An Introductory Tutorial:
Learning R for Quantitative Thinking in the Life Sciences

Scott C Merrill

September 12th, 2012

Chapter 3

Discussing last weeks bugs

Datasets that are automatically included in the base package can be found by compiling:

```
> library(MASS)

> data()
```

Next, I wanted to give you an example of a histogram using the ChickWeight data. Compile:

```
> library(lattice)
> ? histogram
> library(datasets)
> ChickWeight
> # I will discuss the following two functions this week but attach() allows us to
> #         attach and use column names as variable names
> attach(ChickWeight)

> # The names() function shows the names associated with the columns
> names(ChickWeight)

> histogram(weight)
```

This should produce a pretty histogram of the ChickWeight weights data.

Use of the functions require() and library()

```
# the functions below are basically the same
> library(animation)
> require(animation)
```

> "The other reason I use require is that it keeps me from referring to packages as libraries, a practice that drives the R-cognoscenti up the wall. The library is the directory location where the packages sit."
>
> - DWin Stackoverflow user

 # R tutorial code

```
# Setting up a matrix
A = matrix(data = 0,nrow=5,ncol=2)

# putting values into the matrix
A[,1] = c(1995:1999)
A[,2] = c(10.0,12.5,15.6,19.5,24.4)

# checking to see if it worked
A
```

Notes about functions

There are input slots (arguments) that are assumed by the function for input of different values/arguments. For example, the data to be included in the matrix is in the first input slot, the number of rows is the second input slot and the number of columns is in the third input slot. Default values for other options are in additional slots. The names of the input slots do not need to be included but not including them can lead to confusion. Compile:

```
> matrix(0,5,2)
# This is the same as A

# However, you could forget the order (like I did earlier) and type in:

> matrix(nrow=5,ncol = 2, 0)
```

This still looks like A. What happened? R started assigning values in the matrix() function to named slots (e.g., the number of rows slot, nrow), when it came to input (i.e., the 0) it assigned it to the first unfilled input slot in the function, in this case the "data" input slot.

```
> matrix(ncol = 2, 0, 5)
```

This functionality allows you to be a little sloppy in the order as long as you name each of the value slots! I encourage you to always name each of the inputs into the functions. This will be exceptionally important later when you are trying to remember the order for putting in values into a statistical function, etc. (e.g., which input slot holds the statistical formula in the glm() function, which is the generalized linear model – something I imagine many of you will use. I can't remember, but I don't have to if I write "formula =")

So, even though Hobbs didn't encourage using the "data =" in the matrix function, I think it is a good idea.

5.6 Sub-setting matrices

Very often we want to find particular rows and columns of a matrix, for example, we want to find the rows that contain specified values of a column. R has extremely powerful syntax to accomplish these operations, but be warned, it requires some effort to understand. For example, let's say we want to obtain a subset of the array A containing the data for years earlier than 1998. The syntax for doing this is as follows:

```
A[A[,1] < 1998,]
```

to which R responds:

```
[,1] [,2]
[1,] 1995 10.0
[2,] 1996 12.2
[3,] 1997 15.6
```

Well, now that is not exactly intuitive. Let's take this statement apart so that you can understand how it works. First, compile:

```
A[,1] < 1998
```

to which R responds

```
[1] TRUE TRUE TRUE FALSE FALSE
```

What is going on here? Well, you have a logical operator (<) and any time you use a logical operator, then R will return values of TRUE or FALSE. So, A[,1] specifies a vector consisting of all of the rows of A (because there is a "," in the row position of the index) in column 1. R returns a vector of TRUE and FALSE by going element by element through that vector and determining if the element is < 1998. If you don't understand this after thinking about this and discussing it with your lab mates, then call on one of the TA's or me #SCM: Since I don't have TAs you can ask me#. It is very important that you understand this.

