Introduction to scientific programming in R*

John M. Drake

1 Introduction

This course will use the R language programming environment for computer modeling. The purpose of this exercise is to introduce R. This exercise addresses only those features of R that are required for this course. A more complete introduction and handy reference can be found on the Internet at http://cran.r-project.org/doc/manuals/R-intro.html.

2 What is R?

R is both a "high level" computer language and a software platform for writing and executing programs in that language. The structure of R is well suited to scientific programming. It efficiently performs many, many numerical procedures, including many that are extremely useful for statistical analysis and visualization. In what follows, we assume that the student has had no prior experience with R. The goal of this session is to develop some facility with the tasks that will be needed later in this course. Naturally, we do not aim to provide a comprehensive overview of either scientific programming or of R. Rather, these exercises are intended to function as a starting point for further exploration and self-study with an emphasis on applications and illustrations from theoretical epidemiology. All that is required for completing the exercises is the "base" R program. RStudio, which may be downloaded from http://rstudio.org, is recommended for those seeking a more featured development environment.

3 Assigning variables

If you have not done so already, open R. Whether on Mac, PC, or Unix/Linux you should be presented with the *console*, a terminal-like interface with a command line. One way to interact with R is to submit commands to the console. In R, assignment is made with the expression `<-`. For instance, we assign the value 10 to the variable `a` as follows.

```
> a<-10
```

In R numerical quantities may be stored as any of a variety of object types, including vectors, matrices, data frames, time series and lists.

4 Navigating help

Before going further, it will be useful to know where to turn for help. If you just want to browse the help, use the drop down menu. If you know the name of the command that you want help for then just type the name of the command at the command line, preceded by a question mark. For instance, the function `lm` fits a linear model. Type `?lm` to obtain the help for this command.

While some operations (such as assignment, reviewed above, and most numerical operations like +, -, *, and /) are simply instructions, others are *functions*, that is subroutines that require inputs and return outputs. The help on an R function explains what form these inputs/outputs must take as well as any instructions needed for using the function. Help files often contain examples which can be consulted for guidance on use.

Exercise 1. Look at the help for the function `lm` and answer the following questions.

- What are the function inputs called in R terminology?
- What are the function outputs called in R terminology?
- What else does the help file tell you about `lm`?

Exercise 2. Copy and paste the first seven lines from the example in the `lm` help file. Type the following commands into the console and view the output: `lm.D9`, `summary(lm.D9)`, `attributes(lm.D9)`. What is being returned in each case? What is lm.D9?

5 Writing programs

Entering every command in the console is cumbersome. Typically, therefore, one works by writing commands to a file (called a *script*) and either executes all the commands in the script in sequence using the R function `source` or copies the commands from the script to the console using F5 or Ctrl-R. An editing window can be opened from the File drop down menu. Commands written here may be saved as plain text files with the file extension ".R". R Markdown documents should be saved as plain text files with the file extension ".Rmd". Comments (*i.e.*, text that should not be interpreted as code) should be preceded by "#".

You will find it useful to be able to write your own functions. This is accomplished using the R function `function`. The function specification always comprises the same components: a name, assignments, arguments (in parentheses), braces, and a value to return. For example, a function to add two numbers and divide by a third might be specified as follows.

```
> my.function1<-function(a,b,c){
+   tmp<-a+b
+   tmp2<-tmp/c
+   return(tmp2)
+ }
```

Using this function (two examples) we have:

```
> out1<-my.function1(4,7,2)
> out1

[1] 5.5
```

```
> out2<-my.function1(2,5,6)
> out2

[1] 1.166667
```

Of course, the functions we will typically want to create will be much more elaborate.

Exercise 3. Write a function that takes a vector of data (e.g. case counts) and returns a vector that is the corresponding sequence of ratios of the data (i.e. growth rates).

6 Adding packages

Base R comes with only so many functions. As special purpose functions are developed, they are commonly compiled into libraries called *packages*. To use the functions available from a specific package, the package must be downloaded and installed. Later in this module we will need a function (`ode`) for numerical integration of ordinary differential equations. `ode` is available in the package `deSolve`. To install this package, run the command `install.packages('deSolve')` and choose a nearby mirror site. Everything else should follow automatically.

Adding packages cannot be done in this way if you do not have read/write access to the default installation directory. In this case, a new installation directory can be created, for instance "C:\Users\Drake\R" (example for Windows). The following code will make this directory the default for any future installations and add it to the search path (*i.e.*, will instruct R to look there for libraries that might be required to execute a program.)

```
> .libPaths(c("C:\\Users\\Drake\\R", .libPaths()))
```

You will want to add this line of code at the start of any of your programs.

