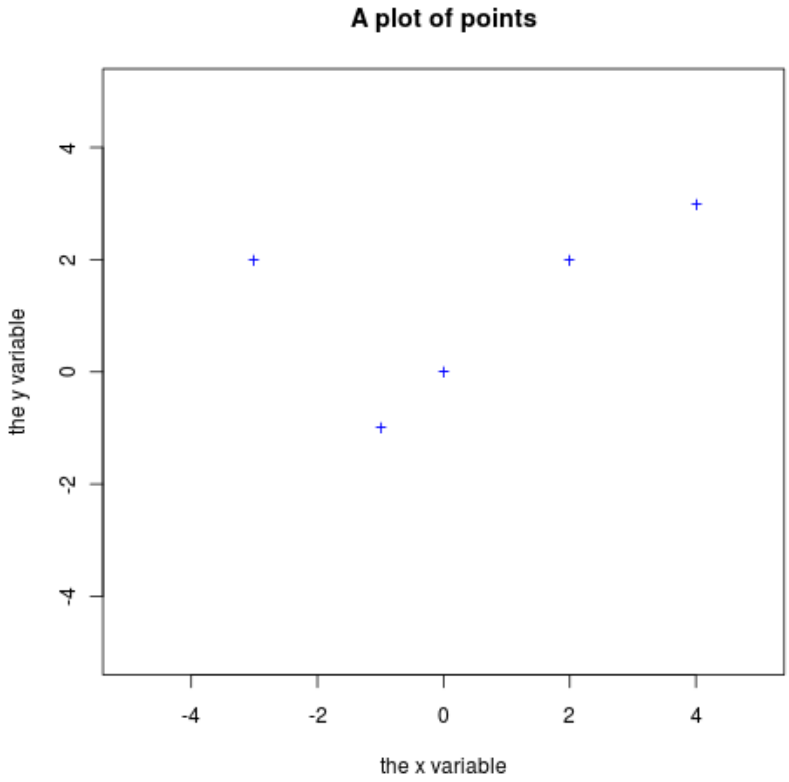

Figure 2 Plot of points as crosses

Quick Review Question 4 Create a vector called *yLst* with numbers 6, 2, 9, 9, and 12. Graph the points with *x* coordinates in *xCoords* from the above example and corresponding *y* coordinates in *yLst*.

6 Lines connecting points

Sometimes, it is helpful to visualize the path of an entity, such as an animal or a molecule, whose movement we are simulating. To generate the path, we simply give *plot*() a *type* parameter value of *l* (for "line").

For example, suppose [-1, 0, -1, 0, 1, 2, 3, 2, 1, 0] is a vector (called *yValues*) of ten *y* values, where each element is randomly one more or less than the previous element. Considering each *y* value to occur at sequential ticks of the clock, we can draw a line graph to display the trend of the *y* values over time. Just plotting *yValues* causes the first coordinates of the points to be 1, 2, …, 10 and the corresponding second coordinates to be from *yValues*. The code to display the trend of *y* over time follows:

```
> plot(yValues,type="l")
```

This produces the plot in Figure 3. We can use the *lwd* parameter to specify a line width (thickness), and *lty* parameter to choose a line type (dotted, dashed, etc.) See the help page for details, and experiment!

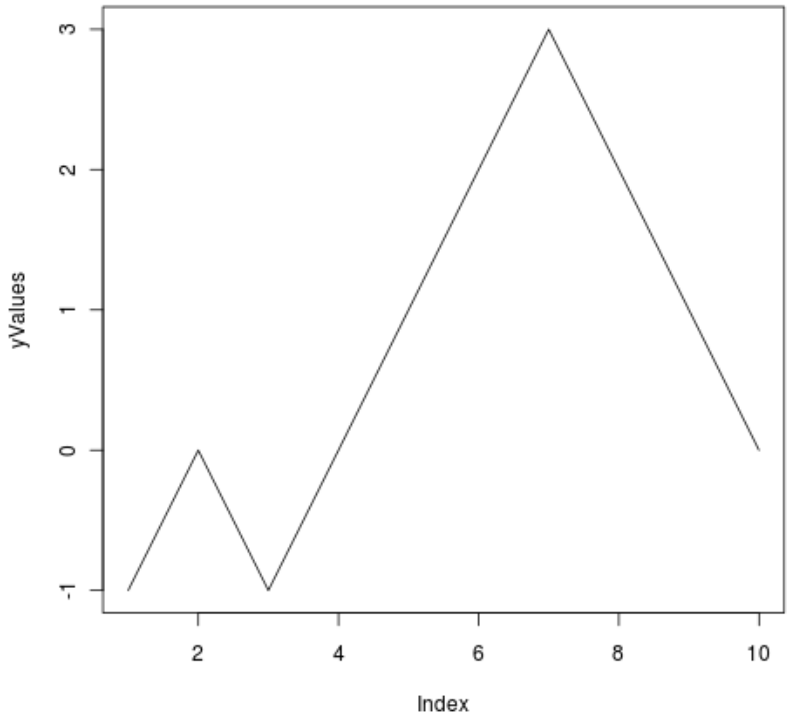

Figure 3 Points connected by line segments

Note that to *add* additional features to an already existing plot (*e.g.*, to add points to a line plot, or a line to a plot that already has lines or points), we use functions *points*() or *lines*() immediately *after* the call to *plot*(). This augments the current graph with additional features. See the help pages on these functions for details, and experiment!