Chapter 2
R Literacy for Digital Soil Mapping

2.1 Objective

The immediate objective here is to skill up in data analytics and basic graphics with R. The range of analysis that can be completed, and the types of graphics that can be created in R is simply astounding. In addition to the wide variety of functions available in the "base" packages that are installed with R, more than 4500 contributed packages are available for download, each with its own suite of functions. Some individual packages are the subject of entire books.

For this chapter of the book and the later chapters that will deal with digital soil mapping exercises, we will not be able to cover every type of analysis or plot that R can be used for, or even every subtlety associated with each function covered in this entire book. Given it's inherent flexibility, R is difficult to master, as one may be able to do with a stand-alone software. R is a software package one can only increase their knowledge and fluency in. Meaning that, effectively, learning R is a boundless pursuit of knowledge.

In a disclaimer of sorts, this introduction to R borrows many ideas, and structures from the plethora of online materials that are freely available on the internet. It will be worth your while to do a Google search from time-to-time if you get stuck—you will be amazed to find how many other R users have had the same problems you have or have had.

2.2 Introduction to R

2.2.1 R Overview and History

R is a software system for computations and graphics. According to the R FAQ (http://cran.r-project.org/doc/FAQ/R-FAQ.html#R-Basics):

B.P. Malone et al., *Using R for Digital Soil Mapping*,
Progress in Soil Science, DOI 10.1007/978-3-319-44327-0_2

> It consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files.

R was originally developed in 1992 by Ross Ihaka and Robert Gentleman at the University of Auckland (New Zealand). The R language is a dialect of the S language which was developed by John Chambers at Bell Laboratories. This software is currently maintained by the R Development Core Team, which consists of more than a dozen people, and includes Ihaka, Gentleman, and Chambers. Additionally, many other people have contributed code to R since it was first released. The source code for R is available under the GNU General Public Licence, meaning that users can modify, copy, and redistribute the software or derivatives, as long as the modified source code is made available. R is regularly updated, however, changes are usually not major.

2.2.2 Finding and Installing R

R is available for Windows, Mac, and Linux operating systems. Installation files and instructions can be downloaded from the Comprehensive R Archive Network (CRAN) site at http://cran.r-project.org/. Although the graphical user interface (GUI) differs slightly across systems, the R commands do not.

2.2.3 Running R: GUI and Scripts

There are two basic ways to use R on your machine: through the GUI, where R evaluates your code and returns results as you work, or by writing, saving, and then running R script files. R script files (or scripts) are just text files that contain the same types of R commands that you can submit to the GUI. Scripts can be submitted to R using the Windows command prompt, other shells, batch files, or the R GUI. All the code covered in this book is or is able to be saved in a script file, which then can be submitted to R. Working directly in the R GUI is great for the early stages of code development, where much experimentation and trial-and-error occurs. For any code that you want to save, rerun, and modify, you should consider working with R scripts.

So, how do you work with scripts? Any simple text editor works—you just need to save text in the ASCII format i.e., "unformatted" text. You can save your scripts and either call them up using the command `source ("file_name.R")` in the R GUI, or, if you are using a shell (e.g., Windows command prompt) then type `R CMD BATCH file_name.R`. The Windows and Mac versions of the R GUI comes with a basic script editor, shown below in Fig. 2.1.

Unfortunately, this editor is not very good by reason that the Windows version does not have syntax highlighting.

