A Brief Introduction to R

Hugh C. Pumphrey

September 22, 2010

Abstract

These lectures[1] are intended to provide you with an introduction to the data analysis and plotting package called R. They cover, in three sessions, all that you need to know to get yourself started using R and to begin producing useful results. They certainly will not teach you everything that R can do – that would take a whole term or more. It is important to remember that learning any computing language or package is largely a matter of practice. You will not learn much by just listening to me. I can show you the sort of things that can be done and I can show you how to get started. Becoming proficient is up to you – I hope these lectures stimulate you to go away and do just that. These lectures do not assume any knowledge of R at all. They do assume that you are comfortable using the PCs running Scientific Linux and that you can program in some reasonably modern programming language (*e.g.* C, Fortran 90, pascal, python, java, Basic ...). These notes are all available online at `http://xweb.geos.ed.ac.uk/~hcp/r_notes.pdf` for easy cut-and-pasting of the examples.

1 Lecture 1: Getting Started

1.1 So, what is R? What can it do for me?

R is a data plotting package and a programming language. In R, you issue commands or combine them to make programs, in order to analyse your data and turn it into exciting-looking, meaningful plots. R is actually a free implementation of the S programming language. [2] Its origins are in the statistics community but it is spreading into many other scientific areas. R is both free, and cross-platform, so you can obtain and install it on your Windows, Linux or MacOS X machine at home at no cost.

R is an example of a type of product which one might call a *scientific data-analysis language.* Matlab and IDL are commercial examples of this kind of tool. Other free examples include Octave, Yorick and Scilab. It is generally the case that having learned to use one of these tools, learning to use one of the others is just a matter of learning a new syntax, not of learning a new way of thinking. This is the real reason for teaching R: you may or may not use R for your project or for work later in your career. But you are almost certain to use a *scientific data-analysis language* of one sort or another.

1.2 Starting R.

To start R, type "`R`" at the prompt in a terminal window. You should get a response like this:

```
halo.hcp>R

R version 2.4.1 (2006-12-18)
Copyright (C) 2006 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
```


[2] You can buy a commercial implementation of S called S-Plus, but R is now sufficiently good that there is little point.

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

If you get that, you are ready to go — the `>` symbol is the R prompt, waiting for you to type a command. If you get something else, ask your instructor what is wrong. Let's assume now that R is working properly and try out the demos. If you type `demo()` at the R prompt you will get a list of available demonstrations (you may need to press `q` to leave the list). Type `demo(graphics)` to get an idea of R's plotting capabilities.

The welcome message tells you how to access the on-line documentation, which is quite detailed. It includes a longer introduction than this one, and a formal language definition. Under the heading `Packages` is information on all of R's commands and functions — most of the functions we will be using are in package `base`. Use of the on-line documentation is *very important*. You will be expected to be able to use it effectively. It should be your first port of call when you have a question about R, because it will nearly always have the answer. Often `help(something)` doesn't tell you what you need to know, because there is no R command called `something`. R will then suggest that you try `help.search("something")` (you need the `""`), which will do a broader search and try to suggest the command that you might be looking for.

1.3 A First Plot

Now that we have R working, let's make a plot with it. Let's suppose that we have several points (each of which has a horizontal and a vertical co-ordinate) and that we want to plot them on a graph. We'll be traditional and call the horizontal co-ordinates x and the vertical co-ordinates y. We can enter the data into R like this[3].

```
> x <- c(1, 2, 4, 5, 6.7, 7, 8, 10 )
> y <- c(40, 30, 10, 20, 53, 20, 10, 5)
```

We now have two array variables called x and y, each containing eight numbers. Note that the left-pointing arrow is traditionally used for assignment in R, although recent versions permit the = used in many other languages. The R command `c()` is used to combine several numbers into an array. You can print the contents of a variable by typing its name:

```
> y
[1] 40 30 10 20 53 20 10  5
```

To make a plot of x against y, we just type this:

```
> plot(x,y)
```

and there it is. It is a bit plain, but we have visualised our data. No-one else is going to know what it means unless we label the axes, like this:

```
> plot(x,y,main="Should he resign?",xlab="Time / weeks",ylab="Popularity")
```

We suppose that the data is the popularity ratings of a politician as determined by a polling organisation. Notice that there are two different ways in which we have provided instructions to the 'plot' routine. The data, `x` and `y`, are provided as positional parameters. The order they come in matters. Here, the first parameter is the horizontal co-ordinates of our data, the second is the vertical co-ordinates. Optional things like the labels are passed as keyword parameters. An important R routine like `plot` usually has many of these. You can supply them in any order you like.

Now let's suppose that our politician's popularity is being measured by two polling organisations. We put both sets of measurements on the same plot by using `plot` for the first set and `points` for the second. We can also use `lines` to add lines joining the dots:

```
> y2<- c(44, 27, 14, 21, 50, 22, 11, 3)
> points(x,y2,col="red",pch=3)
> lines(x,y,col="blue")
> lines(x,y2,col="green",lty="dashed")
```

[3] I have shown the R prompt > in the examples: Remember not to type it when you are trying them out.

Note how we use the lty (line type) and col (colour) keywords to make the second line different from the first.