Now let's add the outer "wrapper" that does the rest of the work we need done. I am intentionally adding some spaces to make this clearer:

```
A[ A[,1] < 1998 ,]
```

What is happening here? We have put the set of TRUE TRUE TRUE FALSE FALSE produced by:

```
A[,1] < 1998
```

within A[,]. R now looks this over finds all of the rows of A that have a value of TRUE produced by our logical comparison, that is that have a value less than 1998 in column 1.
OK we have found the rows, what does R return? R returns all of the columns of A because of the comma immediately before the trailing bracket is not followed by any specific column or

columns. (Remember, A[r,] means row r and all columns.) If we simply wanted the values of column 2 returned for the rows where years < 1998 then our syntax would be:

```
A[ A[,1] < 1998 ,2]
```

to which R would respond

```
[1] 10.0 12.2 15.6
```

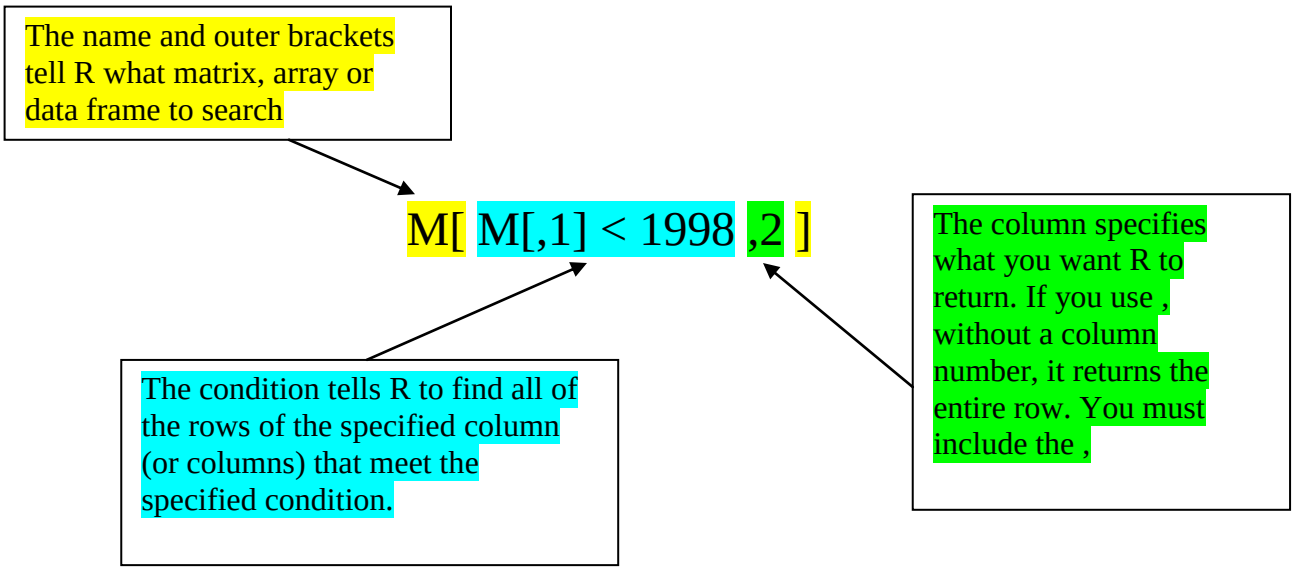

Figure 2: Illustration of command for subsetting a matrix.

So to wrap this up, consider the following diagram to gain some general understanding of subsetting matrices:

Now, the error you will make, I promise. If you compile

```
M[ M[,1] < 1998]
```

R will respond:

```
[1] 1995.0 1996.0 1997.0 10.0 12.5 15.6
```

Hmmm, what is going on here?

Some examples might reinforce the use of subsetting statements:

```
# Returns rows of A that contain 1995 or 1999 in column 1

A[A[,1] == 1995 | A[,1] == 1999,]

#Returns values in column 2 of A that are in a row containing 1995 or 1999 in column

A[A[,1] == 1995 | A[,1] == 1999,2]

#Rows of A that contain values in column 2 that are #greater than the average value

A[A[,2] > mean(A[,2]),]
```

Exercise: matrices, continued.