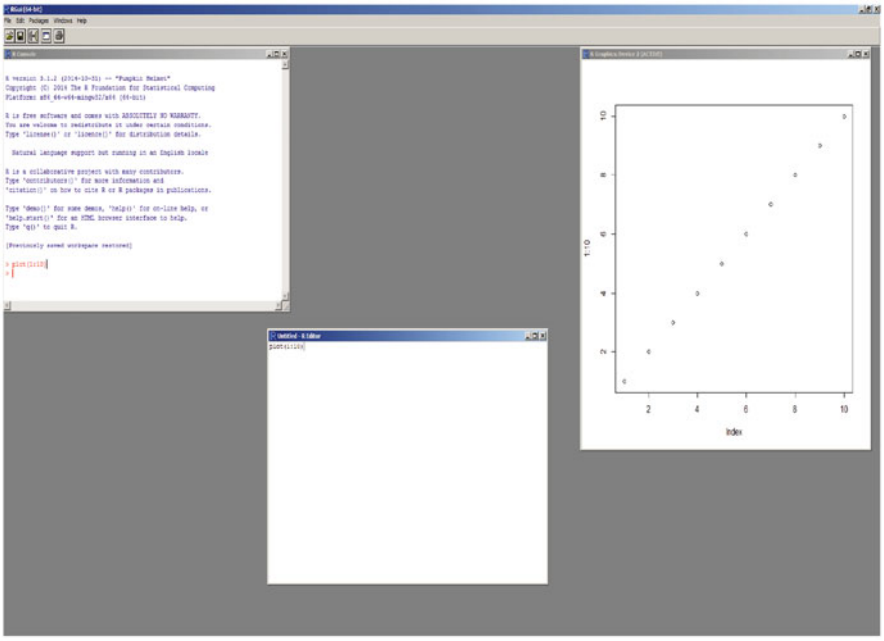

Fig. 2.1 R GUI, its basic script editor, and plot window

There are some useful (in most cases, free) text editors available that can be set up with R syntax highlighting and other features. TINN-R is a free text editor http://nbcgib.uesc.br/lec/software/des/editores/tinn-r/en that is designed specifically for working with R script files. Notepad++ is a general purpose text editor, but includes syntax highlighting and the ability to send code directly to R with the NppToR plugin. A list of text editors that work well with R can be found at: http://www.sciviews.org/_rgui/projects/Editors.html.

2.2.4 RStudio

RStudio http://www.rstudio.com/ is an integrated development environment (IDE) for R that runs on Linux, Windows and Mac OS X. We will be using this IDE during the book, generally because it is very well designed, intuitively organized, and quite stable.

When you first launch RStudio, you will be greeted by an interface that will look similar to that in Fig. 2.2.

The frame on the upper right contains the *workspace* (where you will be able see all your R objects), as well of a history of the commands that you have previously entered. Any plots that you generate will show up in the region in the lower right

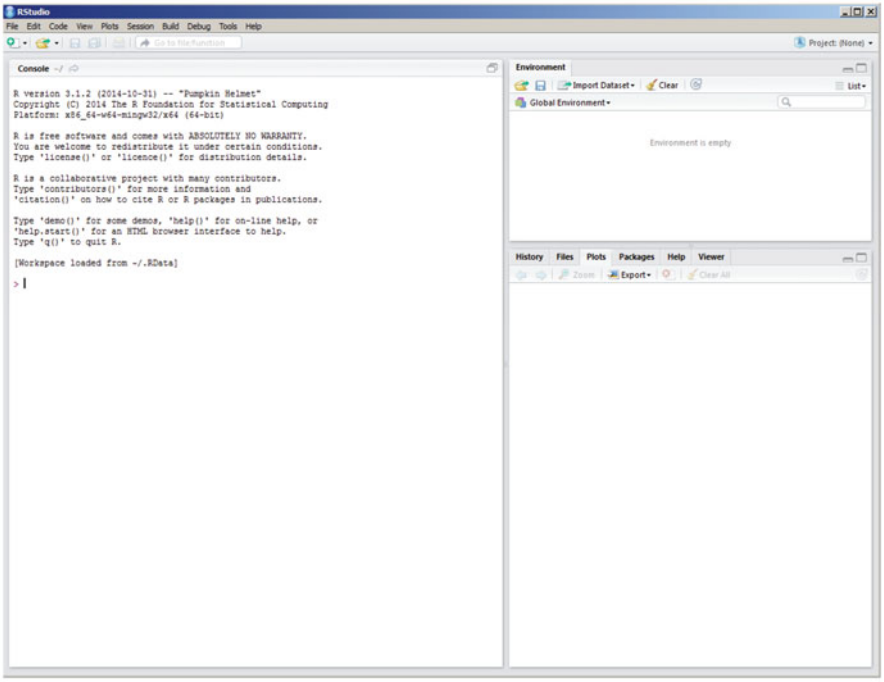

Fig. 2.2 The RStudio IDE

corner. Also in this region is various help documentation, plus information and documentation regarding what packages and function are currently available to use.