This is getting to the stage where it might be tedious to re-type everything to correct a mistake we made earlier. We can avoid this by putting a list of R commands into a file to make a program. As an example, use your favourite text editor[4] to create a file called dubya.R and put the following lines in it:

```
# Program to plot fake poll results
x <- c(1, 2, 4, 5, 6.7, 7, 8, 10 )   ## time
y <- c(40, 30, 10, 20, 53, 20, 10, 5) ## first dataset
y2<- c(44, 27, 14, 21, 50, 22, 11, 3) ## second dataset
plot(x,y,main="Should he resign?",xlab="Time / weeks",ylab="Popularity")
points(x,y2,col="red",pch=3)
lines(x,y,col="blue")
lines(x,y2,col="green",lty="dashed")
legend(7,50,c("Gallup","Mori"),col=c("blue","green"),lty=c("solid","dashed"))
```

Note that the # character is used to indicate a comment – R ignores everything on a line after the first #. To run the program, type

```
> source("dubya.R")
```

It is conventional for the filename to end in `.R`; I suggest you stick to this convention. The file also has to be in the directory from where you started R (or you have to give a full pathname). If you have never used R before, why not take this program, run it, and then make some changes to it. Read up on the 'plot' function in the manual and find out how to do things like change the axis ranges, use logarithmic axes, and so forth. One last thing to note about running R programs is that just giving the name of a variable does not print its value as it does at the prompt. Instead, you need to use the `print` command.

1.4 Importing data

You now know how to make a line plot of some data. In most cases you will have far too much data to want to type it in. Typically the data will be in a file and you will want R to read it in and then make a plot of it. As an example, we will use some numbers generated by the MODTRAN radiative transfer package. The instructions for one of the M.Sc practicals show you how to run MODTRAN and how to remove the header lines from the file. This leaves you with a file containing 14 columns of numbers, like this:

```
 14000.   0.714  7.97E-31  1.56E-26  2.10E-07  4.12E-03  4.11E-08  1.18E-07  2.32E-03  2.94E-08  3.29E-07  6.44E-03  1.64E-05 0.3950
 14100.   0.709  4.95E-31  9.84E-27  2.13E-07  4.24E-03  4.15E-08  1.18E-07  2.35E-03  2.90E-08  3.31E-07  6.59E-03  4.96E-05 0.3940
 14200.   0.704  3.05E-31  6.15E-27  2.11E-07  4.25E-03  4.17E-08  1.15E-07  2.32E-03  2.81E-08  3.26E-07  6.58E-03  8.22E-05 0.3887
 14300.   0.699  1.84E-31  3.77E-27  1.90E-07  3.90E-03  4.06E-08  1.03E-07  2.11E-03  2.56E-08  2.93E-07  6.00E-03  1.12E-04 0.3734
 14400.   0.694  1.13E-31  2.35E-27  1.87E-07  3.88E-03  4.05E-08  9.90E-08  2.05E-03  2.46E-08  2.86E-07  5.93E-03  1.40E-04 0.3664
 14500.   0.690  6.53E-32  1.37E-27  1.60E-07  3.36E-03  3.74E-08  8.57E-08  1.80E-03  2.17E-08  2.46E-07  5.17E-03  1.65E-04 0.3446
.
.
.
 33800.   0.296  0.00E+00  0.00E+00  2.64E-10  3.01E-05  2.62E-10  2.91E-18  3.33E-13  1.99E-20  2.64E-10  3.01E-05  3.27E-03 0.0001
 33900.   0.295  0.00E+00  0.00E+00  1.68E-10  1.93E-05  1.67E-10  1.52E-19  1.75E-14  1.05E-21  1.68E-10  1.93E-05  3.27E-03 0.0000
 34000.   0.294  0.00E+00  0.00E+00  1.82E-10  2.10E-05  1.80E-10  1.83E-20  2.11E-15  1.26E-22  1.82E-10  2.10E-05  3.27E-03 0.0000
```

The columns are 201 rows long. Let us suppose that we want to plot the 12th column (which is total radiance) against the 2nd column (which is wavelength). Here is a short R program which will do this.

```
## Read in rectangular table of data

moddat <-read.table("/home/hcp/wrk/rpms/modtran/pldat.dat")

## Result is a list, with each element of the list being a column from
## the table. The first column becomes moddat$V1, the second moddat$V2 etc.

## Plot col 2 against col 12
plot(moddat$V2,moddat$V12,xlab="Wavelength / microns",
     ylab="Radiance in Watts/(cm2 Steradian Micron)",type="l")
```

1.5 Printing your graph.

It's great to have plots on the screen, but you will often need to get them on paper to include in reports etc. R treats the screen as one device and a postscript file (which you can print) as another. You can save a graph you have on the screen as a postscript file like this:

[4]You can use pico for simplicity if you like. The emacs editor is a good choice for working in R because it will switch automatically into a special R mode to provide sytax colouring.

```
> dev.copy2eps(file="foo.eps")
```

The resulting file `foo.eps` can be printed, viewed on screen or included in a report. If you want R to generate a postscript file without displaying anything on the screen, you can do it by using `postscript()` before your plotting commands and `dev.off()` after them. Here is a variant on the modtran plotting program which does exactly this:

```
## Read in rectangular table of data
moddat <-read.table("/home/hcp/wrk/rpms/modtran/pldat.dat")
## Result is a list, with each element of the list being a column from
## the table. The first column becomes moddat$V1, the second moddat$V2 etc.

# Start a postscript plot
postscript(file="modtran.eps",onefile = FALSE,horizontal=FALSE,
           width=6,height=5)

## Plot col 2 against col 12
plot(moddat$V2,moddat$V12,xlab="Wavelength / microns",
     ylab="Radiance in Watts/(cm2 Steradian Micron)",type="l")

# Make sure the postscript file is closed
dev.off()
```

The `postscript` command has a variety of options to control the size of the plot, the sort of paper it is to be printed on, whether it is to be included in another document etc.