7 Loading data

Data can be stored in lots of different ways. Accordingly, there are lots of different ways to load data into R. For instance, you could type in all your data and place them in a vector using the generic "combine" function `c` as follows.

```
> data.demo<-c(1,17,4,6)   #create a data vector
> data.demo

[1]  1 17  4  6
```

Now is a good time to introduce an important aspect of *referencing* and a trick. First referencing. When you have a vector and want to refer to just a part of it you indicate this with square brackets listing the element numbers. For instance, to refer to the third element of `data.demo` we type

```
> data.demo[3]

[1] 4
```

to refer to the first through third elements we type

```
> data.demo[1:3]
```

```
[1]  1 17  4
```

and to refer to the first, second, and fourth elements we type

```
> data.demo[c(1,2,4)]
```

```
[1]  1 17  6
```

When data are stored as a matrix or a data frame we use two values corresponding to the row number and column number as follows.

```
> data.demo2<-matrix(c(4,7,89,3,4,4,15,0,1),nrow=3)   #create a data matrix
> print(data.demo2)
```

```
     [,1] [,2] [,3]
[1,]    4    3   15
[2,]    7    4    0
[3,]   89    4    1
```

```
> data.demo2[1:2,3]
```

```
[1] 15  0
```

with the following concise notation when we want all the elements of a give row or column referenced.

```
> data.demo2[,3]
```

```
[1] 15  0  1
```

Now the trick. If we want to create an object with nothing in it (perhaps as a storage place for future use), we just combine with no argument, i.e., `c()`.

Now, back to data entry. For research purposes, it is often most convenient to enter data into a spreadsheet or database management software and then save as comma-separated *flat files* with a ".csv" extension. Then, the data are read into R using the function `read.csv`. For the purposes of this class, we have provided a number of datasets that are already stored in R formats. These may be found in the file "data.RData". To read into R enter the following command.

```
> load('data.RData')
```

Now that the data have been read into memory, they are available for use. To see everything that's in memory at the present time, enter the command `ls()`. We can inspect the influenza dataset by typing its name into the console.

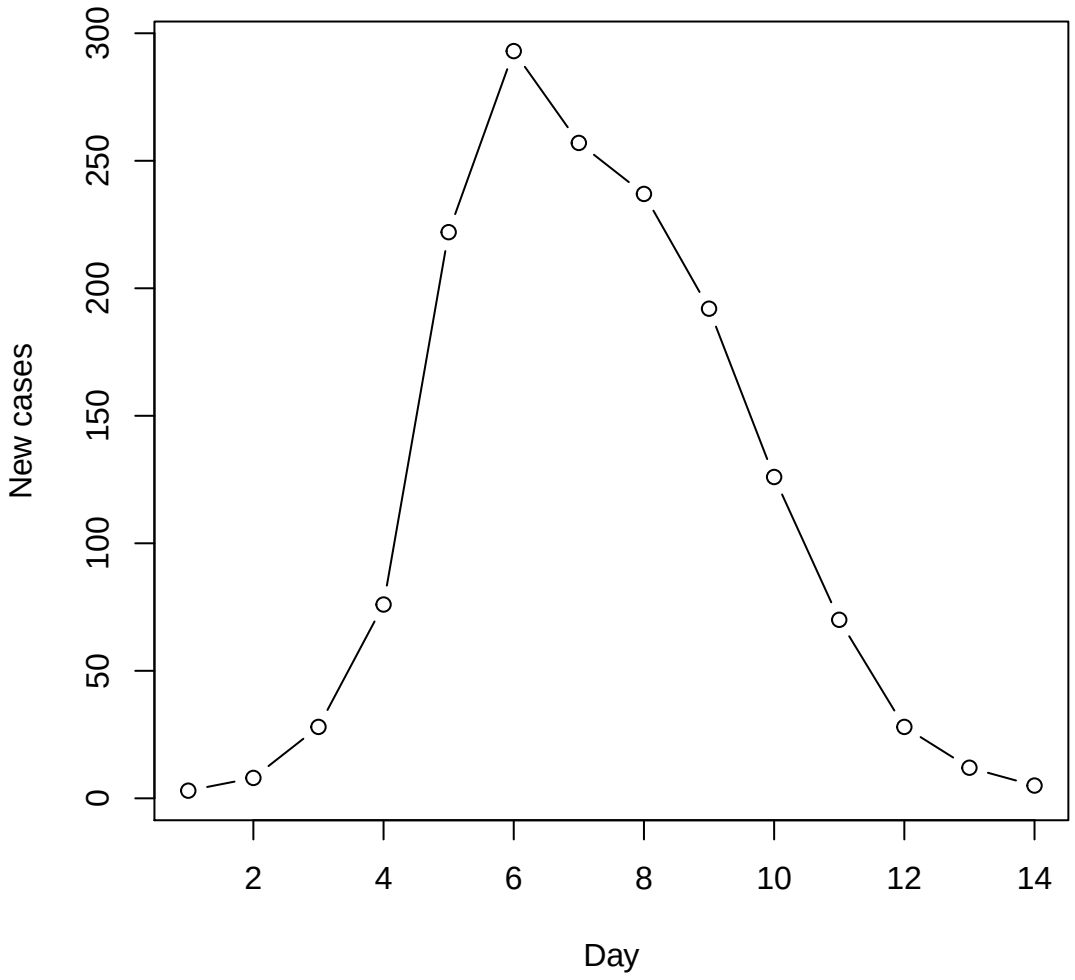
```
> flu
```

```
   day flu
1    1   3
2    2   8
3    3  28
4    4  76
5    5 222
6    6 293
7    7 257
8    8 237
9    9 192
10  10 126
11  11  70
12  12  28
13  13  12
14  14   5
```