The frame on the left is where the action happens. This is the R console. Every time you launch RStudio, it will have the same text at the top of the console telling you the version that is being used. Below that information is the prompt. As the name suggests, this is where you enter commands into R. So lets enter some commands.

2.2.5 R Basics: Commands, Expressions, Assignments, Operators, Objects

Before we start anything, it is good to get into the habit of making scripts of our work. With RStudio launched go the *File* menu, then *new*, and *R Script*. A new blank window will open on the top left panel. Here you can enter your R prompts. For example, type the following: `1+1`. Now roll your pointer over the top of the panel to the right pointing green arrow (first one), which is a button for running the

line of code down to the R console. Click this button and R will evaluate it. In the console you should see something like the following:

```
1 + 1
```

```
## [1] 2
```

You could have just entered the command directly into the prompt and gotten the same result. Try it now for yourself. You will notice a couple of things about this code. The > character is the prompt that will always be present in the GUI. The line following the command starts with a `[1]`, which is simply the position of the adjacent element in the output—this will make some sense later.

For the above command, the result is printed to the screen and lost—there is no assignment involved. In order to do anything other than the simplest analyses, you must be able to store and recall data. In R, you can assign the results of commands to symbolic variables (as in other computer languages) using the assignment operator `<-`. Note that other computer scripting languages often use the equals sign (=) as the assignment operator. When a command is used for assignment, the result is no longer printed to the GUI console.

```
x <- 1 + 1
x
```

```
## [1] 2
```

Note that this is very different from:

```
x < -1 + 1
```

```
## [1] FALSE
```

In this case, putting a space between the two characters that make up the assignment operator causes R to interpret the command as an expression that ask if x is less than zero. However spaces usually do not matter in R, as long as they do not separate a single operator or a variable name. This, for example, is fine:

```
x <- 1
```

Note that you can recall a previous command in the R GUI by hitting the up arrow on your keyboard. This becomes handy when you are debugging code.

When you give R an assignment, such as the one above, the object referred to as x is stored into the R workspace. You can see what is stored in the workspace by looking to the workspace panel in RStudio (top right panel). Alternatively, you can use the `ls` function.

```
ls()
```

```
## [1] "x"
```

To remove objects from your workspace, use `rm`.

```
rm(x)
x
```

As you can see, You will get an error if you try to evaluate what `x` is.

If you want to assign the same value to several symbolic variables, you can use the following syntax.

```
x <- y <- z <- 1
ls()
```

```
## [1] "x" "y" "z"
```

`R` is a case-sensitive language. This is true for symbolic variable names, function names, and everything else in `R`.

```
x <- 1 + 1
x
X
```

In `R`, commands can be separated by moving onto a new line (i.e., hitting enter) or by typing a semicolon (;), which can be handy in scripts for condensing code. If a command is not completed in one line (by design or error), the typical `R` prompt > is replaced with a +.

```
x<-
+  1+1
```

There are several operators that are used in the `R` language. Some of the more common are listed below.

Arithmetic

`+` `-` `*` `/` `^` plus, minus, multiply, divide, power

Relational

`a == b`	a is equal to b (do not confuse with =)
`a != b`	a is not equal to b
`a < b`	a is less than b
`a > b`	a is greater than b
`a <= b`	a is less than or equal to b
`a >= b`	a is greater than or equal to b

Logical/grouping

`!`	not
`&`	and
`|`	or

Indexing

`$`	part of a data frame
`[]`	part of a data frame, array, list
`[[]]`	part of a list

Grouping commands

`{}`	specifying a function, for loop, if statement etc.

Making sequences

`a:b`	returns the sequence a, a+1, a+2, ... b

Others

`#`	commenting (very very useful!)
`;`	alternative for separating commands
`~`	model formula specification
`()`	order of operations, function arguments

Commands in R operate on objects, which can be thought of as anything that can be assigned to a symbolic variable. Objects include vectors, matrices, factors, lists, data frames, and functions. Excluding functions, these objects are also referred to as data structures or data objects.

When you want to finish up on an R session, RSudio will ask you if you want to "save workspace image". This refers to the workspace that you have created , i.e., all the objects you have created or even loaded. It is generally good practice to save your workspace after each session. More importantly however, is the need to save all the commands that you have created on your script file. Saving a script file in Rstudio is just like saving a Word document. Give both a go—save the script file and then save the workspace. You can then close RStudio.

