The Undergraduate Guide to R

A beginner's introduction to the R programming language

Trevor Martin

Princeton University

Table of Contents:

Section 1: Welcome!

1.1 — Who Should Use this Manual?

Hello! Congratulations on deciding to learn the R programming language. Learning R will give you a whole new set of tools with which to manipulate, analyze, compare, and view data. R is designed primarily for use in statistics, but it is useful regardless of which scientific discipline you are pursuing.

As the data sets used in all scientific disciplines get ever larger it is becoming increasingly more critical for scientists to be knowledgeable about how to use high-level programming languages such as R, which allow for easy and intuitive use. I have titled this manual “The Undergraduate Guide to R” because I want to emphasize that R is a skill that should be learned early in the modern student’s career. Of course, however, I hope that this manual is useful to everyone who is just starting to use R, undergraduate or not.

This manual is designed so that no prior knowledge of programming is required or assumed (although rudimentary knowledge of general computer skills and statistics is a must). Thus, it may seem overly simple to many and I would highly recommend that those of you who find yourselves in this situation look at Section 12: Further Resources for more advanced manuals.

The main objectives of this manual are as follows:

1.) Successfully install and run R on your computer

2.) Teach enough R that it is easy to do most common data manipulating, analyzing, comparing, and viewing tasks

3.) Provide knowledge foundation so that learning more advanced R techniques is possible

4.) Give general tips and suggestions about how to program in R

5.) Illustrate the usefulness of R

Once again, welcome to R, and I hope this manual motivates you to use R in your scientific career!

- Trevor Martin

1.2 — Don't be Afraid

Programming can seem like an intimidating and impenetrable subject. But don't worry! It is actually a very intuitive and easy process. This manual heavily uses examples so that it is as minimally theoretical as possible.

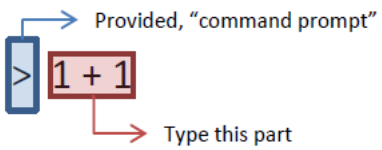
Also, if you ever get stuck or have problems your best resource is actually the people around you! From professors to your friends, chances are someone you know will be knowledgeable and glad to help you out. Additionally, one of the best ways to learn R (or learn anything for that matter) is to learn it with a friend or a group of friends. This method allows everyone to complement each other's skills and understanding. It also makes learning R more fun!

1.3 — How to Use

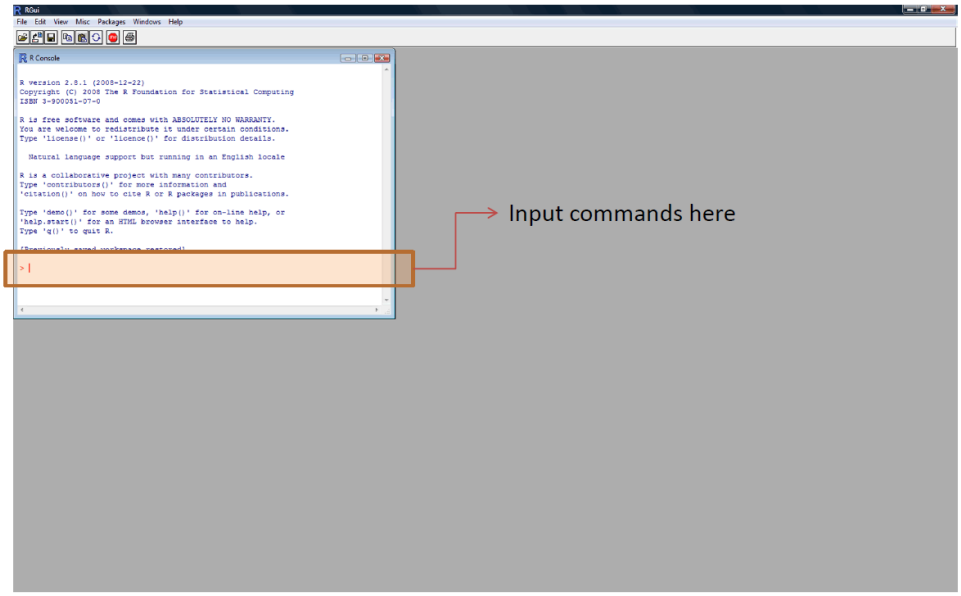
This manual is meant to be read through from beginning to end. After you make it through once, however, you may find it useful to keep it around so you can quickly reference certain points. One tip that may be especially useful is to print out this manual and read a hard copy while having R full screen on your computer (also useful while installing R). But it also works fine, if you want to save some trees (always good!), to split your screen so that half of it is R and the other half is this manual.

All that is needed to follow most of the explanations is a clean installation of R. There are, however, some parts that require a data set to be imported — the manual will clearly indicate when this task is necessary, how to import the data, and where to import the data from.

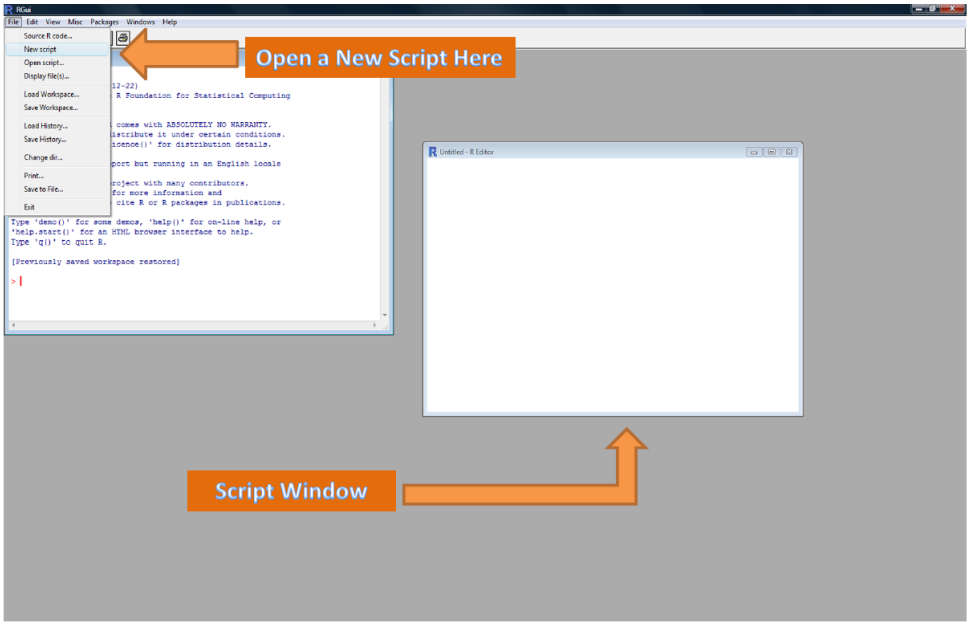
Also, please note that whenever commands to be input into the command line are shown, the bracket to the left of the command (that is provided by the command line interface) is shown (the "command prompt"). To run a typed command, just hit "enter". An example:



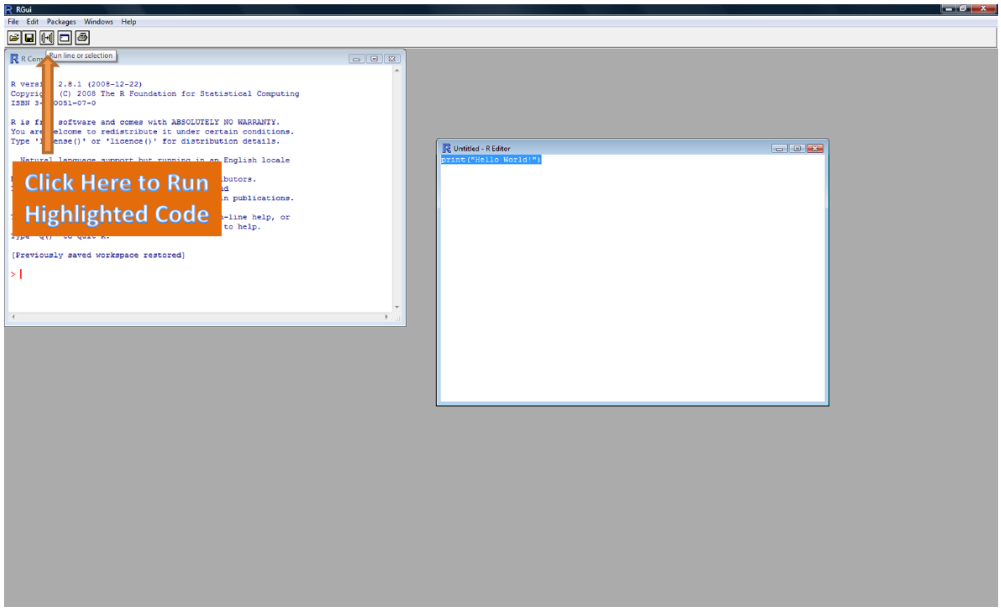
The place in the RGui where you input commands is indicated in the screenshot below (don't worry if you don't know what the RGui is, you will be walked through installing and running it later):



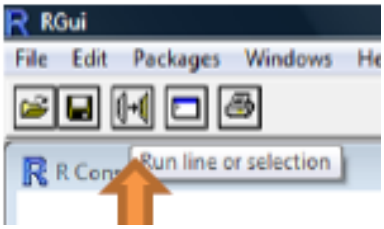
Sometimes you will be asked to type the commands in a script and then "run" these commands all at once. A "script" is a separate window where you can type your commands and run them at your own convenience. To open one of these windows follow the screenshot below:



To run the commands in your script either copy-paste the commands into the command window or highlight the commands you want to execute and then click the button indicated below:



A zoomed in view:



The above method is not the only way to execute code in a script, as you will learn later, but it is the simplest and easiest method for starting off.

Another useful tip is that while in the console, you can cycle through previously typed R commands by pressing the “up arrow” and “down arrow” keys.

Refer back to this section as you go through the manual if you can’t remember where you are supposed to input/how to input a command.

Section 2: What is R?

2.1 — History

The R programming language is an offshoot of a programming language called S. It was developed by Ross Ihaka and Robert Gentleman from the University of Auckland, New Zealand. It was primarily adopted by statisticians and is now the de facto standard for statistical computing.

For more information on the history of R try these links:

http://www.r-project.org/

http://en.wikipedia.org/wiki/R_(programming_language)

2.2 — Currently

The best part about R currently is that it is free to download and use. In addition, there is a large community of R users online who can answer your questions and who contribute what are called “packages” to R. Packages expand the functions that are available for you to use, and thus they expand your abilities, but we’ll get back to this topic later in the manual. The source code of R is maintained now by a group called the R Development Core Team.

Section 3: How to Install R

3.1 — Download

R works on many operating systems including Windows, Macintosh, and Linux. Because R is free software it is hosted on many different servers around the world ("mirrors") and can be downloaded from any of them. For faster downloads, a server closer to your physical location should be chosen. A list of all the available download mirrors is available here: http://www.r-project.org/index.html (look at the "Getting Started" section on the front page and click on download R).

For your convenience I've listed some North American mirrors here:

http://cran.cnr.Berkeley.edu	University of California, Berkeley, CA
http://cran.wustl.edu/	Washington University, St. Louis, MO
http://www.ibiblio.org/pub/languages/R/CRAN/	University of North Carolina, Chapel Hill, NC
http://lib.stat.cmu.edu/R/CRAN/	Statlib, Carnegie Mellon University, Pittsburgh, PA

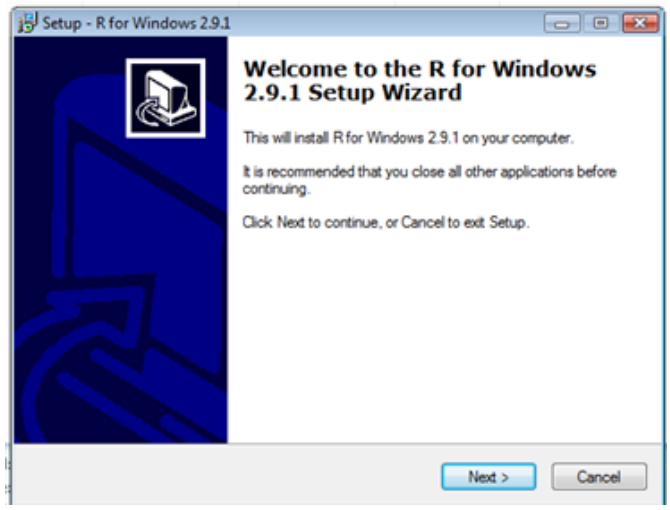
You may notice that most of the prefixes say "CRAN." CRAN stands for the Comprehensive R Archive Network and it ensures that you have the most recent version of R.