1.6 Variables and data types in R

1.6.1 Types of variables

As with most languages, R has several types of variables. I list some of them here:

type	range	to define	to convert
numeric	numbers	`x<-1.0`	`x<-as.numeric(y)`
character	text strings	`x<-"flarp"`	`x<-as.character(z)`
logical	TRUE or FALSE	`x<-TRUE`	`x<-as.logical(w)`

Variable names can have letters, numbers and the full stop in them, but they can NOT have underscores[5]. They are are case-sensitive: foo, Foo and FOO are all different variables. Numeric variables can be integers or floating point, but they are usually floating point and the distiction is less important in R than in most other languages.

R is a dynamically typed language. That means that you don't have to define your variables or say which variable is which type at the start of your program. You can say `gak <-37.5` at any point in your program and a numeric variable called gak will spring into existance and take on the value 37.5 . If you had a variable of a different type called gak at some earlier point in your program, it will vanish when you create the numeric variable with the same name.

1.6.2 Arrays

In addition to single, scalar variables, R has vectors and arrays. A vector is a numbered group of variables, all of the same type. We used vectors in the first lecture to hold the data for line plots. You can define a vector by using the function `c()`

```
> a.vector <- c(3, 5, 6, 2.5, 100, 27.7)
```

You can refer to individual elements of the vector like this:

```
> a.vector[1]
[1] 3
> a.vector[3]
[1] 6
```

Note that the first element of the array is numbered 1 as in Fortran (not 0 as in 'C', Java and python). Note also that you must use `[]` for indexing arrays. You can refer to a subset of the vector like this:

[5]Not traditionally, anyway. A recent relaxation of the language definition now allows this

```
> a.vector[2:4]
[1] 5.0 6.0 2.5
```

R allows you to arrange your data into a 2-dimensional array (a matrix) or indeed an array with any number of dimensions. An array in R is a vector with a set of dimensions attached. We can make a copy of our vector and convert it into an array, like this:

```
> a.matrix<-a.vector
> dim(a.matrix)<-c(2,3)
> a.matrix
     [,1] [,2]  [,3]
[1,]    3  6.0 100.0
[2,]    5  2.5  27.7
```

... and refer to parts of it like this:

```
> a.matrix[1,2]
[1] 6
> a.matrix[1,]
[1]   3   6 100
> a.matrix[,2]
[1] 6.0 2.5
> a.matrix[1,2:3]
[1]   6 100
```

Note how an omitted index is used to represent an entire column or row of the array and how two integers and a colon are used to select part of a row or column. Finally, you can set up an array to use later like this:

```
> foo<-array(3,c(10,10,10))
```

will make foo a 10 by 10 by 10 element numeric array with all elements initialised to 3.0. For 2-dimensional arrays you may find the `matrix` command is more intuitive.

1.6.3 Arithmetic with arrays

One of the most powerful features of R is that you can do mathematical operations on entire vectors and arrays.

```
> x<-matrix(c(2,4,6,8),nrow=2)
> x+10
     [,1] [,2]
[1,]   12   16
[2,]   14   18
> x*x
     [,1] [,2]
[1,]    4   36
[2,]   16   64
> log10(x)
        [,1]      [,2]
[1,] 0.30103 0.7781513
[2,] 0.60206 0.9030900
```

Note how we can take `x` and add either a scalar (which is added to each element) or another array (which is added element-by-element). The `*` sign multiplies element-by-element. To do a matrix multiplication, you need the special operator `%*%`:

```
> x %*% x
     [,1] [,2]
[1,]   28   60
[2,]   40   88
```

R is a very convenient environment for working with matrices.

- A matrix is displayed the way round you expect
- The indices are the same way round as in mathematical texts
- There are built-in functions to find the inverses, determinants and eigenvectors, to solve sets of simultaneous equations and so on

1.6.4 Lists

For some purposes, arrays are not flexible enough. One might want a variable to contain elements of different types. If, for example, you wanted to keep information about a person in one variable, you might want to record his or her name (a string), age and height (numeric) and whether they are a student or not (logical). You can set up such variables, called lists, like this:

```
> elvis<-list(name="Elvis Aaron Presley",age=64,height=1.7,is.student=FALSE)
> fred<-list(name="Fred Smith",age=21,height=1.8,is.student=TRUE)
```

You can then access the different elements of the lists like this:

```
> elvis$name
[1] "Elvis Aaron Presley"
> fred$age
[1] 21
```

Lists can contain other lists and arrays as elements, too. You can also refer to list elements by number:

```
> elvis[[1]]
[1] "Elvis Aaron Presley"
> elvis[[3]]
[1] 1.7
```

When we read data in with `read.table()` the result is essentially a list.