These data (and the other three datasets included here) are stored as an R object called a *data frame*. The data frame allows us to store a table of data, the columns of which may have different formats (e.g., numerical, categorical, etc.). The data frame is one class of R objects that is commonly used. In this course we will use mainly data frames and numerical objects (vector and matrices) though there are many, many kinds of R objects, some of which are very flexible (e.g., lists) and some of which have very specific uses (e.g., time series).

What do the data represent? These data are from an outbreak of influenza in a British boarding school in 1978 (Anonymous 1978). There is a unique entry in the variable `day` for each day of the outbreak. The variable `flu` contains the corresponding number of boys confined to bed. We can plot these data using the following command.

```
> plot(flu$day,flu$flu, type='b',xlab='Day',ylab='New cases')
```



We learn a number of things by studying this code. First, a single variable can be referenced by typing the name of the data frame (e.g., `flu`) followed by the name of a variable (e.g., `day`) separatated by a dollar sign. This allows us to extract, manipulate, assign variables independently. For instance, the following lines creates a new object (a vector), called `prevalence`, by extracting flu cases and dividing through by a presumed population size of 764 (763 boys "at risk" plus index case).

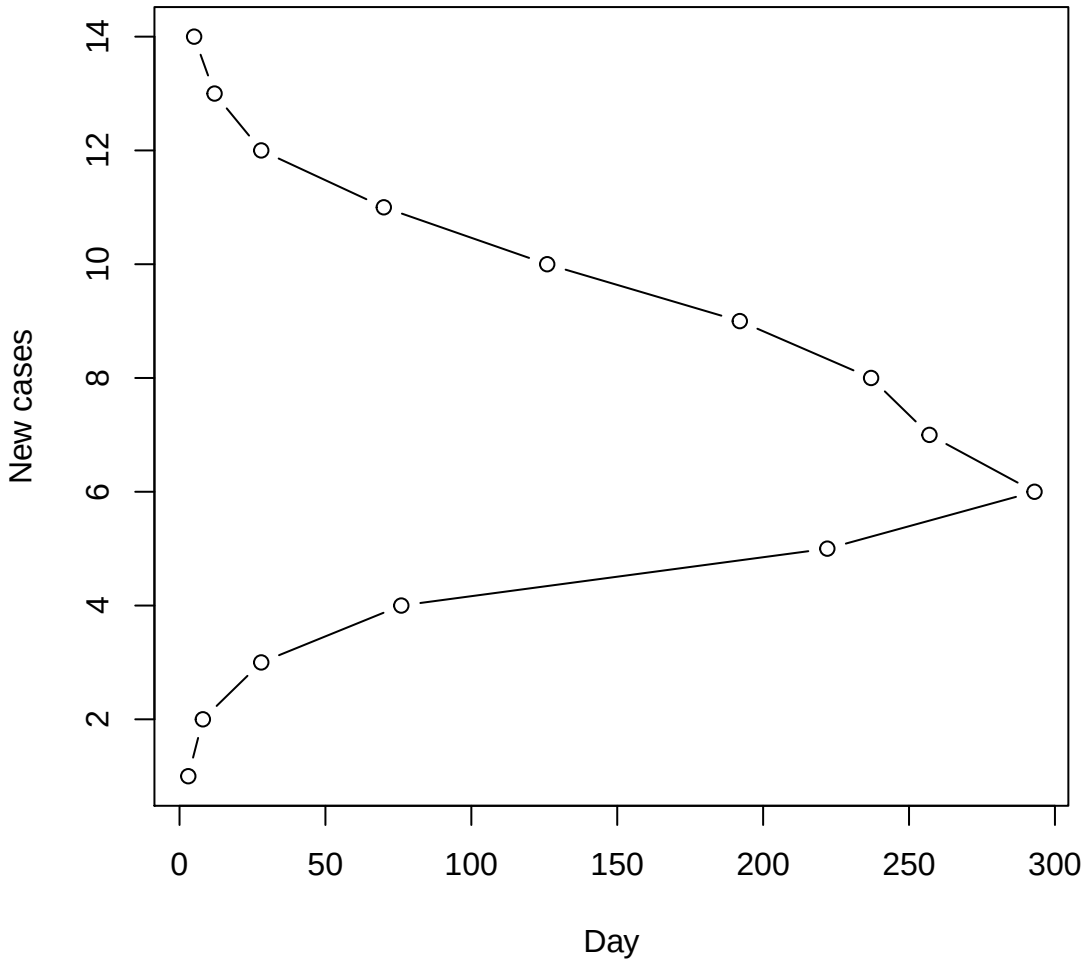
```
> prevalence<-flu$flu/764
> prevalence

 [1] 0.003926702 0.010471204 0.036649215 0.099476440 0.290575916 0.383507853
 [7] 0.336387435 0.310209424 0.251308901 0.164921466 0.091623037 0.036649215
[13] 0.015706806 0.006544503
```