Once you have chosen a mirror, at the top of your screen should be a list of the versions of R for each operating system. Choose the R version that works on your operating system (also, you want the "base" version), then click the "download" link that appears on the screen.

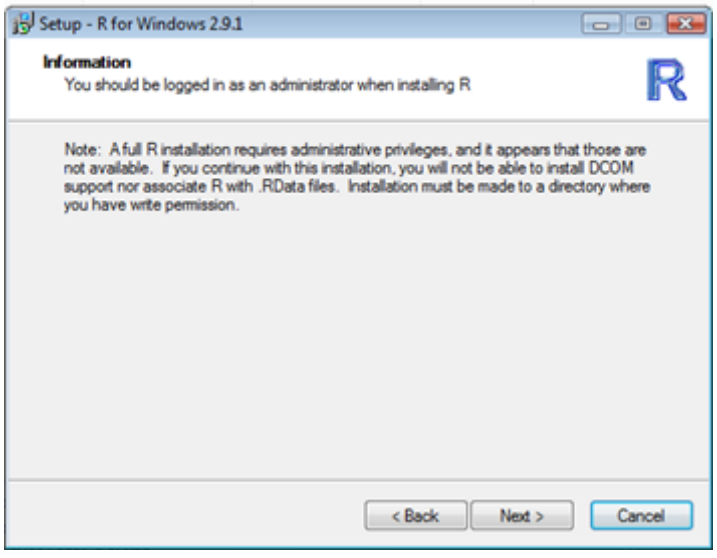
3.2 — Install

Underneath the "download" link should be another link to installation instructions. These instructions can be useful if you run into any problems during installation. Generally, however, to install R all you need

to do is double-click on the executable file and follow the instructions on screen. The default settings are fine. This screen (maybe with a different version of R listed) should be the first that appears if you are installing R on a Windows system:



One problem that is fairly common when installing is that users of Microsoft Vista will see the following error:

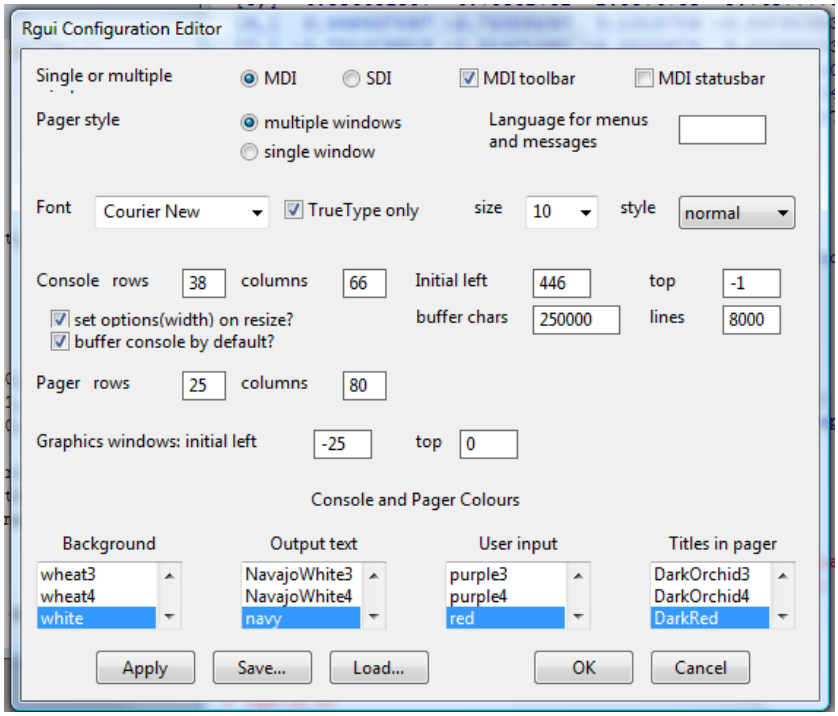


To fix this error (or any other errors with “permissions access” when installing R in Vista), simply close the installer, then right click on the icon, selecting “run as administrator”, clicking OK every time Vista prompts you with security warnings, and R should install fine.

3.3 — Configure

The package that you downloaded and installed from the CRAN mirror included both the R language itself, and a user interface called the RGui. When you click on the R icon you now have, you are taken to the RGui as it is your “editor” (more on editors in Section 11: R Editors). “RGui” is short for “R Graphical User Interface” and it is an intuitive and easy way to interact with the R language.

We will configure some aspects of the RGui later (such as the “working directory”), but for now the only option we need to set is whether we want the RGui to run in SDI or MDI mode. What the SDI mode means is that when you open script windows or graphics windows in R, they open as separate windows, while the MDI mode makes all windows open nested inside a single larger window. Don’t worry if this distinction doesn’t seem very clear, the best mode in my opinion is MDI. To make sure your RGui is set in this mode go to Edit > GUI Preferences. From there, you can set this preference, along with many others:



Section 4: The Basics

OK, now that we've covered some background information and made sure everything is installed correctly, let's get down to the nitty-gritty of actually programming. This section will walk you through the nuts and bolts of using R. Remember to look back to Section 1.3 — How to Use if you can't remember how to input commands.

4.1 — Algebra

R has all the functions of a normal calculator; here we'll quickly look at how to use these functions. Try entering the following:

```
> 1 + 1
```

The output should look like this:

```
[1]  2
```

Don't worry about what the [1] in front of the answer means for now, that will be covered later, in Section 4.3 — Matrices. From now on the input and output will be put right after each other, like so:

```
> 1 - 1

[1]  0
```

The operators available are + (addition), - (subtraction), / (division), * (multiplication), and ^ (raise to a power).

Most of the time, however, you will not want to just have the calculation appear in the console window, you will instead want to save the calculation as a "variable". To save a calculation as a variable, simply think up a variable name, and then use an equal sign to assign the calculation's output to that variable. For example:

```
> variable1 = 1 + 1
```

Now, there will not be an output of "2" after you hit enter because instead of outputting the result, R has saved the result to "variable1".

Now, try typing the following:

> variable1

[1] 2

You get the output that you saw before, as the calculation was the exact same, it was just saved to the variable.

Note: Some other manuals you read may assign calculations to variables as follows:

> variable1 <- 1 + 1

The "<-" operator is, for our purposes, exactly the same as the "=" operator in R except for one exception, which is covered in Section 9: Functions. I prefer to use the "=" operator simply because it is more intuitive for me and is a keystroke less than "<-".

Ok, the last main point to cover about algebra in R is the order of operations. The order is exactly the same as the one you likely learned in high school; that is, multiplication and division are done first in order from left to right and then addition and subtraction are done, again from left to right. For example:

> 1 + 2 / 3 - 2 * 6.5

[1] -11.33333

The order of operations can be controlled by using parenthesis. Operations are performed from the innermost parenthesis outwards, so in the following example:

> 1 * (2 / (1 + 1))

[1] 1

The calculation (1 + 1) is performed first. Then, (2 / 2) is calculated, followed by 1 * 1. Whereas if this operation were written without parenthesis the output would be as follows:

> 1 * 2 / 1 + 1

[1] 3

With 1 * 2 calculated initially, followed by 2 / 1 and finally 2 + 1.

4.2 — Vectors

The next main task that you should get comfortable with in R is creating and manipulating vectors (the word “vector” is always used in the mathematical sense in this section, the data type vector will be covered later). A vector is a string of numbers; sometimes vectors are sequential numbers, other times vectors are random numbers — there is no restriction on the type or amount of numbers that a vector can contain. The simplest way to create a vector is the following command:

```
> vector1 = 1:9

> vector1

[1]  1 2 3 4 5 6 7 8 9
```

The colon “:” operator creates a sequence of numbers from the left hand value to the right hand value, iterating in increments of 1. For example:

```
> 1.2:9

[1]  1.2 2.2 3.2 4.2 5.2 6.2 7.2 8.2
```

R started at 1.2 and increased this number by 1 until it reached the largest number less than or equal to 9.

Another way to make vectors is with numbers you specifically select:

```
> vector2 = c(1, 3, 2, -8.1)

> vector2

[1]  1.0 3.0 2.0 –8.1
```

You can input whatever numbers you want (separated by commas) into the c() function to create a new vector with those numbers. Functions will be covered in more detail in Section 9: Functions.

When adding vectors, R adds the first element of one vector to the first element of the other, the second element of one vector to the second element of the other, etc. For example:

```
> vector3 = 1:9

> vector3
```

[1] 1 2 3 4 5 6 7 8 9

> vector4 = 9:1

> vector4

[1] 9 8 7 6 5 4 3 2 1

> vector3 + vector4

[1] 10 10 10 10 10 10 10 10 10

The first element of this new vector was 9 + 1, the second 8 + 2, etc.

Now, you may be wondering what happens if the vectors are not the same size, well, let's try:

> vector5 = 1:3

> vector5

[1] 1 2 3

> vector3

[1] 1 2 3 4 5 6 7 8 9

> vector3 + vector5

[1] 2 4 6 5 7 9 8 10 12

What happens, as you can see, is that R loops the smaller vector as many times as necessary to add something to each element of the larger vector. In this case the first element is 1 + 1, the second element is 2 + 2, the third element is 3 + 3, but then the smaller vector loops back to its first element to add to the fourth element of the larger vector, giving the calculation 1 + 4 for the fourth element.

When the larger vector is not a multiple of the smaller vector the process is the same, except R will give a warning message informing you that the larger vector is not an integer multiple, so not all elements of the smaller vector were added to the larger vector an equal number of times. Try this situation for yourself.

Multiplying, dividing, exponentiating, and subtracting vectors works in the same, element-by-element fashion.

4.3 — Matrices

Matrices are the bread and butter of R. Your data will likely be in a matrix form and R is optimized for calculations involving matrices. Before reading the rest of this section, however, if you have not taken linear algebra or do not have experience with matrix multiplication I would highly recommend watching at least the first half of Gilbert Strang's MIT lectures on the subject. Those lectures can be found here:

http://ocw.mit.edu/OcwWeb/Mathematics/18-06Spring-2005/VideoLectures/index.htm

A rudimentary knowledge of linear algebra is essential for using R and its powerful matrix operations.

To begin, let's go over how to make a new matrix in R. The following is a simple way to create a new matrix with numbers you input:

```
> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3)

     [ ,1 ] [ ,2 ] [ ,3 ]
[ 1, ]    1      4      7
[ 2, ]    2      5      8
[ 3, ]    3      6      9
```

As you can see, the matrix() function takes the data you input and the number of rows you input (nrow) and makes a matrix by filling down each column from the left to the right. You can also specify the number of columns (ncol):

```
> matrix(1:8, ncol = 2)

     [ ,1 ] [ ,2 ]
[ 1, ]    1      5
[ 2, ]    2      6
[ 3, ]    3      7
[ 4, ]    4      8
```

You can also take a mathematical vector and turn it into a matrix:

```
> matrix(vector3, nrow = 3)

     [ ,1 ] [ ,2 ] [ ,3 ]
[ 1, ]    1      4      7
[ 2, ]    2      5      8
[ 3, ]    3      6      9
```

The numbers above and to the left side of the matrix give the dimensions of the matrix. So you can see how matrices are indexed from left to right and from top to bottom. In the next section you will see how to use the indexes to manipulate matrices.

If you try to create a matrix that has more elements than data provided, then R will give you a warning message and loop the data set until the matrix is filled:

```
> matrix(vector3, nrow = 2)

     [ ,1 ] [ ,2 ] [ ,3 ] [ ,4 ] [ ,5 ]
[ 1, ]    1      3      5      7      9
[ 2, ]    2      4      6      8      1
```

The bottom right corner of the matrix was created by looping the mathematical vector.

By now you can probably also see why answers to calculations and vectors are preceded by [1] as shown:

```
> 1 + 1

[1] 2

> vector3

[1] 1 2 3 4 5 6 7 8 9
```

This [1] is because a vector is equivalent to one row (or one column) of a matrix. The answer to 1 + 1 is a vector with one element (essentially a number) and vector3 is a vector with nine elements. As you may notice a vector with 9 elements is roughly equivalent to a matrix with 1 row and 9 columns (or vice versa):

```
> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 9)

     [ ,1 ] [ ,2 ] [ ,3 ] [ ,4 ] [ ,5 ] [ ,6 ] [ ,7 ] [ ,8 ] [ ,9 ]
[ 1, ]      1      2      3      4      5      6      7      8      9
```

Adding, subtracting, multiplying, and dividing matrices works simply by performing the action on each element of the matrix:

```
> matrix1 = matrix(vector3, nrow = 3)

> matrix1 + 2

     [ ,1 ] [ ,2 ] [ ,3 ]
[ 1, ]      3      6      9
[ 2, ]      4      7     10
[ 3, ]      5      8     11
```

As you can see, 2 was added to each element of the matrix.