1.7 What happens when you quit R

You will notice that when you type `q()` to leave R, it asks you whether you want to `Save workspace image? [y/n/c]:` . If you say `y` then R will create a files called .RData and .Rhistory, which contain all of your variables, and a history of the commands you typed. These will be loaded next time you start R. You can safely delete either one, or both, if you do not want them.

And that's it for the first session. You can now make most of the 2-D graphs that you will need. In the next session we'll look at some more sophisticated programming techniques.

1.7.1 Problems

1. File `/home/hcp/wrk/rpms/week1data.txt` contains three columns of data. The columns are labelled. Read the data in and make a graph of voltage and and pressure as a function of time. Use red squares for voltage and blue triangles for pressure, and provide a legend. Make the title of the graph include your name.

2. Take the fast discrete Fourier transforms (FFT) of the voltage and pressure from the above question and plot both real and imaginary parts against frequency. Note that the spacing between points in frequency space is $1/\Delta t$ where Δt is the difference between the first and last time in the above question. Use red lines for the FFT of voltage and blue lines for the FFT of pressure. Use solid lines for the real parts and dashed lines for the imaginary parts.

3. Find the solution to this set of simultaneous equations

$$\begin{aligned} 2x_1 + x_2 &= 1 && (1)\\ x_1 + 3x_2 + x_3 &= -3 && (2)\\ x_2 + 2x_3 &= 1 && (3) \end{aligned}$$

4. Find the eigenvalues and eigenvectors of this matrix:

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

Your answers to these problems should be handed in for assessment. Please hand in the code you wrote, together with graphical output (for Q1 and Q2) and your answers (for the other questions). Remember: The on-line help is your friend.

2 Lecture 2: Programming in R

This lecture covers the nuts and bolts of R programming. There won't be much graphics but we will look at many things which will help you to make good use of R. This is the bit of the course where I assume that you can program in a computer language already. If you don't like the way I present the material, type `help.start()` at the R prompt and hit the link "An introduction to R".

2.1 Flow Control

R has most of the constructs that you would expect for arranging loops, conditions and the like.

2.1.1 The `if` statement

This is used when you want to do something if a condition is true and something else otherwise. The statement looks like this:

`if (` *condition* `)` *statement 1* `else` *statement 2*

This will execute statement 1 if the condition is true and statement 2 if the condition is false. (Note that in these descriptions of what a statement does I use `typewriter font` to show what you actually type and *italics* to indicate where you have to put in your own stuff.) Here is an example if an 'if' statement:

```
if (x < 3) print("x less than 3") else print ("x not less than 3")
```

Notice how the logical operators you need to set up the condition look like the ones in C (and many other languages). Here is a table of relational and boolean operators which you can use in the condition of an if statement.

Purpose	R	C	FORTRAN
Relational Operators			
Equal to	==	==	.EQ.
Not equal to	!=	!=	.NE.
Less than or equal to	<=	<=	.LE.
Less than	<	<	.LT.
Greater than or equal to	>=	>=	.GE.
Greater than	>	>	.GT.
Boolean Operators			
And	&&	&&	.AND.
Not	!	!	.NOT.
Or	||	||	.OR.

If you want statement 1 and / or statement 2 to consist of more than one statement, then the if construct looks like this:

`if (`*condition*`) {`
statement 1a
statement 1b
statement 1c
`} else {`
statement 2a
statement 2b
statement 2c
}

The group of statements between a { and a } are treated as one statement by the `if` and `else`.

2.1.2 The `for` loop

If you have a statement or statements that you want to repeat a number of times, you can use the `for` statement to do so. The `for` statement loops over all elements in a list or vector:

`for (`*variable* `in` *sequence*`)` *statement.1*

Try these examples at the R prompt.

```
> for (i in 1:3) print(i)
```

```
> fnord <- list("Cheese",TRUE,27.5)
> for (i in fnord) print(i)
```

Note the colon (`:`) operator generates a sequence of integers. You can use this in places other than the for loop or array indexing, too. Also note that *statement.1* can be a group of statements inside braces, just as with `if` and `else`.

2.1.3 The `while` loop

If you need a loop for which you don't know in advance how many iterations there will be, you can use the 'while' statement. It works like this:

`while(`*condition*`)` *statement*

Again, if you are using this in a program you can replace *statement* with a group of statements in braces. You can construct *condition* in the same way as for an if statement.

2.1.4 Other flow control statements

R has two statements `break` and `next`: these discontinue normal execution in the middle of a `while` or `for` loop. The `next` statement causes the next pass through the loop to begin at once, while `break` causes a jump to the statement immediately after the end of the loop.

2.2 Array operations

We found out in the first lecture that R allows you to do many operations on whole arrays. Suppose, for example, you wanted to make an array whose elements were the squares of the elements in another array. You could use a for loop, like this:

```
n<-length(x)
xsquared <- array(0,n)
for ( i in 1:n) xsquared[i]<-x[i]*x[i]
```

This is how you would *have* to go about it in C or Fortran 77. In R, however, you can do this:

```
xsquared <- x*x
```

This will clearly make your code shorter and clearer but it will also make it *much* faster. This is true for matrix multiplication, too.

2.3 Defining functions

Our examples so far have been too short to break up into sub-programs. However, any sizeable R project will be easier to deal with if each specific task is put into its own sub-program or *function*[6], which your main program can call.