The next thing to notice about our plot command is that the first two arguments are not names. Rather, `plot` just assumed that the first two arguments should be placed on the x-axis and y-axis respectively.

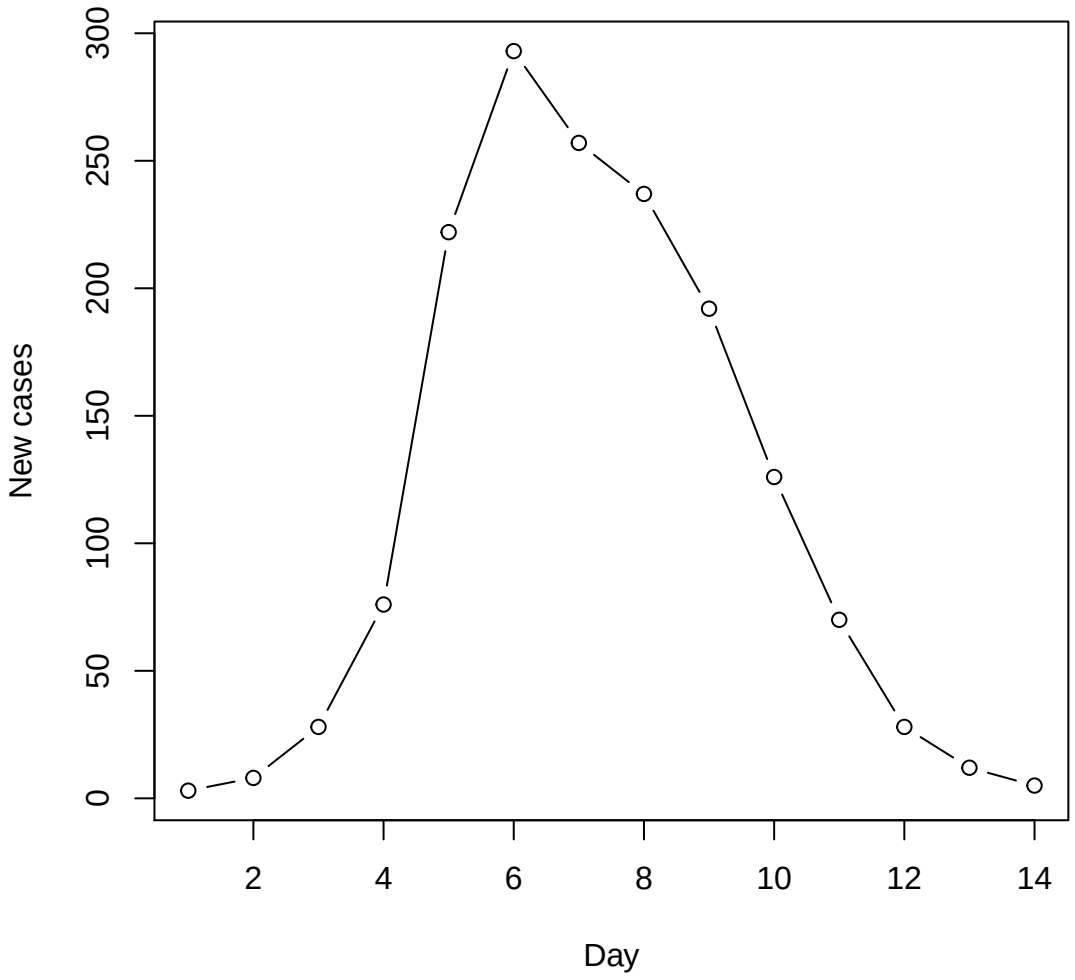
If we switch the sequence, we obtain the following plot.

```
> plot(flu$flu,flu$day, type='b',xlab='Day',ylab='New cases')
```