Adding two matrices works only when the matrices are of the same dimensions (same number of columns and rows) and the matrices are added element by element in the same way as vectors:

```
> matrix1 + matrix1

     [ ,1 ] [ ,2 ] [ ,3 ]
[ 1, ]      2      8     14
[ 2, ]      4     10     16
[ 3, ]      6     12     18
```

In order to perform matrix multiplication in the linear algebra sense (not just multiply each element by a constant) the operation should be put in “%”. For example:

```
> matrix1%*%matrix1

     [ ,1 ] [ ,2 ] [ ,3 ]
[ 1, ]     30     66    102
[ 2, ]     36     81    126
[ 3, ]     42     96    150
```

The above matrix is the product of row and column operations (the product of linear algebra matrix multiplication). In contrast, element-by-element multiplication can also be performed:

```
> matrix1 * matrix1

     [ ,1 ] [ ,2 ] [ ,3 ]
[ 1, ]    1     16     49
[ 2, ]    4     25     64
[ 3, ]    9     36     81
```

As you can see, the matrix above was created by simply multiplying the individual elements in each matrix.

It is also very easy to take the transpose of a matrix (switch the rows and columns):

```
> t(matrix1)

     [ ,1 ] [ ,2 ] [ ,3 ]
[ 1, ]    1     2     3
[ 2, ]    4     5     6
[ 3, ]    7     8     9
```

Well, that covers most of the basic matrix functions. The next section discusses how to manipulate matrices in more detail.

4.4 — Manipulation

Manipulation of matrices and vectors is one of the most common tasks you will undertake in R. Thankfully, it is an easy and (eventually) intuitive process to arrange your data. First, some points regarding the indexing of matrices. Take for example the following matrix:

```
> matrix1

     [ ,1 ] [ ,2 ] [ ,3 ]
[ 1, ]    1     4     7
[ 2, ]    2     5     8
[ 3, ]    3     6     9
```

As you may have noticed, there are commas next to the numbers indicating the number of the rows and columns. For example: [1,]. What this comma means is “take all the elements”; if it comes after the number it means all the elements in that row, if before the number all the elements in that column. This convention is because of the common (row, column) system of listing row number then column number when identifying an element of a matrix.

To use the indexing of the matrix to access its elements type the matrix name followed by the element(s) you want in brackets. For example:

```
> matrix1[1, 3]

[1]  7

> matrix1[ 2, ]

[1]  2 5 8
```

As you can see, the result is returned as a mathematical vector.

Alternatively, data can be changed, not just manipulated. For example, to remove the second column from the matrix one would use the following command:

```
> matrix1[,-2]

        [ ,1 ]   [ ,2 ]

[ 1, ]      1       7

[ 2, ]      2       8

[ 3, ]      3       9
```

R takes the matrix, removes the second column, and shifts everything over.

If you want to change the actual values of the data just access the part of the matrix you want to change and use the “=“ operator as follows:

```
> matrix1[1, 1] = 15

> matrix1

        [ ,1 ]   [ ,2 ]   [ ,3 ]

[ 1, ]     15      4       7
```

```
[ 2, ]    2    5    8
[ 3, ]    3    6    9
> matrix1[ ,2 ] = 1
> matrix1
          [ ,1 ]  [ ,2 ]  [ ,3 ]
[ 1, ]    15      1       7
[ 2, ]    2       1       8
[ 3, ]    3       1       9
> matrix1[ ,2:3 ] = 2
> matrix1
          [ ,1 ]  [ ,2 ]  [ ,3 ]
[ 1, ]    15      2       2
[ 2, ]    2       2       2
[ 3, ]    3       2       2
> matrix1[ ,2:3 ] = 4:9
> matrix1
          [ ,1 ]  [ ,2 ]  [ ,3 ]
[ 1, ]    15      4       7
[ 2, ]    2       5       8
[ 3, ]    3       6       9
```

You can also assign matrix values to be vectors you've created:

```
> matrix1[ ,1 ] = vector5
> matrix1
          [ ,1 ]  [ ,2 ]  [ ,3 ]
[ 1, ]    1       4       7
[ 2, ]    2       5       8
[ 3, ]    3       6       9
```

So far you have been manipulating your data based on location, but it is also possible (and very useful) to be able to manipulate data based on the value of the data itself. For example, the statement:

```
> matrix1[matrix1 > 5]

[1]  6 7 8 9
```

The statement above uses what is called a *Boolean operator*. Basically a Boolean operator uses true/false statements to decide which data to select. In this case, we asked R to give us all the values of matrix1 which had values greater than 5, and it returned these values as a mathematical vector. We could also have asked for values less than, equal to, greater than or equal to, less than or equal to, or not equal to using, "<", "==", ">=", "<=", and "!=", respectively.

A simpler use of Boolean operators can be seen by using the command below:

```
> matrix1 > 5

        [ ,1 ]     [ ,2 ]     [ ,3 ]

[ 1, ]  FALSE  FALSE  TRUE

[ 2, ]  FALSE  FALSE  TRUE

[ 3, ]  FALSE    TRUE  TRUE
```

As you can see, R has returned the matrix as a matrix of true and false values instead of numbers, where the true or false tells you whether that element satisfied the Boolean operation "> 5". When you asked for "matrix1[matrix1 > 5]" R internally created this matrix of true/false values and then returned to you only the values that were true.

Now we can combine many of the techniques we've learned into a statement such as the following:

```
> matrix1[matrix1 >= 8] = 3

        [ ,1 ]   [ ,2 ]   [ ,3 ]

[ 1, ]       1       4       7

[ 2, ]       2       5       3

[ 3, ]       3       6       3
```

In the statement above, you changed all the values in matrix1 that were greater than or equal to 8 to 3.

Later on, we will go over some other more advanced ways of selecting parts of your matrices, but you already have a powerful set of tools for manipulating your data.

4.5 — Loops/Statements

Although R is built around operations on matrices, it still contains a powerful set of tools called loops and statements that you will find in many other programming languages. Also, the following section has commands that need to be input into a script window; please see Section 1.3 — How to Use if you do not remember how to open and use a script window.

The first and perhaps most important loop is called the “for-loop”. It should be used sparingly (more on this part later), but is appropriate for many tasks. Below is an example illustrating how a for-loop works (please input the following commands into the script window and then run them):

```
for( i in 1:3 ) {
print(i)
print(i + 5)
}
```

The output should be as follows:

```
[1]  1
[1]  6
[1]  2
[1]  7
[1]  3
[1]  8
```

As you can see, the for-loop works by selecting a value for i (in this case the values of i it iterates through are contained in the vector 1:3), then goes through the code from top to bottom. First, i = 1, and the loop calls print(i). The function print(), true to its name, outputs whatever is within its parentheses. Thus, the number 1 is output first. Then, the function print() is called again, this time as print(i + 5), outputting the number 6, or 1 + 5. Then, the process starts over for i = 2 and finally for i = 3, after which the loop finishes. Only commands within the { } are run.

A great simple use of for-loops that you will probably take advantage of is using for-loops to fill a matrix with specific values. For example:

```
matrix3 = matrix1 + 5

print(matrix3)

for( i in 1:3) {

matrix3[ i, ] = matrix1[ i, ]

}

print(matrix3)
```

The output should look as follows:

```
        [ ,1 ]  [ ,2 ]  [ ,3 ]
[ 1, ]      6       9      12
[ 2, ]      7      10      13
[ 3, ]      8      11      14
        [ ,1 ]  [ ,2 ]  [ ,3 ]
[ 1, ]      1       4       7
[ 2, ]      2       5       8
[ 3, ]      3       6       9
```

The above set of commands creates a new matrix, then replaces all the values inside of it with the values from a separate matrix. This very simple example can be extrapolated from with powerful results.

Next, let's look at a commonly used statement — the "if-statement". The following commands illustrate a usage:

```
print(matrix1)
for(i in 1:3) {
        if(matrix1[i,1] >= 2) {
        matrix1[i,1] = 0
        }
}
print(matrix1)
```

The output should look like the following:

```
        [ ,1 ]  [ ,2 ]  [ ,3 ]
[ 1, ]     1       4       7
[ 2, ]     2       5       8
[ 3, ]     3       6       9
        [ ,1 ]  [ ,2 ]  [ ,3 ]
[ 1, ]     1       4       7
[ 2, ]     0       5       8
[ 3, ]     0       6       9
```

The commands above combine the for-loop and the if-statements. These R commands tell the computer to look at the first element of each row in matrix1 and check if it is greater than or equal to 2, then, if that element is greater than or equal to 2, that value is set to 0.

If you ever have trouble figuring out what loops are doing (and nested loops can get tricky!) the best way to figure out what is going on is often to choose values and walk through a possible iteration of the loops. A value of i = 1 is usually sufficient to test for yourself what is going on in a loop.

Another loop that can be useful is the “while-loop”. An example of the while-loop in action is provided below:

```
vector6 = 9:7
print(vector6)
```

```
while(sum(vector6) >= 6) {
vector6 = vector6 - 1
print(vector6)
}
```

The output of this loop should be as follows:

```
[1]  9 8 7
[1]  7 6 5
[1]  6 5 4
[1]  5 4 3
[1]  4 3 2
[1]  3 2 1
[1]  2 1 0
```

This time, the commands create a new vector, and then decrease the numbers in this vector until they sum to a number that is less than 6. Note the introduction of the new function sum(), which calculates the addition of all the elements of vector6.

The last main statement that we will cover in this section is the "ifelse-statement". This statement is similar to the if-statement but allows you to control what happens for both the true and false Boolean values created. To illustrate:

```
> ifelse(matrix1 > 5, 1, 0)
      [ ,1 ]  [ ,2 ]  [ ,3 ]
[ 1, ]      0       0       1
[ 2, ]      0       0       1
[ 3, ]      0       1       1
```

The ifelse() function is checking which values of matrix1 are larger than 5, and then it is assigning those values that are TRUE to 1, and the values that are FALSE to 0. The important part about this statement is to remember that the parameter order is what you want to set true values to, followed by what you want to set false values to. Other more complex functions similar to the elseif-statement will be introduced in Section 9: Functions.

Section 5: Data Types

So far we've covered a lot regarding how to manipulate and change data. Now we're going to take a moment and look back at how R is actually storing and outputting the results. R has a myriad of data types for handling all sorts of data and it is important to have a general idea of how they work, and how to switch between the available types ('classes'). The 'class' of a variable tells R how that variable should be handled. There are many subtle distinctions regarding R data types, and here we only scratch the surface for the simple purpose of dividing the most common data types into broad, but very useful, categories.

5.1 — Types

We'll start off with the "simplest" data type in R, the 'numeric' class. To begin, try saving as a variable the simple calculation we performed near the beginning of this manual:

```
> calc1 = 1 + 1

> calc1

[1]  2
```

Now, we can use a function called class() to ask R to tell us what the data type of our variable is:

```
> class(calc1)

[1]  "numeric"
```

When you simply enter numbers into R, they are defaulted into saving as class 'numeric'. The next example is perhaps a little less intuitive, but will be explained:

```
> numbers1 = c(1, 2, 3)

> numbers1

[1]  1 2 3

> class(numbers1)

[1]  "numeric"
```

You are probably thinking that the above numbers1 variable should be a

vector (the data type), as it is a string of numbers, not one number as you would perhaps normally think the ‘numeric’ class would denote. The trick lies, however, in how the c() function that we used to put the numbers together works. The c() function is literally the ‘concatenate’ function and it takes the numbers you enter and puts them together in a string. Each of the numbers you entered into the c() function had class numeric, and a concatenation of numeric numbers is numeric as well.

A data type closely related to the ‘numeric’ class is the ‘integer’ class. Try the following:

> numbers2 = 1:3

> numbers2

[1] 1 2 3

> class(numbers2)

[1] “integer”

This example may also, at first, seem confusing. Shouldn’t the class of numbers2 be numeric as well, since it is a string of numeric numbers (1, 2, 3)? The answer is that the numeric class is a more general class than the integer class. R recognized that the string you were inputting consisted solely of integers and thus chose the more specific data type to store the values. The reasoning behind this choice is that the more specific a data type is, the less memory the computer takes to store the values, thus increasing the efficiency of your program as space is not wasted. R is simply trying to help you out!