2.2.6 *R Data Types*

The term "data type" refers to the type of data that is present in a data structure, and does not describe the data structure itself. There are four common types of data in R: numerical, character, logical, and complex numbers. These are referred to as *modes* and are shown below:
Numerical data

```
x <- 10.2
x

## [1] 10.2
```

Character data

```
name <- "John Doe"
name
```

```
## [1] "John Doe"
```

Any time character data are entered in the R GUI, you must surround individual elements with quotes. Otherwise, R will look for an object.

```
name <- John
```

```
## Error in eval(expr, envir, enclos): object 'John' not found
```

Either single or double quotes can be used in R. When character data are read into R from a file, the quotes are not necessary.

Logical data contain only three values: TRUE, FALSE, or NA, (NA indicates a missing value—more on this later). R will also recognize T and F, (for true and false respectively), but these are not reserved, and can therefore be overwritten by the user, and it is therefore good to avoid using these shortened terms.

```
a <- TRUE
a
```

```
## [1] TRUE
```

Note that there are no quotes around the logical values—this would make them character data. R will return logical data for any relational expression submitted to it.

```
4 < 2
```

```
## [1] FALSE
```

or

```
b <- 4 < 2
b
```

```
## [1] FALSE
```

And finally, complex numbers, which will not be covered in this book, are the final data type in R

```
cnum1 <- 10 + (0+3i)
cnum1
```

```
## [1] 10+3i
```

You can use the mode or class function to see what type of data is stored in any symbolic variable.

```
class(name)

## [1] "character"

class(a)

## [1] "logical"

class(x)

## [1] "numeric"

mode(x)

## [1] "numeric"
```

2.2.7 *R Data Structures*

Data in R are stored in data structures (also known as data objects)—these are and will be the that you perform calculations on, plot data from, etc. Data structures in R include vectors, matrices, arrays, data frames, lists, and factors. In a following section we will learn how to make use of these different data structures. The examples below simply give you an idea of their structure.

Vectors are perhaps the most important type of data structure in R. A vector is simply an ordered collection of elements (e.g., individual numbers).

```
x <- 1:12
x

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Matrices are similar to vectors, but have two dimensions.

```
X <- matrix(1:12, nrow = 3)
X

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

Arrays are similar to matrices, but can have more than two dimensions.

```
Y <- array(1:30, dim = c(2, 5, 3))
Y

## , , 1
##
##      [,1] [,2] [,3] [,4] [,5]
```

```
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
##
## , , 2
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   11   13   15   17   19
## [2,]   12   14   16   18   20
##
## , , 3
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   21   23   25   27   29
## [2,]   22   24   26   28   30
```

One feature that is shared for vectors, matrices, and arrays is that they can only store one type of data at once, be it numerical, character, or logical. Technically speaking, these data structures can only contain elements of the same mode.

Data frames are similar to matrices—they are two-dimensional. However, a data frame can contain columns with different modes. Data frames are similar to data sets used in other statistical programs: each column represents some variable, and each row usually represents an "observation", or "record", or "experimental unit".

```
dat <- (data.frame(profile_id = c("Chromosol", "Vertosol", "Sodosol"),
FID = c("a1", "a10", "a11"), easting = c(337859, 344059, 347034),
northing = c(6372415, 6376715, 6372740), visited = c(TRUE, FALSE, TRUE)))

dat
```

```
##   profile_id FID easting northing visited
## 1  Chromosol  a1  337859  6372415    TRUE
## 2   Vertosol a10  344059  6376715   FALSE
## 3    Sodosol a11  347034  6372740    TRUE
```

Lists are similar to vectors, in that they are an ordered collection of elements, but with lists, the elements can be other data objects (the elements can even be other lists). Lists are important in the output from many different functions. In the code below, the variables defined above are used to form a list.

```
summary.1 <- list(1.2, x, Y, dat)
summary.1
```

```
## [[1]]
## [1] 1.2
##
## [[2]]
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
##
## [[3]]
## , , 1
##
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
##
## , , 2
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   11   13   15   17   19
## [2,]   12   14   16   18   20
##
## , , 3
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   21   23   25   27   29
## [2,]   22   24   26   28   30
##
##
## [[4]]
##   profile_id FID easting northing visited
## 1  Chromosol  a1  337859  6372415    TRUE
## 2   Vertosol a10  344059  6376715   FALSE
## 3    Sodosol a11  347034  6372740    TRUE
```