1. Find the all of the values of column 2 of A that are less than 19.

2. Find and output the row that contains the minimum value contained in column 2.

3. Now, to illustrate a particularly devious error, enter the following at the console:

```
A
A[1]
A[10]
mean(A[1])
mean(A[,1]).
```

Discuss what is going on here with your lab mates. What problems that can arise if you leave out the comma when you intend to index a row or column? When you enter a single subscript (with no comma) for a two dimensional matrix, how is R treating the matrix? Hint: compile the commands:

```
length(A)
length(A[1,])
length(A[,1]
```

This is particularly important to understand because it is a source of potential errors that are very hard to detect. This illustrates that flexible languages like R give you tremendous power—but with that power comes the responsibility to understand what you are doing. Don't go forward until you really understand this part of the exercise.

5.6.1 A common error when creating matrices
A common error is something like:

```
> Z =matrix(0, nrows = 5, ncol = 7)
```

in which case you will get the sort of cryptic error message:

Error in matrix(0, nrows = 5, ncol = 7) : unused argument(s) (nrows = 5)

This tells you, in R's way, that you put an s at the end of nrow, an s that shouldn't be there.

5.7 Arrays

In this course, we will use lots matrices, which are essentially 2 x 2 arrays. I say essentially because matrices have some special mathematical properties that are not shared with 2 x 2 arrays, but we needn't go into these in detail. So although matrices, lists, and data frames (more about these soon) are the data structures that we need in this course,, most of the time there will be occasions when we need "a stack of matrices", which, roughly speaking is a 3 dimensional array. A three dimensional array is analogous to a workbook—it is like a stack of sheets, each containing rows and columns. A four dimensional array is like a set of workbooks containing a stack of sheets containing rows and columns, and so on. So, a 2 x 5 x 3 dimensional array is stack of 2 dimensional arrays—it consists of 3, 2 x 5 sub-arrays. The syntax is a pretty simple. To set-up a 3 dimensional array name A3 consisting of 3 sub-arrays, each of which contains 2 rows and 5 columns each use:

> A3 = array (data = 0, dim = c(2,5,3))

Again, the indexing works, rows, columns, arrays (RCA)—the first number in the dim statement gives the number of rows in the "subarray" (i.e, 2), the second number gives the number (i.e., 5) of columns in the sub array and the third number (i.e., 3) gives the number of sub arrays.

Exercise: Three dimensional arrays I am now working on a problem that involves capture histories of mule deer that are potentially infected with Chronic Wasting Disease.

These histories take the following form. The animal can be in 1 of 4 unique states:

1) alive and susceptible
2) dead and susceptible
3) alive and infected
4) dead and infected

Our research team will observe the animal's state over a 5 year period. Data for years are contained in columns and there is a 4 x 5 subarray for each of 100 animals. For a given year, the animal's state is indexed by rows in the column for that year (row 1=susceptible alive, row 2 = infected alive, row 3 = susceptible dead, row 4 = infected dead); there is a 1 in the row if the animal is in that state and a 0 otherwise. Please do the following.

1. Create a 3 dimensional array containing all 0's to hold these data and output the sub arrays for the first 5 animals.

2. Enter initial conditions for this data structure, assuming all animals are susceptible in year 1.

3. Presume that the 5th animal is susceptible and alive in years 1 and 2 becomes infected in year 3 and dies in year five. Create the appropriate capture history for animal # 5 by filling in its sub array with 1's and 0's.

5.7.1 Functions for arrays and matrices

It is important to understand that you can use bracket notation, (i.e., A[1,]) to create a vector from a matrix. In this case, all of the function that apply to a vector (Table3) will apply to the vector that you extract from a matrix. Moreover, these functions will also apply to the matrix itself, but with results that may not be immediately intuitive. For example, consider matrix A above. What do you get if you execute the command

mean(A)?
length(A)?
min(A)?

There are times when you want means or sums of each for the rows or each of the columns of a matrix. Of course, you could do this one row or one column at a time using the bracket notation as shown above. However, there are functions that make this easier, rowSums(), rowMeans(), nrow(),colSums(), colMeans(), and ncol(). Consider the matrix Q:

$$Q = \begin{pmatrix} 8 & 81 & 12 \\ 6 & 2 & 34 \\ 13 & 12 & 1 \end{pmatrix}$$

Figure out how each of these matrix functions work by entering Q into R and using each function, e.g., rowSums(Q). The reshape package offers a much more general set of tools for summarizing data in matrices (and data frames), but that is an advanced topic, treated later in this primer.