You may also look back at the numbers1 example and ask why R did not store that variable as an integer when it clearly also consists solely of integers. The answer is that you entered the data for numbers1 using the c() function, and you had the ability to enter anything you wanted — R does not check the inputs into the c() function and thus simply always assumes the ‘numeric’ class so that you have the freedom to enter any numbers you want.

Another example that may help your understanding is the following:

> numbers3 = 1.1:3

> numbers3

[1] 1.1 2.1

> class(numbers3)

[1] “numeric”

In the case above, numbers3 is ‘numeric’, not ‘integer’, as R recognized that the numbers were not integers, and thus chose the broader class, ‘numeric’.

The ‘character’ class is another broad class that is used for any text input into R. For example:

> char1 = “Hello World!”

> char1

[1] “Hello World!”

> class(char1)

[1] “character”

Similarly to how when class ‘numeric’ are concatenated using the c() function they are still ‘numeric’, concatenated ‘characters’ are still class ‘character’:

> char2 = c(“red”, “green”, “blue”)

> char2

[1] “red” “green” “blue”

> class(char2)

[1] “character”

Also note that if numbers and text are combined, everything is converted to class ‘character’:

> combined1 = c(1, “test”, -3.3)

> combined1

[1] “1” “blue” “-3.3”

> class(combined1)

[1] “character”

Above, the numbers are no longer numbers in the sense that you can no longer add and subtract them — they are simply characters in the same way that the word ‘test’ is a string of characters.

Yet another simple class is the ‘logical’ class. The ‘logical’ class contains True/False values (Boolean values):

```
> logical1 = c(T, F, F)

> logical1

[1]  TRUE   FALSE   FALSE

> class(logical1)

[1]  “logical”
```

The next data type we will cover is the ‘vector’ class (please note that the data type ’vector’ is defined differently than the mathematical vector, that we have been referring to previously in this manual). It may seem to you at this point that the vector data type is redundant, as a string of numbers is class ‘numeric’, a string of characters is class ‘character’, and a string of T/F values is class ‘logical’. The advantage of the vector, however, is that it can contain either ‘numeric’ or ‘character’ or ‘logical’. The caveat is that it can only contain one type at a time. For example:

```
> vector1 = vector(mode=“logical”, 3)

> vector1

[1]  FALSE   FALSE   FALSE
```

As you can see, to create (“initialize”) a new vector simply define the type of class (the mode) that you want the vector to contain and how many elements. You can then fill these place-holders with your actual values:

```
> vector2 = vector(mode=“numeric”, 4)

> vector2

[1]  0 0 0 0

> vector2[3] = 5

> vector2
```

[1] 0 0 5 0

The ‘matrix’ data type functions exactly as you would expect it to, and you have seen how to declare this class previously:

> matrix1 = matrix(1:9, nrow = 3)

A ‘matrix’ is in effect a two-dimensional ‘vector’. Matrices can contain any of the data types that a vector can, and still contains the restriction that only one type of data may be in the matrix.

> matrix2 = matrix(c(T, F, T, T, F, T), ncol = 3)

> matrix2

[,1] [,2] [,3]

[1,] TRUE TRUE FALSE

[2,] FALSE TRUE TRUE

> class(matrix2)

[1] “matrix”

When you take a row (or column) out of a matrix, your new object defaults to the class of the individual components of the matrix:

> matrix2[1,]

[1] TRUE TRUE FALSE

> class(matrix2[1,])

[1] “logical”

A slightly more complicated version of the ‘matrix’ data type is the ‘array’ data type. The ‘array’ data type can still only have one type of data type inside of it, but the set of data types it can store is larger. In addition to the data types a ‘matrix’ can store, an ‘array’ can store ‘matrices’ as its elements and two other data types called ‘data frames’ and ‘lists’ which we will cover momentarily. Arrays can also be multi-dimensional, which you control by creating a vector of values for the number of elements in each dimension and setting “dim” equal to this vector, as illustrated below:

> array1 = array(1:16, dim = c(4, 2, 2))

```
> array1
, , 1
         [ ,1 ]  [ ,2 ]
[ 1, ]       1       5
[ 2, ]       2       6
[ 3, ]       3       7
[ 4, ]       4       8
, , 2
         [ ,1 ]  [ ,2 ]
[ 1, ]       9      13
[ 2, ]      10      14
[ 3, ]      11      15
[ 4, ]      12      16
```

The , , 1 and , , 2 tell you the “depth” you are at. The array above has dimensions 4 x 2 x 2 (dim interprets the vector you enter as increasing dimension from left to right, so the first number is the first dimension elements, the second number the second dimension, etc.).

An even more complex data type is the ‘data frame’. Data frames are similar to arrays in that they can contain any other data type regardless of its complexity, but they have certain advantages over arrays. Data frames allow one to associate each row and each column with a name of your choosing and allow each column of the data frame to have a different data type if you so desire. An example:

```
> myvalues1 = c(348, -343, 937, 394, 124)
> myvalues2 = c(T, F, T, T, F)
> names = c(“Trial 1”, “Trial 2”, “Trial 3”, “Trial 4”, “Trial 5”)
> dataframe1 = data.frame(myvalues1, myvalues2, row.names = names)
> dataframe1
```

	myvalues1	myvalues2
Trial 1	348	TRUE
Trial 2	-343	FALSE
Trial 3	937	TRUE
Trial 4	394	TRUE
Trial 5	124	FALSE

When you use the data.frame() command simply provide your data and they will be set as columns, then set the names you want for your rows by modifying the variable “row.names”. You can change the column names by using the function names(). For example:

> names(dataframe1)[1] = “Velocity”

> dataframe1

	Velocity	myvalues2
Trial 1	348	TRUE
Trial 2	-343	FALSE
Trial 3	937	TRUE
Trial 4	394	TRUE
Trial 5	124	FALSE

Data frames are great ways of visualizing your data sets.

The final data type that we are going to go over is the ‘list’. Lists are similar to vectors, but they can contain any number of any data type. For example:

> list1 = list(“thing1”, 1.5, FALSE)

> list1

[[1]]

[1] “thing 1”

[[2]]

[1] 1.5

[[3]]

[1] FALSE

The list command has created a section for each item you input and now you can access those sections by typing (paying attention to the double brackets):

> list1[[1]]

[1] “thing 1”

Alternatively, you can create names for different sections and input values into those sections:

> list2 = list(category1 = c(“thing1”, “thing2”), category2 = 1.5, category3 + = c(FALSE, TRUE))

Note: The plus sign above means a continuation of the previous command. If you accidentally hit enter in the console before your command is complete, R will prompt you with a + to finish your command and hit enter again.

> list2

$category1

[1] “thing1” “thing2”

$category2

[1] 1.5

$category3

[1] FALSE TRUE

Now you can access the different parts of list2 by typing the following:

> list2$category3

[1] FALSE TRUE

You can also modify the parts of your list using similar commands:

> list2$category2 = 5

> list2$category2

[1] 5

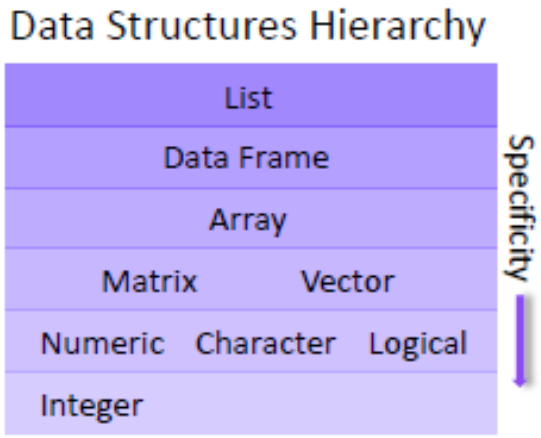
> list2$category1[2] = “thing3”

> list2$category1

[1] “thing1” “thing3”

To see all the elements in a list, simply type str(listnamehere). Lists are another great way to keep your data organized and have as few variable names as possible.

The following is a chart that should help summarize and organize the data structures:



5.2 — Converting/Using

Now that you have a good idea what data types are available and what they are used for, let’s discuss a bit about how to convert between data types.

If you ever want to coerce an object to be a certain type of class simply use a variant of the following functions:

> matrix4 = as.matrix(c(1, 2, 3))

> matrix4

[,1]

[1,] 1

[2,] 2

[3,] 3

> class(matrix4)

[1] “matrix”

> vector7 = as.numeric(matrix4)

> vector7

[1] 1 2 3

> class(vector7)

[1] “numeric”

Similarly one can use the functions as.vector, as.logical, as.character, etc. to “cast” current objects to new data types.

Learning to convert between data types in R is important because some functions will only work with certain data type inputs and because data organization is almost as critical as analyzing the data itself. In the next section, when we look at how to read in data, it will also become apparent that many times you will want to change the data type that R automatically creates for your read in data. For example, R may read your data into a data frame, but you want to do matrix multiplication calculations, so you have to recognize that your data is in a data frame and then convert the object to a matrix in order to perform your calculations.

Section 6: Reading in Data

Much of the data you use in R will be data you input from other sources where you have been storing your experimental data, such as excel spreadsheets. Learning to correctly read in this data is important and will allow you to use R on data from any of the most common types of files. You will hopefully find this process quick and intuitive.

6.1 — Types of Input

There are two main types of data that you will read into R: spreadsheet type data and data that is already in R object form, but saved to its own file. Spreadsheet type data is read in as "delineated" files. Delineated files mean that there is some standard separator to tell R how to separate cells in a table. For example, a comma delineated file uses commas between cells:

thing1, thing2, thing3, thing4, thing5

The above would be read in as:

[1] "thing1" "thing2" "thing3" "thing4" "thing5"

If you want to import the data you have saved in an Excel spreadsheet into R, the most painless method is to save the excel spreadsheet as a comma-separated file ".csv" (it's under 'Save As > Other Formats' in Excel 2007, or you can simply use Save As and then type the .csv extension) and then simply follow the instructions below for importing a comma-delineated file into R.

There are, however, several main methods for importing spreadsheet format data. The method that you use depends on the format that you save your data into before importing the data into R. As mentioned above, I prefer using the ".csv" format, but other formats such as delineation by semicolon and tab are possible and you may encounter them and have to use one of the other methods presented below.

6.2 — How to Read in Data

Let's start by reading in a comma-delineated ('.csv') file called "proteinconc.csv".

To perform the following example you must download the .csv file we are going to import and put it into the same folder as your R working directory. A working directory is the place where R looks without further instruction for any files you ask it to retrieve. In order to check what your R working directory is type:

> getwd()

[1] "C:/Users/myusername/Documents/My R Stuff"

If the directory it returns is not the directory you would like to use, then simply use the following function to set your working directory:

> setwd(C:/Users/myusername/Documents/My Other R Stuff)

Keep in mind that the setwd() command cannot create folders for you, it can only access them. Thus, you should create your "My R Stuff" folder first and then set your working directory to this folder.

OK, now that we've covered how to set up your working directory, you can find the "proteinconc.csv" file here:

http://sites.google.com/site/undergraduateguidetor/manual-files

Make sure to download the file and save it in your R working directory. Then, type the following:

> proteinconc = read.csv("proteinconc.csv", header=TRUE)

> class(proteinconc)

[1] "data.frame"

Above, we use the function read.csv() to import the data into R (R creates a new 'data frame' to hold the data), first supplying the name of the file to be imported inside "" marks and then telling the function that the first row contains the column names by setting header=TRUE. The only problem is that now, when we print the data frame by simply typing:

> proteinconc

You will notice that we have the row number as the row name and the

protein names as the first column. If we do not care to have a well organized data set, we could leave this data set as is and just keep in mind that the first column lists the name of the proteins — but being the R aficionados that we are, we are of course not going to leave it at that! To reset the first column in your data frame as the row names, type the following:

> row.names(proteinconc) = proteinconc[,1]

> proteinconc

Now, on your screen should be the modified data frame. You can see that now the row names have changed from numbers to the protein names, but the first column of the data frame still contains the protein names as well! To fix this problem, we type the following:

> proteinconc = proteinconc[,-1]

As you learned back in Section 4.4 — Manipulation, this command removes the first column of the data frame. Now, type:

> proteinconc

Congratulations! Your data frame should now look well organized and ready for analysis.

If we wanted to read in a file that was delineated by something other than commas (in this case semicolons, ";") we could use the following similar command:

> read.delim("filename.xxx", header=TRUE, sep=";", dec=".")

The above command reads in some file named "filename.xxx" where the extension is whatever extension the file has, then R takes the first row as the column names, as header=TRUE, and differentiates cells of the data matrix by looking for semicolons, as sep=";". The last part, dec="." tells R that when there are numbers, the decimal point is indicated by a period '.'. The commands sep and dec can be used in read.csv as well, except sep obviously defaults to "," and dec defaults to ".". Some common separators you may encounter are "," (comma), ";" (semicolon), "/t" (tab), "." (period), and " " (whitespace).