Here is an example of a function. This calculates $\sin(x)/x$. For $|x| \gg 0$ this can be calculated directly using R's built-in function `sin()`, but for values near 0, we would be calculating 0/0, or something very close to it, so we use the power series:

$$\frac{\sin(x)}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} + \cdots$$

Note the use of `if`, `for` and `break`.

```
sinc  <- function(x){
  if (abs(x) > 1) {
    ## x not near 0: calculate in the obvious way.
    s <- sin(x)/x
  }else{
    ## x too close to 0 for sin(x) / x to work. Use power series instead.
    s <- 1
    term <- 1
    for( j in seq(3,100,by=2) ){
```

[6]Like C, R has only one sort of sub-program. Some languages, notably Fortran, have both functions which return a value and subroutines which do not.

```
      term <- term*(-x*x)/(j*(j-1))
      s <- s+term
      if(abs(term) < 1.e-10) break
    }
  }
  ## Value returned is value of last expression: s in this case
  s
}
```

To use this, put the text into a file and `source` it. Nothing appears to happen, but your function has been added to the large set already available to you, so you can now type:

```
> sinc(0.01)
```

then the value of $\sin(0.01)/0.01$ will be printed out.

To summarise, we have learned about R's programming features and that we should use array operations instead of loops where that is possible. In the next lecture we will learn how to display two-dimensional data sets and how to make use of colour in R.

2.4 Problems

Please indent your code tidily: it makes it much easier for me to understand what it does. And please put the `}` at the end of a block on a line by itelf.

1. Write a program to calculate all the prime numbers between 3 and 500. You can use any algorithm you like. You will need to know that the operator `%%` is used to give the remainder from an integer division. Ensure that the program is all in one file, so that you can type
   `source("myprimesprogram.R")`
   to make it do the calculations. If p_j is the j-th prime number, make a plot of $p_{j+1} - p_j$ as a function of j.

2. Write a function which takes a vector `x` as its only argument and returns a list containing the mean and standard deviation of the values in `x`. Your function should do all the calculation itself, using loops and standard arithmetic operators. It should not use any of R's built-in functions. (As an exception, you can use `length(x)` to get the number of elements in `x`.) I have made a file containing 1000 random numbers, which you can read into R like this:

   ```
   x<-scan("/home/hcp/wrk/rpms/randomdata1.txt")
   ```

   Use your function to calculate the mean and standard deviation of these numbers. Compare your results with those returned by R's built-in functions `mean` and `sd`. (Do remember to hand in the results as well as your program.) For those of you not doing Inverse Theory, for a list of n numbers x_i, the mean $\bar{x}$ is given by

   $$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

   and the variance by

   $$\mathrm{var}(x) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

   The standard deviation is the square root of the variance.

3 Lecture 3: More graphics

In this lecture we will learn how to display data that are a function of two variables. An example would be the height of the ground above sea level as a function of distance East and distance North. We'll also look at a few of R's more esoteric graphics features. There are no figures in these notes – if you want to see what the plots look like, start R up and run the examples. We will construct an example data set using this short program:

```
## Set up vectors of x and y  values
x <- seq(-2,2,by=0.05)
y <- seq(-2,2,by=0.1)

## Define a function of x and y
myfn <- function(x,y){exp(-(x^2 +y^2)) + 0.6*exp( -((x+1.8)^2 + y^2)) }

## use outer()  to apply that function to all our x and y values
z <- outer(x,y,myfn)
```

If you type this program into a file called setup2d.R (or, if you are looking at the HTML version of these notes with a web browser, cut and paste it) and then type:

```
> source("setup2d.R")
```

you will be left with two vectors called x and y and a 2-d array called z. You can imagine that x is distance East, y is distance North and z is height above sea level. We will investigate several ways of displaying these data.

3.1 Surface plots

Perhaps the most immediate way of displaying a data set like our example is as a surface plot. Type

```
> persp(x,y,z,theta=-40,phi=30)
```

You can get more informative axis labels like this:

```
> persp(x,y,z,theta=-40,phi=30,xlab="East",ylab="North",zlab="Height",ticktype="detailed")
```

Like many R functions, persp has lots of keywords to enable you to tweak the plot to look how you want it to look. Note particularly the theta and phi keywords, which let you look at the plot from different angles. Try these examples:

```
> persp(x,y,z,theta=40,phi=60)
> persp(x,y,z,theta=10,phi=10)
```

An alternative to the wire frame is a shaded surface, which you can get with the `shade` keyword:

```
> persp(x,y,z,theta=-70,phi=50,shade=0.7)
```

If you don't want the grid lines, you can remove them:

```
> persp(x,y,z,theta=-70,phi=50,shade=0.7,border=NA)
```

3.2 Contour Plots

It is less visually striking, but sometimes more useful, to display 2-d data as a contour plot. For a basic contour plot, do:

```
> contour(x,y,z)
```

It's a start, but we might not like the contour levels chosen. We can specify them by hand:

```
> contour(x,y,z,lev=seq(0,1,by=0.05))
```

Note that `seq(0,1,by=0.05)` returns a vector with the elements set to $0, 0.05, 0.1, 0.15, \cdots, 1.0$. You can use any vector as long as the values increase monotonically.