Note that a particular data structure need not contain data to exist. This may seem unusual, but it can be useful when it is necessary to set up an object for holding some data later on.

```
x <- NULL
```

2.2.8 *Missing, Indefinite, and Infinite Values*

Real data sets often contain missing values. R uses the marker NA (for "not available") to indicate a missing value. Any operation carried out on an NA will return NA.

```
x <- NA
x - 2
```

```
## [1] NA
```

Note that the NA used in R does not have the quotes around it—this would make it character data. To determine if a value is missing, use the `is.na`—this function can also be used to set elements in a data object to NA.

```
is.na(x)
```

```
## [1] TRUE
```

```
!is.na(x)
```

```
## [1] FALSE
```

Indefinite values are indicated with the marker `NaN`, for "not a number". Infinite values are indicated with the markers `Inf` or `-Inf`. You can find these values with the functions `is.infinite, is.finite`, and `is.nan`.

2.2.9 *Functions, Arguments, and Packages*

In `R`, you can carry out complicated and tedious procedures using functions. Functions require arguments, which include the object(s) that the function should act upon. For example, the function `sum` will calculate the sum of all of its arguments.

```
sum(1, 12.5, 3.33, 5, 88)
```

```
## [1] 109.83
```

The arguments in (most) `R` functions can be named, i.e., by typing the name of the argument, an equal sign, and the argument value (arguments specified in this way are also called tagged). For example, for the function `plot`, the help file lists the following arguments.

```
plot (x, y,...)
```

Therefore, we can call up this function with the following code.

```
a <- 1:10
b <- a
plot(x = a, y = b)
```

With names arguments, `R` recognizes the argument keyword (e.g., `x` or `y`) and assigns the given object (e.g., `a` or `b` above) to the correct argument. When using names arguments, the order of the arguments does not matter. We can also use what are called positional arguments, where `R` determines the meaning of the arguments based on their position.

```
plot(a, b)
```

This code does the same as the previous code. The expected position of arguments can be found in the help file for the function you are working with or by asking `R` to list the arguments using the `args` function.

```
args(plot)
```

```
## function (x, y, ...)
## NULL
```

It usually makes sense to use the positional arguments for only the first few arguments in a function. After that, named arguments are easier to keep track of. Many functions also have default argument values that will be used if values are not specified in the function call. These default argument values can be seen by using the `args` function and can also be found in the help files. For example, for the function `rnorm`, the arguments `mean` and `sd` have default values.

```
args(rnorm)
```

```
## function (n, mean = 0, sd = 1)
## NULL
```

Any time you want to call up a function, you must include parentheses after it, even if you are not specifying any arguments. If you do not include parentheses, `R` will return the function code (which at times might actually be useful).

Note that it is not necessary to use explicit numerical values as function arguments—symbolic variable names which represent appropriate data structure can be used. it is also possible to use functions as arguments within functions. `R` will evaluate such expressions from the inside outward. While this may seem trivial, this quality makes `R` very flexible. There is no explicit limit to the degree of nesting that can be used. You could use:

```
plot(rnorm(10, sqrt(mean(c(1:5, 7, 1, 8, sum(8.4, 1.2, 7))))), 1:10)
```

The above code includes 5 levels of nesting (the sum of 8.4,1.2 and 7 is combined with the other values to form a vector, for which the mean is calculated, then the square root of this value is taken and used as the standard deviation in a call to `rnorm`, and the output of this call is plotted). Of course, it is often easier to assign intermediate steps to symbolic variables. `R` evaluates nested expressions based on the values that functions return or the data represented by symbolic variables. For example, if a function expects character data for a particular argument, then you can use a call to the function `paste` in place of explicit character data.