Now that we've read in spreadsheet type data, let's try reading in data that is an R object, but saved. The file extension for a saved R object(s) is '.rdata'. A single '.rdata' file can contain a multitude of R objects of different classes. The '.rdata' file you are about to read in can be found

here:

http://sites.google.com/site/undergraduateguidetor/manual-files

To load in this data set make sure it is saved in your working directory, and type the following:

> load("proteinconc.rdata")

Now, the two objects that were contained within proteinconc.rdata are another copy of the protein concentration data called:

> proteinconc2

And a surprise called:

> yourmessage

Section 7: Plotting Data

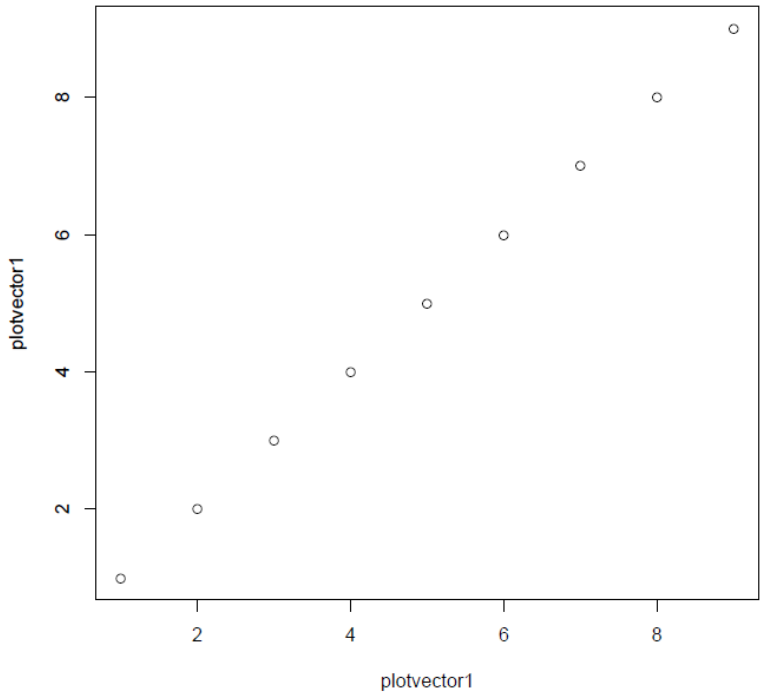
One of R's most powerful aspects is its suite of graphics devices. Visualizing data is often one of the best ways to tease out patterns and interpret what you have collected. Towards this goal, R has a variety of different plotting tools that you can use and customize to your needs.

7.1 — Dot Plots

One of the most simple (and most useful) plotting tools in R is the dot plot. The dot plot is called through the plot() command. The plot() command takes in two vectors of equal length and plots them against each other (vectors in the mathematical sense, R can plot input of all the most common data types) . The first vector provided goes on the x-axis and the second vector provided goes on the y-axis. An example:

```
> plotvector1 = 1:9

> plot(plotvector1, plotvector1)
```

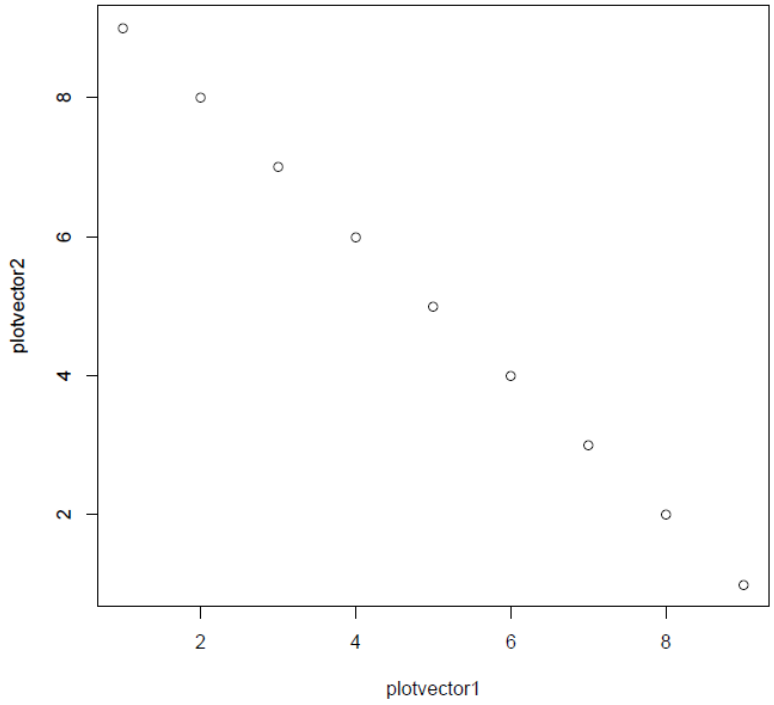
The following screen should appear in a new window:



The top of the screen should tell you that the window is “R Graphics: Device 2 (ACTIVE)”. What this statement means is that this window is a plotting window and that it is where any new plots will be displayed. If you plot one graph as we have done, and then plot another graph, by typing the code below:

```
> plotvector2 = 9:1

> plot(plotvector1,plotvector2)
```

You will see that your old plot is gone and replaced by the new plot:



We will not always want to replace our old plots, so it can be really helpful to open up a new window for our next plot. To do so, we use the following command:

```
> x11()
```

Now you should see a new empty window that opens next to your old window that says “R Graphics: Device 3 (ACTIVE)” at the top. In addition, your old window should now say “R Graphics: Device 2 (inactive)”. Any new plots you make will be put in your new “Device 3” window, as it is active. Let’s try this action, through the following commands:

> plot(plotvector1, plotvector1)

The first plot we created should now be visible again in your “Device 3” window. You can keep typing the command x11() to get more and more windows, so that effectively you can have as many plots saved at once as you want.

Now, let’s see how to modify the actual graphical parameters of your plot. First, close any existing plot windows. We are going to first use the par() function to set some general parameters:

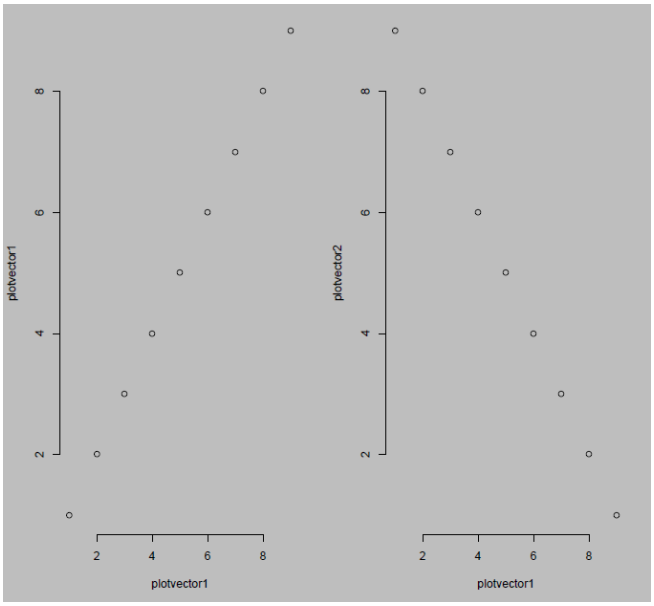
> par(mfrow=c(1,2), bg = “grey”, bty=“n”, cex=.75)

A new graphics window should pop-up. What the par() command does is set the parameters this new graphics window uses. In this case, we have used the command “mfrow” to allow the graphics window to have two plots instead of one, with the format being (rows, columns), just as the c() function is normally used. Then, we set the background color to “grey” using the “bg” command, removed the box that normally surrounds our plot using “bty”, and made the text size 3/4 of what it normally is through the “cex” command. There are many more par() commands available and these can be accessed through R’s built in Help function (more on using this help database in Section 10: Tips for Writing Good R Code). Now, let’s again plot our two graphs:

> plot(plotvector1, plotvector1)

> plot(plotvector1, plotvector2)

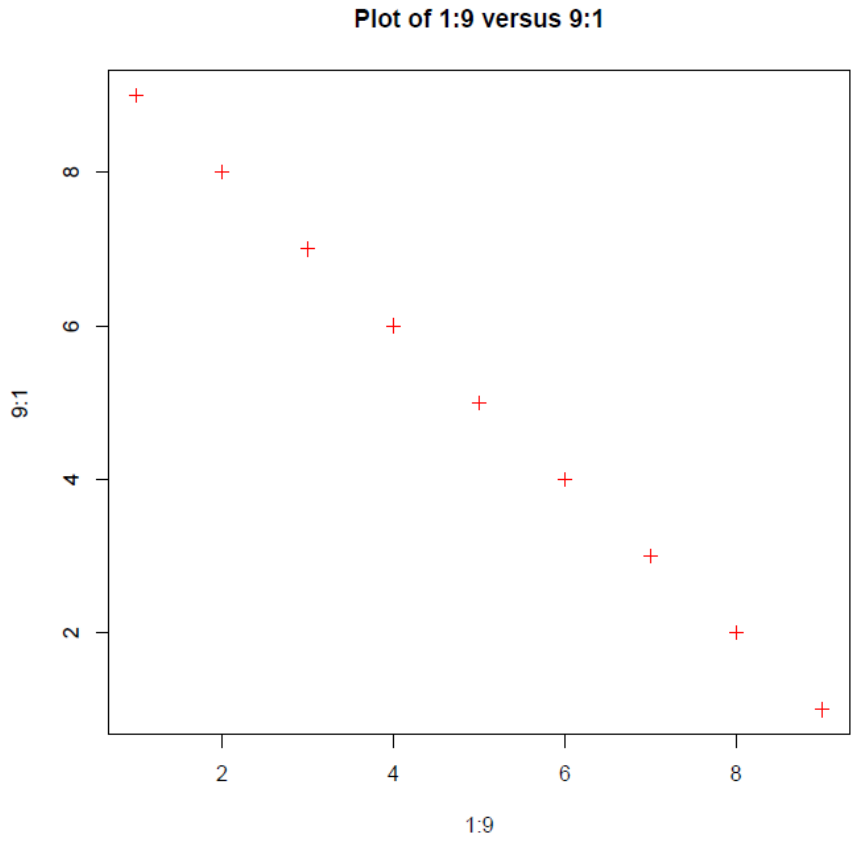
You should see the following image in your graphics window:



Now, we're going to see how to adjust parameters of each individual plot. Close the last graphics window and type the following command:

> plot(plotvector1, plotvector2, xlab="1:9", ylab="9:1", main="Plot of 1:9 versus 9:1", col="red", pch=3)

The following image should appear in a new graphics window:



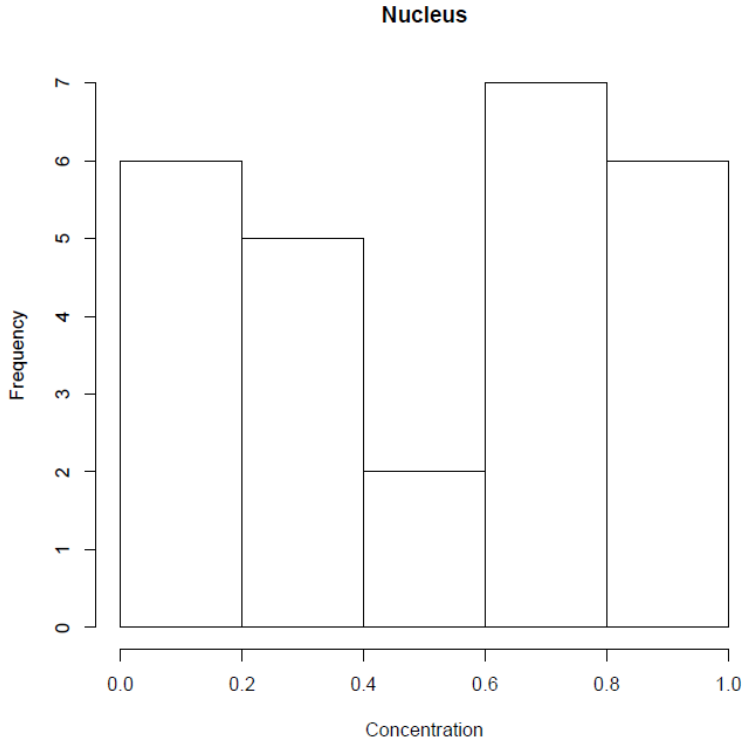
Above, we have again plotted plotvector1 and plotvector2, but we have also labeled all the axes using "xlab" for the x-axis, "ylab" for the y-axis and "main" for the title of the plot. We additionally set the color of the plot points to be red using "col" and the shape of the plot points to plus signs using "pch". The various symbols available to plot and the other functions available to use with plot() will be learned through experience, but you have enough basic knowledge now to create some very presentable graphs! Play around with the parameters and you will quickly discover the richness of the R graphics window.