Plain contours are not very striking – it looks more eye-catching if we fill the spaces between the contours with colours, like this:

```
> filled.contour(x,y,z)
```

That makes the general form of the data clearer as you can see immediately where the highs and lows are. Those colours are a bit nasty, though. There are better choices available:

```
> filled.contour(x,y,z,color.palette=terrain.colors)
```

If you want to see your array without it being contoured, try this:

```
image(x,y,z)
```

The `image` function doesn't provide a scale bar. But you can overlay some contours:

```
> contour(x,y,z,add=TRUE)
```

3.3 Colours in R

We have already seen that colours in R can be specified by name, and also that some functions make use of some of R's built-in colour schemes. Individual colours can also be specified by giving the brightness of red, green and blue on a scale from 0 to 255. These are given as a # symbol and a string of 6 hexadecimal digits, so that `#ff0000` is red, `#00ff00` is green, `#ffff00` is yellow, `#0000ff` is blue and so on. You can convert these rgb strings into numbers with `col2rgb` and generate them with `rgb`. Try these examples:

```
> col2rgb("orange")
> col2rgb("#ffff00")
> rgb(1,0.5,0)
```

If you don't like R's built-in colorscales, you can use `colorRampPalette` to make your own:

```
>  filled.contour(x,y,z,color.palette=colorRampPalette(c("black","green","white")))
```

3.4 Maps in R

R does not have a built-in capacity to put data on a map of the world. Like many of R's non-core features, mapping is provided by external packages[7] To use this, start R and type

```
> library(maps)
> library(mapproj)
```

Nothing will appear to happen[8] Try out these examples:

```
> map()
> map(proj="mercator",xlim=c(-178,178),ylim=c(-70,70)
> map(proj="aitoff",xlim=c(-178,178))
> map.grid()
> map(proj="orthographic",orientation=c(55,0,0))
> map.grid()
```

To get rid of the white space around the map, you may want to do

```
> par(mai=c(0,0,0,0))
```

before making the plots: this sets all the margins to 0 inches. To add data to your map, you need to convert its latitude and longitude to the projected map co-ordinates:

```
longs<-c(0, -3.1, 2.4, -3, -77)
lats<-c(51.5, 55.9, 48.8, 17, 38.8)
labs<-c("London","Edinburgh","Paris","Timbuktu","Washington")
points(mapproject(longs,lats),col="red",pch=19)
text(mapproject(longs,lats),labs,col="red",adj=-0.1)
```

Note that `mapproject` converts your lat and long to the correct x and y co-ordinates for the last map you drew. Also, don't forget that the on-line documentation works for add-on packages like those used for mapping, so you can find out more by typing

```
> help(map)
```

[7] Packages for R may be found via a network of mirrored web sites called CRAN (see CRAN link at www.r-project.org).

[8] If you get the message `Error in library(maps) :  there is no package called 'maps'` then you need to ask your system administrators to install the packages, or learn how to do this yourself. You don't need to be root, but it is outside the scope of this introduction. If the packages are installed in a non-standard location, you will need to tell R where they are. Do this by typing `export R_LIBS=/my/rlibs/directory` before you start R .

3.5 Images in R

To most computer systems a grayscale image is just a 2-D data set such as we looked at earlier. The differences between images and other data sets are simply that:

- Images are (usually) bigger, perhaps several hundred data points in each direction.
- Images are (usually) of type byte. This means that each pixel can have only 256 values. Remote-sensed images may have more bits per pixel than this, though.

. . . and that's all. You can display and process greyscale images just like any other 2-dimensional array. You will find that contour or surface plots of an image of any size are not very useful and take a very long time to draw. In this section we look at ways of reading, displaying and processing images in R. As with maps, R's image-handling capabilities are provided by add-on packages, so you need to type

```
> library(pixmap)
> library(imageio)
```

The pixmap package provides a way to read in the portable pixmap image format, while the imageio package adds on the ability to read many other image formats[9].

3.6 Greyscale images

An image can be read in like this[10]:

```
> im<-read.image("/home/hcp/wrk/rpms/light-g.png",fformat="png")
```

Here, im is not a simple 2-D array, but is a more complex object, containing information about the image, whether it is greyscale or colour, what size it is, and so forth. This means that you can display the image like this:

```
plot(im)
```

. . . but if you want to get the data out and process them, you need to do this:

```
imdat<-getChannels(im)
```

The array imdat is now a perfectly ordinary 2-d array: you can try drawing a contour plot of it if you like. You will notice that if you try to plot it using `image` or `contour` that the axes are reversed. This is a consequence of the order in which the data are traditionally arranged in image files. You may also notice that the data lie in the range between 0 and 1: the pixmap package always scales images so that 0 means dark and 1 means bright.

We can do some mathematical operations on the image, plot the result and write the result to a file, like this:

```
> d<-dim(imdat)
> newdat<-1-imdat + imdat[,d[2]:1]
> newim<-pixmapGrey(newdat)
> plot(newim)
> write.image(newim,file="newim.png",fformat="png")
```

What is the result? Was it what you expected? Note that when you convert your data to a pixmap object with `pixmapGrey` the data are automatically scaled to the range $(0, 1)$. Also note that R's image plotting is rather slow for large images. In many cases it is more convenient to write the image out with `write.image` and view it with a separate image viewing program such as `gthumb`, or the GNOME image viewer `eog`.