Alternatively, the order doesn't matter if we specify which value is which argument, i.e., the following line reproduces the original plot even though the x and y arguments are out of sequence.

```
> plot(y=flu$flu,x=flu$day, type='b',xlab='Day',ylab='New cases')
```



Finally, we see that `plot` may take additional arguments (indeed, many more arguments than those used here are possible). Here we specify the type of plot and labels for the x and y axes.

Exercise 4. The plots generated above consist of 'b'oth *points* and *lines*, hence the argument `type='b'`. Retrieve the help for the plot function and try some of the other plot types, e.g., line plot and point plot. What does `type='h'` do?

The function `plot` is intended only for producing scatterplots (in several flavors). Other functions are available for other kinds of plots, including `hist` to generate histograms and `boxplot` to generate box and whisker plots. Additionally, all aspects of the image may be manipulated (e.g., line thickness, colors, fonts, etc.) with greater or lesser difficulty depending on how deeply related they are to the plots design. Also, lines and points may be added to an existing plot using the functions `lines` and `points`, respectively.

Exercise 5. Experiment with the functions `plot`, and `points` by plotting the measles incidence data from three different sites in the city of Niamey, Niger in different colors on the same figure.

8 Control statements

Control statements are used to determine the flow of a computer program. Control statements include conditional commands (i.e., `if` and `else`) and the commands that allow *looping*. Often, effective use of control statements requires *grouping*. Grouping is achieved by enclosing a set of codes in curly braces { }. The value of the group is not determined until all the enclosed statements are evaluated, at which point the value is given by the last expression in the group. Note that we have already used grouping in function definition above.

Conditional execution in R is achived using the `if` statement, optionally followed by `else`. Repeated evaluation is typically performed with a `for` loop. The use of these control statements is easiest to learn from illustration. For instance, the following code shows both grouping and the use of `if` and `else`.

```
> a<-5
> if(a<1){
+   b<-5
+   c<-4
+ } else {
+   b<-10
+ }
> b

[1] 10

>
```

Here we look at a simple `for` loop.

```
> a<-c()
> for(i in 1:10){
+   a[i]<-i^3
+ }
> a

 [1]    1    8   27   64  125  216  343  512  729 1000
```

9 Random number generation

We conclude this session with an introduction to (pseudo-) random number generation in R. Both statistical analysis and numerical simulation can require random number generation and R is enabled to generate random numbers from any of a large number of distributions, including beta, binomial, Cauchy, chi-squared, exponential, Fisher, gamma, geometric, hypergeometric, logistic, lognormal, negative binomial, normal (Gaussian), Poisson, Student t, uniform, and Weibull. Each of these distributions has an abbreviated "code name". For instance, for the normal distribution it is `norm` and for the uniform it is `unif`. Generating random numbers with these distributions is achived by prefixing the code name with the letter "r" so that generation of uniform random numbers is done with the function `runif`, generation of normally distributed random numbers is done with the function `rnorm`, and so forth. When used, the first argument will be the number of random numbers to be generated. Other arguments are the parameters of the distribution. So, for example, the following code generates 3 random numbers from a normal distribution with mean 5 and standard deviation 3.

```
> random.numbers<-rnorm(3,mean=5,sd=3)
> random.numbers

[1] 5.137939 1.912747 5.802751
```

In a later session we will generate random numbers from the exponential distribution with the function `rexp`. This distribution has only one parameter, the rate, which is also the reciprocal of the mean.

Exercise 6. Generate one hundred random numbers and plot a histogram.