7.2 — Histograms

Histograms are another powerful way to visualize a distribution of data. Histograms in R are created using the hist() command. Let's try visualizing the data frame of protein concentrations we imported in Section 6.2 — How to Read in Data:

```
> hist(proteinconc[,1], main = colnames(proteinconc)[1], xlab =
+ "Concentration")
```

The following image should appear in a new graphics window:



As you can see, we have histogrammed all the concentrations in the first column of our protein concentration data frame, which was the data on concentrations in the nucleus. Most of the parameters available in plot() are available in hist() as well, notice that we used "main" and the function colnames(), which returns a vector of all the column names in the data frame, to set the title of the plot, and "xlab" to set the name of the x-axis.

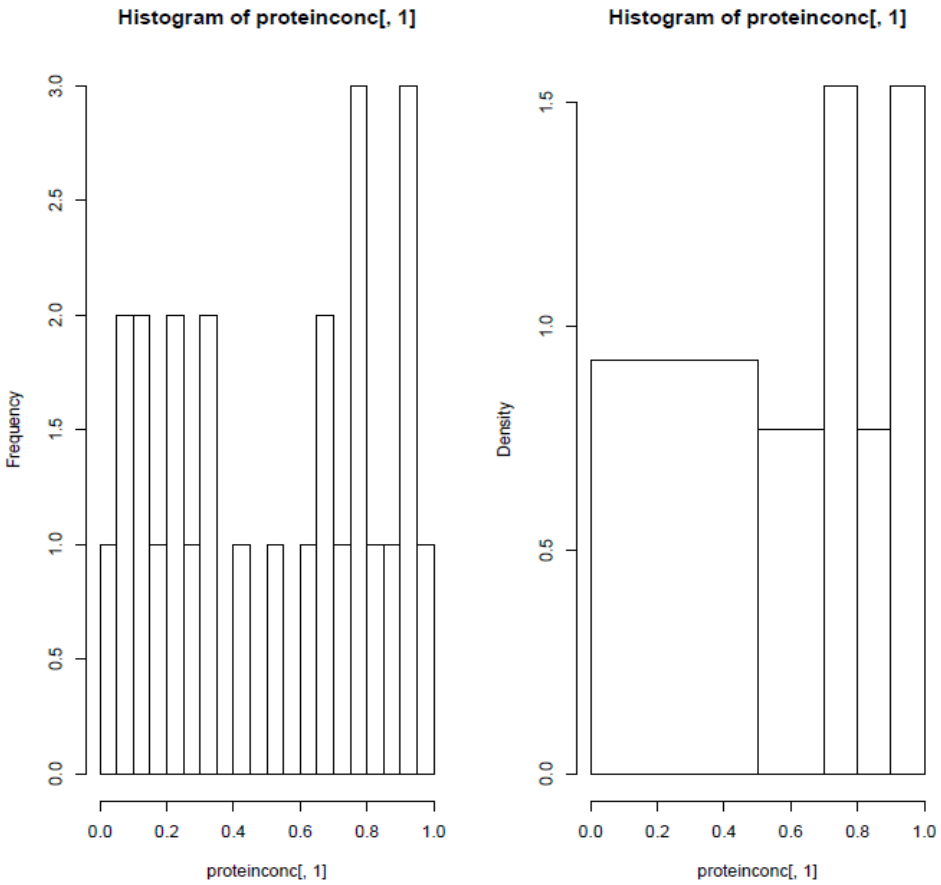
The function hist() also, however, has several unique parameters that are important to know how to use. Type the following commands:

> par(mfrow=c(1,2), cex=.75)

> hist(proteinconc[,1], breaks = 50)

> hist(proteinconc[,1], breaks = c(0, .5, .7, .8, .9, 1)

You should see the following appear in a new graphics window:



Above, we used the parameter “breaks” in two ways. In the first, we used breaks to set the number of bins that the values were put into (in this case, only 17 of which actually contained values), in the second, we used “breaks” to set the limits of the bins we wanted.

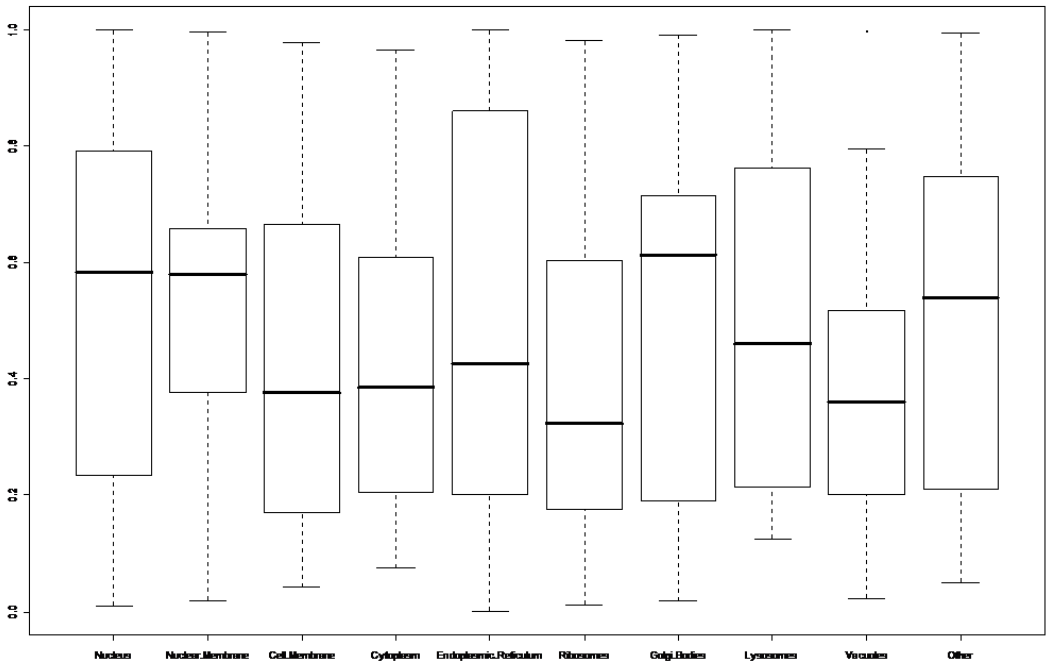
The final parameter of R histograms that we will cover is the ability to set frequency or density as the y-axis of your plot. Above, you see that hist() usually defaults to plotting frequency, but if you set bin limits that make bins of unequal length it changes to density (as it mathematically should). The function hist() can also be forced to plot density by adding the parameter “freq=FALSE”.

7.3 — Box Plots

Box plots allow you to easily see how your data is distributed and graphically identifies the quantiles of your data in addition to pointing out any outliers. To create a boxplot in R, we use the appropriately titled command boxplot(). Let's try using this command to visualize the distribution of the protein concentration data we imported in section 6.2 — How to Read in Data:

```
> par(cex = .6)

> boxplot(proteinconc)
```

The image below should appear in a new graphics window:



The black line inside each box is the median for each category. The whiskers extend out for 1.5 times IQR (Inter-Quartile Range) by default, but this parameter can be changed by setting "range" to whatever value you want to multiply the IQR by. Also, outliers are plotted as points, as can be seen by looking at the top of the 'Vacuoles' category.

The boxplot() command can also take many of the same par() commands as plot() can, and you can see that the text size was reset in the commands above using par().

7.4 — Additions

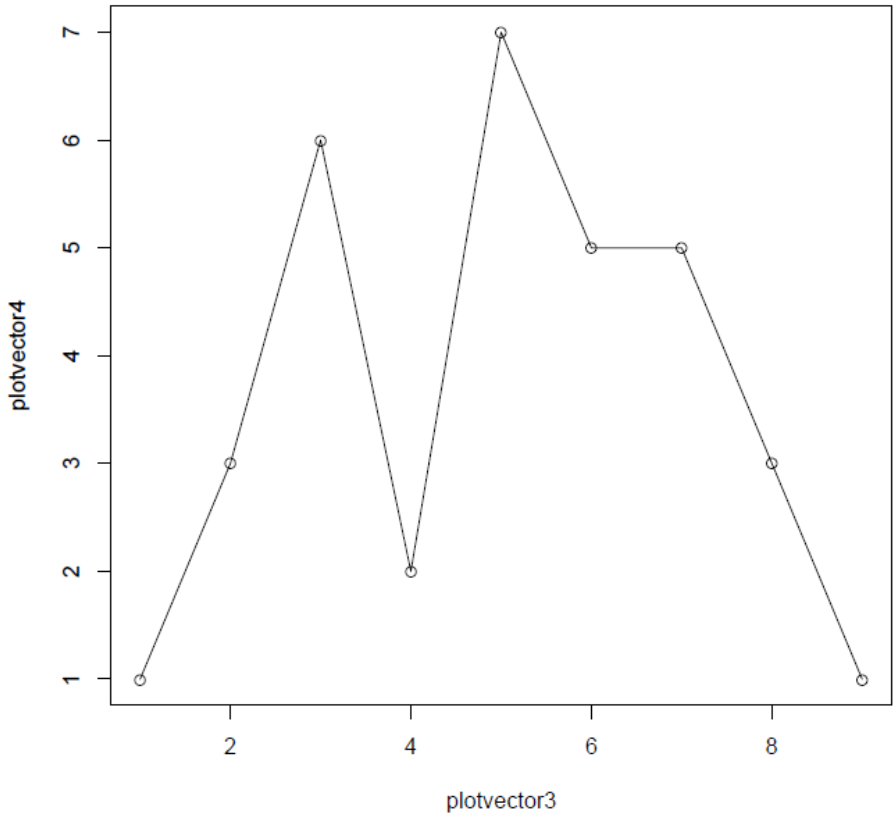
It is often useful to add various new graphical elements to the plots you have created. The most common of these additions would be lines and points. To use the functions below, first create a new plot:

```
> plotvector3 = 1:9

> plotvector4 = c(1, 3, 6, 2, 7, 5, 5, 3, 1)

> plot(plotvector3, plotvector4)
```

The function lines() is very similar to the plot() function, except instead of just plotting a point or symbol at the coordinates you have input, it connects them with line segments. An example (this command should be entered after the plot is created above):

```
> lines(plotvector3, plotvector4)
```

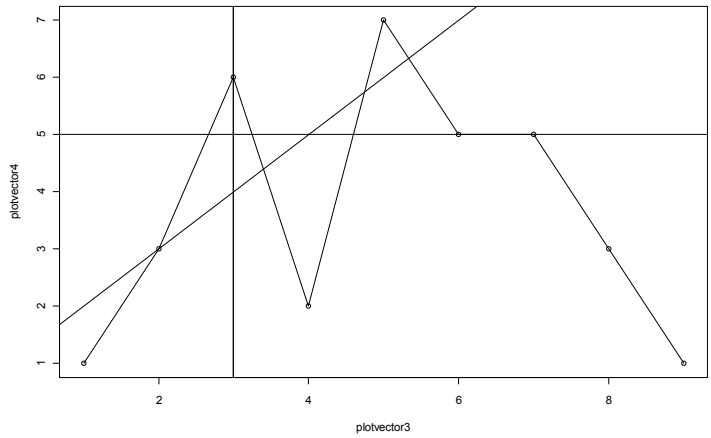
Your points should have been connected by lines:



If we simply want to add a horizontal or vertical line along a certain segment of the plot, we can use the function abline(). Try the following command:

> abline(coef = c(1, 1), v = 3, h = 5)

The graphic below should appear in your current plotting window:

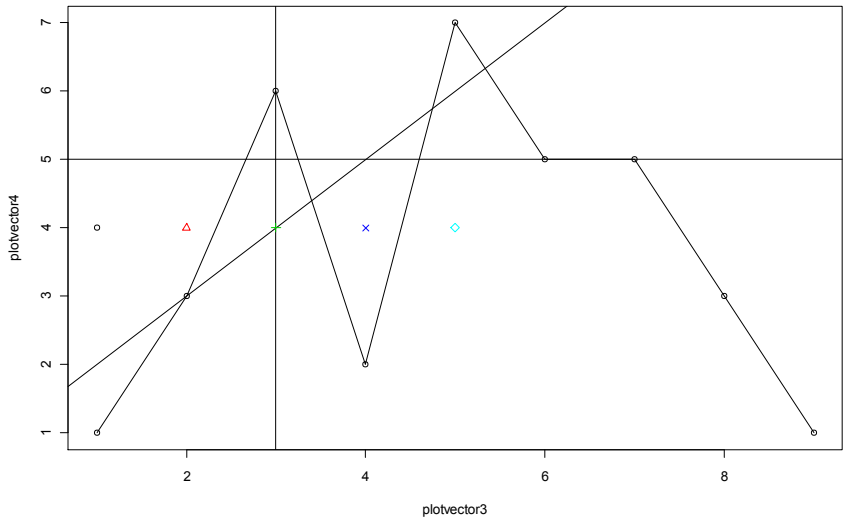


The parameter “coef” allowed us to specify the (intercept, slope) pair for a line, the parameter “v” specified the x-axis coordinate of a vertical line, and the parameter “h” specified the y-axis coordinate of a horizontal line.