3.7 The rimage package

Some additional image-processing tools are provided in the rimage package. Rather annoyingly, this uses its own sort of image object, (an *imagematrix*) which is slightly different from the one used by pixmap and imageio. An *imagematrix* object can be regarded as an ordinary matrix for many purposes. Like pixmap, rimage scales its data to lie between 0 and 1. Try these examples: they demonstrate several of the functions provided by the rimage package.

[9] I wrote the imageio package myself – I am in discussions with the maintainers of the pixmap package about adding all of the imageio functions to the pixmap package

[10] If you omit `fformat="png"` then `read.image` will try to guess whether your image is a tiff, a png, a jpeg etc. This feature is not reliable on Solaris or Windows, though.

```
> library(rimage)
> flarp<-sobel(imdat)
> plot(normalize(flarp))
> flarp<-laplacian(imdat)
> plot(normalize(flarp))
> flarp<-clipping(imdat,low=0.45,high=0.55)
> plot(pixmapGrey(normalize(flarp)))
> flarp<-equalize(imdat)
> plot(normalize(flarp))
> flarp<-fftImg(imdat)
> plot(normalize(flarp))
```

Note in particular the function `fftimg()` which calculates the Fourier transform of the image.

3.8 Colour images

A colour image is really three images, a red one, a green one and a blue one. If you want to do image processing things to a colour image, smoothing for example, then you have to apply your algorithm to the three colours separately. In R, you can read a colour image like this:

```
> imc<-read.image("/home/hcp/wrk/rpms/light.tif")
```

As before, imc is a special pixmap object: you need to use `getChannels` to extract the data from it. What you get, for an image of size $m \times n$ is an array of dimensions $m \times n \times 3$. Try this example, which swaps around the red, green and blue planes of the image.

```
imcdat<-getChannels(imc)
newdat<-array(0,dim(imcdat))
newdat[,,1]<-imcdat[,,2]
newdat[,,2]<-imcdat[,,3]
newdat[,,3]<-imcdat[,,1]
newim<-pixmapRGB(newdat)
plot(newim)
```

3.9 Problems

- Make both a surface plot and a contour plot of this function:

$$f(x, y) = \cos(2x) \sin(y) \exp(-0.2(x^2 + y^2))$$

  Your plots should cover the range $-\pi \leq x \leq \pi, \quad -\pi \leq y \leq \pi$. How many maxima and minima does the function have in this range?

- The EOS MLS instrument makes measurements at 3495 positions spread over the globe every day. The latitudes and longitudes of these positions for a particular day can be read in like this:

  ```
  flarp <- read.table("/home/hcp/wrk/rpms/latlong.dat")
  ```

  Plot these positions on a map of the world, using (a) the Mollweide projection and (b) the stereographic projection. For (b) your map should cover latitudes between 90°S and 30°S.

- Read in the greyscale image in the file `/home/hcp/wrk/rpms/light2.png` and display it. The original image pixels had values between 0 and 255. Make a new image which has black pixels where the original pixel value was an odd number, and white pixels where the original pixel value was an even number. Hint: try to avoid writing any loops: the `which` function will help.

4 Hints, suggestions etc

There is only so much that can be fitted into the assessed part of the course. In an attempt to make these notes a more useful reference, I am collecting here various things which have proved useful in the past to me or to others.

4.1 Output in various forms

To get encapsulated postscript output for including in a MicroSoft Word, OpenOffice Writer or LaTeX document:

```
postscript(file="myplot.eps",width=6,height=5,paper="special",onefile=FALSE,
           family="Times",horizontal=FALSE,pointsize=14)
myplot() ## Replace this with all your plotting commands
dev.off()
```

You can tinker with the width, height and pointsize arguments to get the plot to look right. The `family="Times"` bit will make the axis labels come out in a serif font, which will probably match the one in the body of your document. The `paper="special"` bit is necessary to prevent R worrying about whether your width and height values are too big for your paper.

If you have an application that can import PDF figures, you might want to generate these with R. (The only such application that I have used is `pdflatex`.) The procedure is almost identical to that for postscript files:

```
pdf(file="myplot.pdf",width=6,height=5,paper="special",onefile=FALSE,
           family="Times",horizontal=FALSE,pointsize=14)
myplot() ## Replace this with all your plotting commands
dev.off()
```

If you want to be able to annotate your figure by hand after you have made it, you need to be able to import it into a vector drawing program. An old-fashioned (but widely available) program of this type is `xfig`. You can get R to produce figures in xfig's native format like this:

```
xfig(file="myplot.fig",width=6,height=5,onefile=TRUE,
           family="Times",horizontal=FALSE,pointsize=14)
myplot() ## Replace this with all your plotting commands
dev.off()
```

Xfig can export your figure in a variety of formats that R cannot generate, such as enhanced metafile (`.emf`) and scalable vector graphics[11] (`.svg`). If you really need such a format and do not want to annotate your figure, you can use the `fig2dev` command to convert xfig output without having to start xfig:

```
bash$ fig2dev -L emf myplot.fig myplot.emf
bash$ fig2dev -L svg myplot.fig myplot.svg
```

If you want your figure as a bitmapped image, R can do this for you. Typically you would do this to include the plot in a web page or a presentation slide.

```
png(file="myplot.png",width=600,height=500,pointsize=14)
myplot()## Replace this with all your plotting commands
dev.off()
jpeg(file="myplot.jpg",width=600,height=500,pointsize=14,quality=95)
myplot()## Replace this with all your plotting commands
dev.off()
```

For most sorts of R plot, png is a more suitable choice than jpeg. Note that the size is given in pixels. If you find that you need to stretch or shrink the image once you get it into another application, you will probably want to re-run your R program with different width and height settings, to re-generate the image at the right size.