Many functions (including `sum`, `plot`, and `rnorm`) come with the `R` "base packages", i.e., they are loaded and ready to go as soon as you open `R`. These packages contain the most common functions. While the base packages include many useful functions, for specialized procedures, you should check out the content that is available in the add-on packages. The CRAN website currently lists more than 4500 contributed packages that contain functions and data that users have contributed. You can find a list of the available packages at the CRAN website http://cran.r-project.org/. During the course of this book and described in more detail later on, we will be looking and using a number of specialized packages for application of DSM. Another repository of `R` packages is the R-Forge website https://r-forge.r-project.org/. R-Forge offers a central platform for the development of R packages, R-related software and further projects. Packages in R-Forge are not necessarily always on the CRAN website. However, many packages on the CRAN website are developed in R-Forge as ongoing projects. Sometimes to get the latest changes

made upon a package, it pays to visit R-Forge first, as the uploading of the revised functions to CRAN is not instantaneous.

To utilize the functions in contributed R packages, you first need to install and then load the package. Packages can be installed via the packages menu in the right bottom panel of RStudio (select the "packages" menu, then "install packages"). Installation could be retrieved from the nearest mirror site (CRAN server location)—you will need to have first selected this by going to the *tools*, then *options*, then *packages* menu where you can then select the nearest mirror site from a suite of possibles. Alternatively, you may just install a package from a local zip file. This is fine, but often when using a package, there are other peripheral packages (or dependencies) that also need to be loaded (and installed). If you install the package from CRAN or a mirror site, the dependency packages are also installed. This is not the case when you are installing packages from zip files—you will also have to manually install all the dependencies too.

Or just use the command:

```
install.packages("package name")
```

where "package name" should be replaced with the actual name of the package you want to install, for example:

```
install.packages("Cubist")
```

This command will install the package of functions for running the Cubist rule-based machine learning models for regression.

Installation is a one-time process, but packages must be loaded each time you want to use them. This is very simple, e.g., to load the package `Cubist`, use the following command.

```
library(Cubist)
```

Similarly, if you want to install an `R` package from R-Forge (another popular hosting repository for R packages) you would use the following command:

```
install.packages("package name", repos = "http://R-Forge.R-project.org")
```

Other popular repositories for R packages include Github and BitBucket. These repositories as well as R-Forge are version control systems that provide a central place for people to collaborate on everything from small to very large projects with speed and efficiency. The companion `R` package to this book, *ithir* is hosted on Bitbucket for example. *ithir* contains most of the data, and some important functions that are covered in this book so that users can replicate all of the analyses contained within. *ithir* can be downloaded and installed on your computer using the following commands:

```
library(devtools)
install_bitbucket("brendo1001/ithir/pkg")
library(ithir)
```

The above commands assumes your have already installed the devtools package. Any package that you want to use that is not included as one of the "base" packages, needs to be loaded every time you start R. Alternatively, you can add code to the file *Rprofile.site* that will be executed every time you start R.

You can find information on specific packages through CRAN, by browsing to http://cran.r-project.org/ and selecting the *packages* link. Each package has a separate web page, which will include links to source code, and a pdf manual. In RStudio, you can select the *packages* tab on the lower right panel. You will then see all the package that are currently installed in your R environment. By clicking onto any package, information on the various functions contained in the package, plus documentation and manuals for their usage. It becomes quite clear that within this RStudio environment, there is at your fingertips, a wealth of information for which to consult whenever you get stuck. When working with a new package, it is a good idea to read the manual.

To "unload" functions, use the detach function:

```
detach("package:Cubist")
```

For tasks that you repeat, but which have no associated function in R, or if you do not like the functions that are available, you can write your own functions. This will be covered a little a bit later on. Perhaps one day you may be able to compile all your functions that you have created into a R package for everyone else to use.

2.2.10 Getting Help

It is usually easy to find the answer about specific functions or about R in general. There are several good introductory books on R. For example, "R for Dummies", which has had many positive reviews http://www.amazon.com/R-Dummies-Joris-Meys/dp/1119962846.You can also find free detailed manuals on the CRAN website. Also, it helps to keep a copy of the "R Reference Card", which demonstrates the use of many common functions and operators in 4 pages http://cran.r-project.org/doc/contrib/Short-refcard.pdf. Often a Google search https://www.google.com.au/ of your problem can be a very helpful and fruitful exercise. To limit the results to R related pages, adding "cran" generally works well. R even has an internet search engine of sorts called *rseek*, which can be found at http://rseek.org/—it is really just like the Google search engine, but just for R stuff!