We can also add additional points to our plot using the points() function. Try this command (still keeping our old plot window open):

> points(1:5, c(4, 4, 4, 4, 4), pch=1:5, col=1:5)

Now you should have the following image in your graphics window:

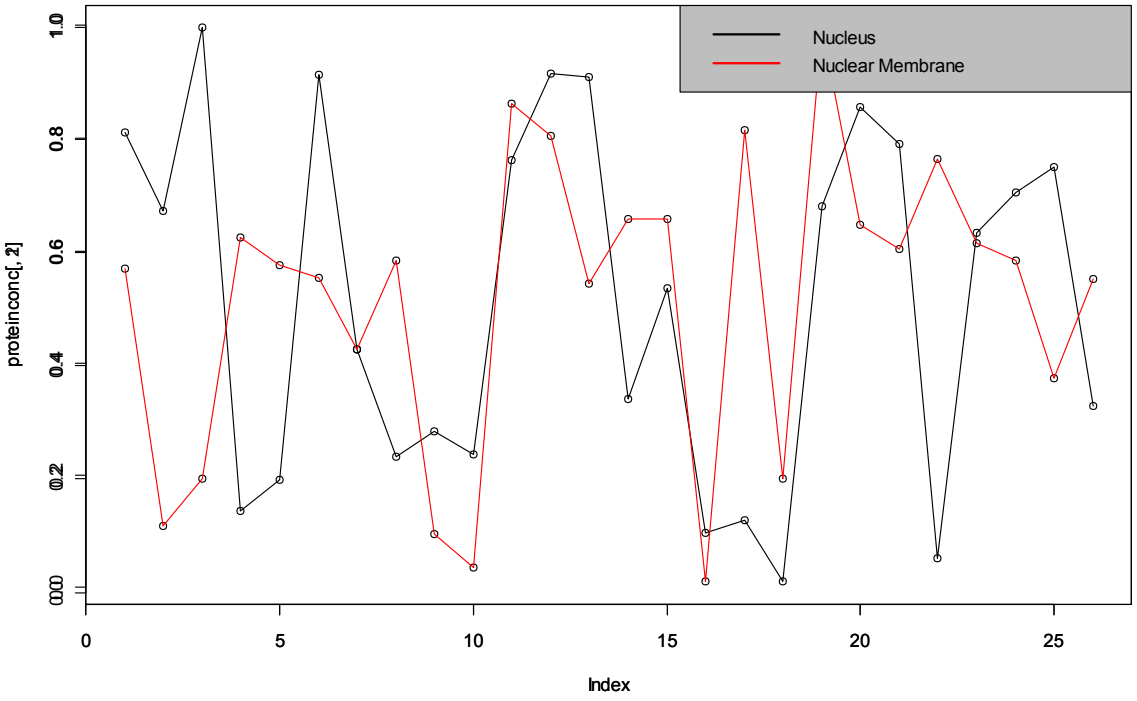


The points() function took in two vectors for the coordinates of where to plot the points, then plotted five different shapes using the “pch” parameter and a different color for each shape using the “col” parameter.

We can also combine different plots together using the par() command. After closing your old plotting window or opening a new one with x11(), try the following commands (requires the import of the data set from Section 6.2 — How to Read In Data):

```
> plot(proteinconc[,1])

> lines(proteinconc[,1])

> par(new = T)

> plot(proteinconc[,2])

> lines(proteinconc[,2], col=“red”)

> legend(x=“topright”, legend=c(“Nucleus”, “Nuclear Membrane”), lwd=2,
+ col = 1:2, bg=“grey”)
```

Now you should have the image below in a new graphics window:



You can see that by setting the “new” parameter in par() to “TRUE” we forced the new plot() to go on top of the old one. In addition, we introduced the function legend() which allows you to place a legend on your graph. The “x” parameter of legend is either a pair of coordinates (x, y) or a specific string such as “topright” which tells R where to put the legend. The “legend” parameter takes in a string of names and these are the categories of the legend. In this case, we wanted to use lines, so we set the line width to twice the normal size by using the “lwd” parameter. Other symbols can be used as well by specifying the “pch” parameter we have seen before. We then set the colors to match up with the names we provided using “col” and finally, we set the background color of the legend box to grey using “bg”.

Now you should have the ability to make some great plots of your data that can aid you in interpretation and in illustrating your data to others. The next section will show you how to export your data and these graphs.

Section 8: Exporting Data

After manipulating your data and using your data to create graphs you will often want to export the final product for use in presentations, storage, and other programs.

8.1 — Types of Output

There are three main types of output from R. Two of the main types of output are the same as the two main types of input — delineated files such as ".csv" and R object files such as ".rdata" (for more information on delineated file types, see Section 6: Reading in Data). Any R data type can be written to a ".csv" or ".rdata" file and these data files are stored in your current R working directory. The ".csv" files can then be copy-pasted into Excel or other programs, although often computers helpfully default to opening ".csv" files in Excel to begin with.

The third type of output is graphical output, and the most common form of graphical output is the ".pdf" file. Other file types such as ".jpg", ".png", ".bmp", and ".tiff" are available as well. Below, we see how to output in these formats.

8.2 — How to Export Data

First, we will look at how to export to ".csv" and similar formats. The write.csv() function is used for this task, and its parameters are basically the same as the read.csv() function. For example, let's create a new delineated file from a simple data frame that we create:

```
> printvector1 = 1:10

> write.csv(printvector1, file="myfirstoutput.csv")
```

Now, saved in your working directory should be a file named "myfirstoutput.csv". Open it and you should see that it contains two columns, the first is a list of index numbers and the second is the actual data (although in this case the data and the indices are the same numbers). Notice that the first input in write.csv() is the data and the second is the file name, entered using the parameter "file".

Let's try a slightly different output with a data frame as input. Input the following commands:

```
> printvector2 = 1:5

> printnames1 = c("Category 1", "Category 2", "Cateogry 3", "Category
+ 4", "Category 5")

> printdataframe1 = data.frame(printvector2, row.names=printnames1)

> printdataframe1
            printvector2
Category 1             1
Category 2             2
Category 3             3
Category 4             4
Category 5             5

> write.table(printdataframe1, file = "mysecondoutput.csv", sep=" ")
```

Now, you can open your file "mysecondoutput.csv" in Excel to see its contents. Also, try opening "myfirstoutput.csv" and "mysecondoutput.csv" in notepad to confirm for yourself that the separator was changed from a "," in the first to a " " in the second. Both files, however, use the ".csv" extension even though only one is actually comma separated.

Now, let's look at how to export R objects. This process is very simple and uses the save() command:

```
> savevector1 = 1:9

> savevector2 = "Yay!"

> save(savevector1, savevector2, file = "myfirstRobjects.rdata")
```

Now, you should have a file in your R working directory with the title "myfirstRobjects.rdata", which contains the objects 'savevector1' and 'savevector2'. You can include as many objects as you want in a ".rdata" file and they can be of any class.

The next major category of files that we want to export is image

files. The first type of image file we will look at is the ".pdf". Let's try an example:

> plotvector5 = 1:9

> plot(plotvector5)

> pdf("myfirstpdf.pdf")

> plot(plotvector5)

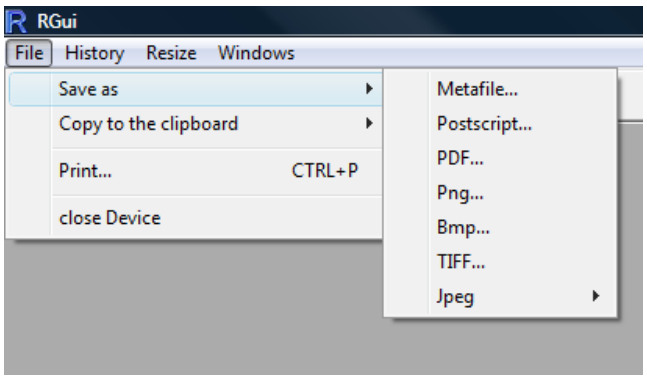
> dev.off()

You should now have a graphics window open with a plot and a new ".pdf" file in your working directory titled "myfirstpdf.pdf". It is not necessary to plot the vector first, but the pdf() function suppresses the opening of a new graphics window, so it is always good to first check what a plot looks like before saving it to pdf. The way the pdf() function works is that you type the name of the file you want in the parentheses, then hit enter and plot all the graphs you want. When finished, use the dev.off() command to turn off the pdf creation device. Each plot you created will be a new page within your pdf file unless you used a function like par(mfrow = c(x,y)) to put the plots on one page. Also keep in mind that when using Adobe Acrobat Reader, you have to close a pdf file before you can update it with new plots, or else R will give you an error.

Alternatively, you can save your graphs as other image files. To do so first plot what you want to save:

> plot(plotvector5)

A new graphics window should appear with your plot. Now, click on the graphics window and the toolbar options at the top of the RGui should change from what they are when you have the console or a script window selected to a new set of options. You can now click on File > Save As to get a list of the available image formats.



Simply click on any of the available objects to save your graph in that image format. The file will automatically be saved to your R working directory.

The functions above should allow you to save and export any type of graph or data that you create in R. I would encourage you to play around with each function as there are many adjustable parameters that can make your data or image into just the format you are looking for.

Section 9: Functions

Functions are the other bread and butter of R. They can be used to automate tasks, make code simpler, and allow other people to use your code more easily. You are actually rather familiar with functions already, but here we will formalize a bit more exactly what they are and how to use them.

9.1 — Built In

You have already encountered a variety of built-in functions. Everything from matrix() to plot() is actually just a built-in function that you can even modify if you want (although I would not recommend that for now)! Even c(), which you have used to create mathematical vectors of numbers and strings is a function. To see the code of any function simply type the name of the function. For example, try typing:

```
> plot
```

The command should return a complicated looking function — this text tells you exactly what plot() is doing with your input. Generally, you won't have to worry about interpreting this code. The simpler approach is to use the in-built help function. The way this help function works is you simply type a "?" followed by the name of the function. For example:

```
> ?plot
```

This command should return a new window with more informative information on exactly how to use the plot() function, which parameters it takes and what these parameters do, and a couple examples of using the function itself. I would highly encourage you to use this help function extensively, even though at first these help windows can seem a bit confusing. With practice, you will see that these help files are invaluable.

Now, let's look at some very useful built-in functions. What follows is simply a list of some of the functions I use the most followed by an example of its use. For more information on a function, simply access the in-built help using "?functionname".

Take an absolute value: abs()

```
> abs(-1)

[1]  1
```

Take a square root: sqrt()

```
> sqrt(c(2, 4))

[1]  1.414214 2.000000
```

Run a script without copy-and-paste: source()

```
> source("nameofmyscript.r")
```

Your script commands will then be run in the console.

Create a sequence of numbers: seq()

```
> seq(0, 8 ,2)

[1]  0 2 4 6 8
```

The first input is the beginning number, the second, the end number, and the third number is what you want to iterate by. If the numbers do not work out evenly, R starts at the lower number and truncates the result at the highest number within the set boundary.

Repeat numbers: rep()

```
> rep(c(0,2), 5)

[1]  0 2 0 2 0 2 0 2 0 2
```

The first input is a number or numbers that you want to repeat and then the second input is how many times to repeat that set.

Get the length of a mathematical vector: length()

```
> length(c(1, 5, 6, -2))

[1]  4
```

Time a process: system.time()

```
> system.time((seq1 = seq(0, 1e6, 1)))

    user     system     elapsed

     0.01        0.01         0.01
```

The time that R is actually taking to do the calculation itself is "user".