5.7.2 Matrix algebra

R has a rich set of functions (for example, eigenvalues and eigenvectors) as well as operators that apply to matrices (for example, multiplying a matrix with a vector). These are beyond the scope of this tutorial, but you should know they exist. We will learn some of these later in the course. If you are curious now, take a look at the Introduction to R manual or the popbio package.

6 Programming in R

6.1 Iteration

6.1.1 The basics of for loops

One of the things we will do a lot in this class is to iterate an equation over time. As an example, consider the equation,

$$N_{t+1} = \lambda N_t \qquad \text{(Equation 2)}$$

which is the basic, discrete time model of exponential population growth. On the left hand side, we have the value of a state variable, N, at a future time and on the right hand side, we have a function that calculates the future value of a state variable in terms of its current value and the parameter λ. This is known as an iteration equation.

R, and all other computer languages, have several methods for executing operations iteratively, saving you the tedium of doing the same thing many times. The method we will learn is called a for loop. It will be a genuine workhorse in this class, so you should pay close attention here. For loops are best explained by example. Consider the following:

Algorithm 1 Illustration of a for loop. Compile:

```
lambda = 1.2
N = 10
for (t in 1:10){
        N = lambda * N
} #end of N loop
N
T
```

Write this code as a script and execute it. What is going on? We start by assigning an initial value to the scalar N. This tells R that N exists and has the value of 10. The code then tells R to do the following things:

1. Starting with the value of t = 1,
2. Do the operations within the { }, in this case update the value of N by making it equal to the old value of N multiplied by the (constant) value of lambda.
3. Each time you do whatever is within the { }, increase the value of t by 1
4. When the value of t = 10, stop and go to the next statement.
5. Output the values of N and t to the console.

Well, that is interesting, but not terribly useful. The loop allows us to find out the value of N after 10 intervals of time. But what we really want for most models is the value of N at each time. It does illustrate how the right hand side of an expression for a variable replaces the value on the left hand side, but you probably knew that anyway. For this to be really useful, we need vectors and arrays. But before we go on those topics, notice two programming conventions. The code within the {} is indented to make it easy to see it is part of a loop. Also, there is a comment at the end to the loop to identify it.

To make this construct more useful, study the code in Algorithm 2.

Algorithm 2 for loop including storage of state variable in a vector. Compile:

```
#value of growth rate
lambda = 1.2
#Vector for holding state variable
N = numeric(10)
# Initial value of N
N[1] = 10
#Loop to calculate N over time. Use t to index vector.
for (t in 2:10){
# store the values of N in a vector
        N[t] = lambda * N[t-1]
}
plot(N)
```

Ok what is going on here? This is pretty important, so let's look at it in detail:

1. lambda = 1.2 lambda is a parameter in our model. This model has only one parameter, other models will have many, but the general idea is exactly the same–the value of a parameter does not change with time . We assign the variable lambda the value 1.2.

2. N = numeric(10) Now we create a vector to hold the values of N in sequence.

3. N[1] = 10 If you think about equation 2, above, we can't calculate the left hand side without knowing an initial value for the right hand side. Given that starting value, each subsequent value can be calculated because the new value is based on the old one. So, we must give an initial value for N, that is, the first element in the N vector.

4. for (t in 2:10){N[t] = lambda * N[t-1] } This is where the real work is done. The for loop starts with a value of t = 2 . Why not start at 1? We start at 2 because we have specified the value of N at t = 1,so the first time through the loop, we calculate the value of N[2] = 1.2 * N[1]. The second time through the loop, we calculate N[3] = 1.2 * N[2], the third time through the loop, N[4] = 1.2 * N[3] and so on until t = 10.

5. plot(N)We now use a plot statement to display the results of the model. Usually, the plot statement has the syntax plot(x,y)where x is a vector of values for the x axis and y is a vector of values for the y axis. Again, these vectors can be named anything; x and y are simply placeholders in this example. If plot has only one argument, as in our example above, then the plot statement assumes this is a time-series creates a sequence of x values of the appropriate length starting at 1 to match with the y values we gave it. So, in this case the plot statement determines the length of N and makes an x-axis for you with values 1 – 10.