Each function in R has a help file associated with it that explains the syntax and usually includes an example. Help files are concisely written. You can bring up a help file by typing ? and then the function name.

```
>?cubist
```

This will bring up the help file for the cubist function in the help panel of RStudio. But, what if you are not sure what function you need for a particular task? How can you know what help file to open? In addition to the sources given

below, you should try `help.search("keyword")` or `??keyword`, both of which search the R help files for whatever keyword you put in.

```
>??polygon
```

This will bring up a search results page in the help panel of RStudio of all the various help files that have something to do with `polygon`. In this case, i am only interested in a function that assesses whether a point is situated with a polygon. So looking down the list, one can see (provided the "SDMTools" package is installed) a function called `pnt.in.poly`. Clicking on this function, or submitting `?pnt.in.poly` to R will bring up the necessary help file.

There is an R help mailing list http://www.r-project.org/mail.html, which can be very helpful. Before posting a question, be sure to search the mailing list archives, and check the posting guide http://www.r-project.org/posting-guide.html.

One of the best sources of help on R functions is the mailing list archives (urlhttp://cran.r-project.org/, then select "search", then "searchable mail archives"). Here you can find suggestions for functions for particular problems, help on using specific functions, and all kinds of other information. A quick way to search the mailing list archives is by entering:

```
RSiteSearch("A Keyword")
```

For one more trick, to search for objects (including functions) that include a particular string, you can use the `apropos` function:

```
apropos("mean")
```

```
##  [1] ".colMeans"    ".rowMeans"    "colMeans"     "kmeans"
##  [5] "mean"   "mean.Date"    "mean.default"   "mean.difftime"
##  [9] "mean.POSIXct" "mean.POSIXlt" "rowMeans" "weighted.mean"
```

2.2.11 Exercises

1. You can use for magic tricks: Pick any number. Double it, and then add 12 to the result. Divide by 2, and then subtract your original number. Did you end up with 6.0?
2. If you want to work with a set of 10 numbers in R, something like this:
   11 8.3 9.8 9.6 11.0 12.0 8.5 9.9 10.0 11.0
   - What type of data structure should you use to store these in R?
   - What if you want to work with a data set that contains site names, site locations, soil categorical information, soil property information, and some terrain variables—what type of data structure should you use to store these in R?
3. Install and load a package—take a look at the list of available packages, and pick one. To make sure you have loaded it correctly, try to run an example from

the package reference manual. Identify the arguments required for calling up the function. Detach the package when you are done.
4. Assign your full name to a variable called `my.name`. Print the value of `my.name`. Try to subtract 10 from `my.name`. Finally determine the type of data stored in `my.name` and 10 using the `class` function. If you are unsure of what `class` does, check out the help file.
5. You are interested in seeing what functions `R` has for fitting variograms (or some other topic of your choosing). Can you figure out how to search for relevant functions? Are you able to identify a function or two that may do what you want.

2.3 Vectors, Matrices, and Arrays

2.3.1 Creating and Working with Vectors

There are several ways to create a vector in `R`. Where the elements are spaced by exactly 1, just separate the values of the first and last elements with a colon.

```
1:5
```

```
## [1] 1 2 3 4 5
```

The function `seq` (for sequence) is more flexible. Its typical arguments are `from, to`, and `by` (or, in place of `by`, you can specify `length.out`).

```
seq(-10, 10, 2)
```

```
##  [1] -10  -8  -6  -4  -2   0   2   4   6   8  10
```

Note that the `by` argument does not need to be an integer. When all the elements in a vector are identical, use the `rep` function (for repeat).

```
rep(4, 5)
```

```
## [1] 4 4 4 4 4
```

For other cases, use `c` (for concatenate or combine).

```
c(2, 1, 5, 100, 2)
```

```
## [1]   2   1   5 100   2
```

Note that you can name the elements within a vector.

```
c(a = 2, b = 1, c = 5, d = 100, e = 2)
```

```
##   a   b   c   d   e
##   2   1   5 100   2
```