Find a variance: var()

```
> var(c(0, 5, 1, -10, 6))

[1]  40.3
```

Find a mean: mean()

```
> mean(c(0, 5, 1, -10, 6))

[1]  0.4
```

Find a median: median()

```
> median(c(0, 5, 1, -10, 6))

[1]  1
```

Sample with a probability (random vectors): sample()

```
> sample(c(0, 1), 10, replace = TRUE, prob = c(.5, .5))

[1]  1 1 1 1 1 0 0 0 1 1
```

You will likely not get the exact output above, as sample() creates a mathematical vector of numbers where the numbers come from your first argument and are sampled with the probabilities from the “prob” parameter, respectively. The second argument indicates the number of samples to take, and you can only sample without replacement if you have as many numbers provided as samples you wish to take.

Analyze vector information: which()

```
> samplevector = c(1:5, 1:2)

> which(samplevector == 1)

[1]  1 6
```

The which() function returns the location of where the Boolean value tested is equal to TRUE. In this case, the number 1 appears in the first element and the sixth element of the mathematical vector.

Hypothesis testing: t.test(), wilcox.test(), etc.

```
> t.test(1:5, 6:10)
```

The command above should return a lot of information on the results of a Student's t-test. R defaults to conducting a Welch Two Sample t-test when two data sets are provided. The t-test is just an example, however, as R essentially has a function for every type of hypothesis test. The tests are commonly named as testname.test().

Distributions: runif(), rnorm(), etc.

```
> runif(5, min = 0, max = 1)

[1]  0.5865153 0.5776229 0.9358731 0.7300911 0.7658244
```

The function runif() returns a random sampling of, in this case, 5 numbers from a uniform distribution where the samples have a minimum of 0 and maximum of 1, as set by the parameters "min" and "max". R has every major distribution built in and has functions for density, distribution, quantile, and random sampling for each. Use the built-in help function to access more information.

View what objects are currently in the workspace: ls()

```
> ls()
```

The function above should return a list of all the objects that are in the R workspace. All the objects in the workspace are accessible simply by typing their names. If an object is not in the workspace, you cannot access it and may need to load the object into your workspace.

Remove objects from the workspace: rm()

```
> rm(samplevector)
```

The rm() function removes objects from the workspace. If an object you remove is not saved separately as a ".rdata" file (or other file type) it is deleted permanently.

Quit R: q()

```
> q()
```

When you quit you will be prompted to save your workspace, if you do not save the workspace anything you did not export will be lost. If you do save the workspace, all the objects will be available next time you use R.

The list above is by no means exhaustive, but it should give you a good introduction to some of the built-in functions of R. Next, we are going to tackle how to create your own functions.

9.2 — Custom

Custom functions are a great way to keep your code organized and to save time. A new function can be written in either the console or a new script window, but it is much easier in the script window, so to begin open a script window (see Section 1.3 — How to Use if you do not remember how to open a script window). Now, in the script window we are going to write an example function. Type the following:

```
doubledip <- function(x, y) {

temp1 = x*y

temp1 = abs(temp1)

temp2 = sqrt(temp1)

z = sqrt(temp2)

return(z)

}
```

The above commands might seem a bit complicated, but as we go through them line-by-line everything should become clear. The first line assigns the function a name, in this case “doubledip”. Assigning a function a name is also the only time you must use “<-” instead of “=“. The part to the right of the arrow “function(x, y)” tells R that we are assigning the variable name a function and that the function can take two inputs, which we call “x” and “y” for internal use. Then, within the “{}” brackets, we perform our calculations. The only value that we return to the user of the function, however, is what is inside the return() brackets, which in this case is the variable “z”. If you do not specify a return() function, then R simply returns the last variable it modified. Let’s try using our new function now. Save the script file as “myfunctions.r”. A script file can have as many functions in it as you want, but you can only use a function after it has been run in the console window (so that the function is in the R workspace). Run the script and then type:

```
> doubledip(5, -2)

[1]  1.778279
```

Congratulations! You have successfully written and executed your first function in R. You now have a whole new world of abilities open to you.

One word of caution, however — make sure not to give names to your custom functions that are already the names of built-in functions such as plot() and c(). Your new function will override the built-in one! A quick way to check if a function name is already taken, is to simply type that name in the console window. If a function's code gets printed out, then there's a function, if not, then you're good to go.

Another important note about custom functions is that you will have to rerun them to use them every time you close and open R unless you save your workspace after you use the custom function.

You can find out much more about functions and how to use them in more advanced manuals. Please see Section 12: Further Resources for information on where to find these manuals.

Section 10: Tips for Writing Good R Code

Simply knowing all the commands available in R is just the beginning. Each programming language has its own tricks and optimizations, and R is no exception. This section will try and give you a broad overview of the most important best practices when programming in the R environment.

10.1 — General

One of the most important things you can do while programming is to comment your code, especially when writing functions or using script windows. Comments are placed by using the "#" character. For example:

```
sqrt(x) # Takes the square root of a number
```

Only the part of the code to the left of the "#" is actually run, the rest is just there for the human reader's benefit. Commenting is important not only because after writing a lot of code you probably won't remember exactly how the beginning of what you wrote works, but also because often it will be other people who are using your code. The example above is very simple, and you probably don't want to comment on the use of built-in functions such as sqrt(). But you should comment on how your custom functions work and make notes on generally what you are doing in each part of any script. Ask yourself, if I were seeing this code for the first time, could I understand it? Use your good judgment and common sense!

Also, when writing code (once again, especially functions) do not hard-code things like the length of a vector. Instead, use functions such as length() to find the value each time, even if you think it will not change. Doing so is a good programming practice and will save you a lot of headache in the future. For example, instead of coding the following:

```
sampleforloop <- function(inputvector) {

for(i in 1:10) {

inputvector[i] = i

}

}
```

You would want to code the following:

```
sampleforloop <- function(inputvector) {
for(i in 1:length(inputvector)) {
inputvector[i] = i
}
}
```

The second approach is better because you are dynamically getting the mathematical vector length for the “for” loop. You should take this step even if you are positive you will only be inputting vectors of length 10, as others who use your function may not!

Also, as a general rule you want to stay away from using single-letter variable names. As has been mentioned before, letters like “c” are actually the names of critical functions in R! Using “c” as a function name will replace this function, while using “c” as a variable name will not replace the function c(), but will make it so that you cannot type “c” to see the function’s inner workings. Another example is that “T” (TRUE) could accidentally be reassigned by you to “F” (FALSE).

The final general tip I find helpful is that you should use short variable names when coding in R. This manual uses long variable names for the purpose of clarity, but when you are writing R code you should try and keep your names concise yet clear. Additionally, when writing R names do not start variable names with numbers, and avoid using “_” and “$”, but you can freely use “.” (unlike in some other programs).

10.2 — Matrix Multiplication

R was built around the idea that most of your computations would involve matrix–matrix multiplication, so you should try and avoid loops and use matrices to their full extent. This tip may not matter much at first, but as you start writing longer programs and using larger data sets, using matrix multiplication or using a “for” loop can mean the difference between waiting a day for your program to run and waiting an hour. When “for” loops are unavoidable, those with programming experience can read manuals on how to make C interface with R, and write the “for” loops in C. A simple example of how you can replace a “for” loop with matrix multiplication is the following:

```
matrix1 = matrix(sample(c(0, 1), 100, replace = T), nrow = 10)

matrix2 = matrix(rnorm(100), nrow = 10)

matrix3 = matrix1%*%matrix2
```

The above code takes a matrix of 1s and 0s and a matrix of values (in this case randomly taken from a normal distribution) and then uses the 1s and 0s to essentially "sum" the columns of the values matrix (sketch out this code and think about the linear algebra). Alternatively, nested "for" loops could have done the same calculation, but much more slowly:

```
for(i in 1:length(matrix1[1,])) {

        for(j in 1:length(matrix1[,1])) {

        matrix3[i,j] = matrix1[j,]%*%matrix2[,i]

        }

}
```

As you can see, taking a moment to think about the linear algebra saves a lot of headache!

10.3 — Plan

Perhaps the most important tip about programming in this section, planning is critical to success. You should spend at least twice as much time sketching out and formulating how you are going to program than doing the actual programming itself. This way, you can quickly correct mistakes on paper before you have coded anything and don't have to go back and delete what you have written. This tip is true for any programming language, not just R.

10.4 — Debug

Debugging your code can be one of the most frustrating aspects of programming. But don't despair! Just stay calm and think through your program from beginning to end. Often, it is very useful to get some paper and sketch out what happens to some sample input you have created at each stage (see planning above). In this way, you will very often catch obvious mistakes in logic and can check your output values as you go.

When debugging a function you have written I like to use the function debug(). To use this function you simply type:

```
> debug(functionname(input))
```

Each time you press enter the debug command will output the value of the next line of code in your script. This way you can go line-by-line through your function and catch at what point things start going awry. You can also hard-code print() statements at critical points in your program that output the value of some data object. For example after performing a calculation, you can add a print() statement that outputs the result:

```
samplevector1 = samplevector1^2

print(samplevector1)
```

Feel free to sprinkle these statements liberally throughout your code, they can only help.

10.5 — Help

As mentioned before, the "?functionname" built-in help function is probably your best help resource in R. In addition, there are many forums online where you can find fellow users of R ready to help you. Some of these forums are listed in Section 12: Further Resources. Remember also that those around you can be your best resource! Ask your friends and professors who use R any questions you have and maybe you'll end up teaching them something they didn't now about R either.

10.6 — Packages

The ability to download and install packages is one of the key factors that makes R an excellent language to be proficient in. Many research papers will include the methodology they used as an R package freely available for use. Because R is free to use, there are many packages for niche needs that you will not find available in other programming languages. Sometimes, if you face an especially complicated task you can look online at http://cran.r-project.org/web/packages/ and find a package that does what you are looking for. Unfortunately, installing and using packages is beyond the scope of this manual. Look at more advanced manuals in Section 12: Further Resources for this information.

Section 11: R Editors

There are many different “editors” you can use to write R code. Editors are essentially the textpad you will use to type in your code. There is no single editor which is “best” but instead each has its benefits and drawbacks, the choice of which editor to use comes down to personal preference.

11.1 — Built-In

The RGui has a built-in editor, namely the script windows that you have been opening to type in longer bits of code. These script windows are severely limited, however, as they do not have what is called “syntax highlighting”. Syntax highlighting is where the words in your code are divided into colors based on their function (for example, comments are in green, loop names are in red, etc). In addition, modern editors also help you keep track of where your parenthesis and brackets are, so that you do not get confused and can auto-complete commonly used commands. Therefore, it is highly recommended that after getting mildly comfortable with R, you download one of the editors mentioned below and use it for any long coding projects you want to do.

11.2 — Others

One of the most popular editors for R is WinEDT. WinEDT is unfortunately not a free program; it is instead shareware, which means that you will have to redownload it each month. It is a very handy editor, however, and you can of course pay for it if you appreciate its benefits. You can find WinEDT here:

WinEDT: http://www.winedt.com/

Another editor that is great for those just beginning to program is Tinn-R. Tinn-R is entirely free and contains many of the features of WinEDT. This editor is recommended for those who are new to programming:

Tinn-R: http://www.sciviews.org/Tinn-R/index.html

Aside from the above two editors, many popular editors such as Eclipse and Emacs can be configured to use R syntax highlighting as well. Those who already program in these environments may find using these editors easier than downloading an entirely new one.

Section 12: Further Resources

CRAN Home Page: http://cran.r-project.org/

Useful site for updating R and finding more information on packages and current R events.

R Journal: http://journal.r-project.org/

An academic journal that has articles directly relating to R and new R packages.

R Manuals: http://cran.r-project.org/ > Documentation > Manuals

Out of the manuals listed, I highly recommend “An Introduction to R” as the next manual you look at. At this point, you shouldn’t have to read it all the way through; you can just look at the sections that relate to what you are programming so you can get a bit more insight into that area.

R Forum: http://www.nabble.com/R-f13819.html

This forum is just one of many active forums on the R programming language that can be found through a simple online search.

Section 13: Acknowledgements

I would like to first thank all those who have taken the time to write manuals before this one, as I have gained a new appreciation for your work after undergoing this task myself and learned so much from your manuals. Also, I would like to thank all the members of John Storey's lab who have proofread this manual and offered their counsel and advice, especially Keyur Desai and Narayanan Raghupathy, and John Storey himself. I also thank all my friends who have offered their comments and ideas. Finally, I would like to thank the Integrated Science program at Princeton, without which I would likely not have gotten involved in computational science.

This work was supported in part by NIH grant R01 HG002913 (PI: John Storey).