4.2 Fancy maths in text labels

R can put plain text on a plot in a number of ways. You can use the `xlab`, `ylab`, and `main` keywords to `plot`, and you can also use the `text` function to put text in an arbitrary position. If you want mathematical

[11] SVG is an open standard for vector graphics and is growing in popularity. There is an SVG device for R under development, but it is lacking in many features at the time of writing.

formulae in any of these places, you can use the `expression` function instead of a text string. For example, instead of `"x squared"` you can use `expression(x^2)` and get x^2. To get the online docs for this feature, type `help(plotmath)` and for a demo of the possibilities try `demo(plotmath)` One thing which may not be obvious is how to mix text and mathematics. The answer seems to be

```
expression(paste("The area,",x^2,",is too large"))
```

4.3 Two lines and y-axes on one plot

Suppose you have two sets of values which are both functions of the same variable, but have different units and wildly different values. Maybe you measured Ozone (in parts per million) and Temperature (in K) at the same point in the atmosphere at 12:00 every day for a year. How can you get the two sets of data onto the same graph, with a different axis for each? The hidden magic is to use `par(new=TRUE)` to prevent the second plot erasing the first one, and to do the axes for the second plot by hand. Here is a complete example:

```
### Generate the data to plot
tm<-seq(0,4*pi,by=pi/20) ## The times
x1<-cos(tm)              ## the first data set
x2<-117*exp(-0.1*tm)*sin(tm) + 33     ## the second data set


### ... and plot them
plot(tm,x1,type="l",frame.plot=FALSE) ### the first dataset
par(new=TRUE) ### The magic to stop the second plot erasing the first
plot(tm,x2,type="l",frame.plot=FALSE,axes=FALSE,col="red") ## the second
axis(side=4,col="red") ### The axis for the second dataset
```

4.4 Dots with a colour scale

Often, one has a set of points in which each point has cartesian co-ordinates (x, y) and a value z to be plotted. Perhaps the simplest way to display the data is to put a point at each (x, y) position whose colour represents the z value. Here is a full example that generates a dataset, plots it, and adds a colour bar:

```
## Generate a test set of data
npts <- 1000
x <- runif(npts,min=0,max=10)
y <- runif(npts,min=0,max=10)
z <- sin(x)*cos(y) + rnorm(npts,mean=0,sd=0.01)

## Choose the z-values at which the colour will change
levs <- pretty(z,n=20)

## Work out which colour level each point is at
nl <- length(levs)
icols <- as.integer(1 +  (nl-1) * (z-levs[1])/(levs[nl]-levs[1]))
mypalette <- rainbow(nl-1)
palette(mypalette)
## Divide up the plot so we can have a colour bar
m <- matrix(c(1,2),nrow=1,ncol=2)
layout(m,widths=c(1,0.25))

## Plot the data
plot(x,y,pch=19,col=icols)

## Make the colour bar
plot(c(0,1),c(0,1),type="n",xlim = c(0, 1), ylim = range(levs), xaxt = "n",
            yaxs = "i",xlab="",ylab="",axes={axis(2)})
rect(0, levs[-nl], 1, levs[-1], col = mypalette)
```

4.5 Running R on Windows

R runs well on all modern flavours of Unix, where it is essentially identical to R on Linux. It also runs on MacOSX and Windows. I can't offer much information about the MacOSX port as I don't have a Mac. [12] On the other hand, I have access to plenty of University managed-desktop machines on which R is installed. Most of the experience on which these notes are based was gained on the managed desktop, although I do have R installed on a Windows XP machine at home. Most aspects of R are the same on Windows as on other platforms. In this section I not a few of the important differences.

4.5.1 The R GUI

On Linux, most users run R in a terminal window (although there is at least one GUI in development.) On Windows, R is usually started from the start menu, presenting the user with a GUI. The GUI takes up the whole screen by default. An R command line and any graphics windows that you start live inside this big R window. If you don't like this MDI interface, you can switch to an SDI one where the R windows live directly on your desktop. The button to do this is under `Edit --> GUI Preferences`.

4.5.2 Setting the directory

One of the initially puzzling things about R on Windows is that the working folder used when R starts up may not be your home directory. Instead, it may turn out to be the system directory into which R was installed and for which you probably do not have write permission. You can find out what the current working directory is by typing `getwd()` and set it to be a directory of your choice by doing `setwd("my/working/folder")`. For example, you can set the working directory to be your home directory like this: `setwd{"M:/"}`. Note that you will want to use forward slashes to separate folders, even though this is Windows.

4.5.3 Getting more help

One of the menu items under `Help` is the R FAQ for Windows. This covers many of the obscure issues to do with using R on Windows — it should be the first thing you turn to if you have any Windows-specific R problems.

[12] But I do know that R is a properly integrated (native quartz) OSX application. Unlike many free programs it does not require you to run an X server.