Any of these expressions could be assigned to a symbolic variable, using an assignment operator.

```
v1 <- c(2, 1, 5, 100, 2)
v1
```

```
## [1]   2   1   5 100   2
```

Variable names can be any combination of letters, numbers, and the symbols . and _, but they can not start with a number or with _. Google has a R style guide http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html which describes good and poor examples of variable name attribution, but generally it is a personal preference on how you name your variables.

```
probably.not_a.good_example.for.a.name.100 <- seq(1, 2, 0.1)
probably.not_a.good_example.for.a.name.100
```

```
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

The c function is very useful for setting up arguments for other functions, as will be shown later. As with all R functions, both variable names and function names can be substituted into function calls in place of numeric values.

```
x <- rep(1:3)
y <- 4:10
z <- c(x, y)
z
```

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

Although R prints the contents of individual vectors with a horizontal orientation, R does not have "column vectors" and "row vectors", and vectors do not have a fixed orientation. This makes use of vectors in R very flexible.

Vectors do not need to contain numbers, but can contain data with any of the modes mentioned earlier (numeric, logical, character, and complex), as long as all the data in a vector are of the same mode.

Logical vectors are very useful in R for sub-setting data i.e., for isolating some part of an object that meets certain criteria. For relational commands, the shorter vector is repeated as many as necessary to carry out the requested comparison for each element in the longer vector.

```
x <- 1:10
x > 5
```

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

Also, note that when logical vectors are used in arithmetic, they are changed (coerced in R terms) into a vector of binary elements: 1 or 0. Continuing with the above example:

```
a <- x > 5
a

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE TRUE TRUE

a * 1.4

##  [1] 0.0 0.0 0.0 0.0 0.0 1.4 1.4 1.4 1.4 1.4
```

One function that is commonly used on character data is `paste`. It concatenates character data (and can also work with numerical and logical elements—these become character data).

```
paste("A", "B", "C", "D", TRUE, 42)

## [1] "A B C D TRUE 42"
```

Note that the `paste` function is very different from `c`. The `paste` function concatenates its arguments into a single character value, while the `c` function combines its arguments into a vector, where each argument becomes a single element. The `paste` function becomes handy when you want to combine the character data that are stored in several symbolic variables.

```
month <- "April"
day <- 29
year <- 1770
paste("Captain Cook, on the ", day, "th day of ", month, ", "
, year, ", sailed into Botany Bay", sep = "")

## [1] "Captain Cook, on the 29th day of April, 1770,
##        sailed into Botany Bay"
```

This is especially useful with loops, when a variable with a changing value is combined with other data. Loops will be discussed in a later section.

```
group <- 1:10
id <- LETTERS[1:10]
for (i in 1:10) {
    print(paste("group =", group[i], "id =", id[i]))
}

## [1] "group = 1 id = A"
## [1] "group = 2 id = B"
## [1] "group = 3 id = C"
## [1] "group = 4 id = D"
## [1] "group = 5 id = E"
## [1] "group = 6 id = F"
## [1] "group = 7 id = G"
## [1] "group = 8 id = H"
## [1] "group = 9 id = I"
## [1] "group = 10 id = J"
```

`LETTERS` is a constant that is built into `R`—it is a vector of uppercase letters A through Z (different from `letters`).

2.3.2 *Vector Arithmetic, Some Common Functions, and Vectorised Operations*

In R, vectors can be used directly in arithmetic operations. Operations are applied on an element-by-element basis. This can be referred to as "vectorised" arithmetic, and along with vectorised functions (described below), it is a quality that makes R a very efficient programming language.

```
x <- 6:10
x
```

```
## [1]  6  7  8  9 10
```

```
x + 2
```

```
## [1]  8  9 10 11 12
```

For an operation carried out on two vectors, the mathematical operation is applied on an element-by-element basis.

```
y <- c(4, 3, 7, 1, 1)
y
```

```
## [1] 4 3 7 1 1
```

```
z <- x + y
z
```

```
## [1] 10 10 15 10 11
```

When two vectors having different numbers of elements used in an expression together, R will repeat the smaller vector. For example, with vector of length one, i.e., a single number:

```
x <- 1:10
m <- 0.8
b <- 2
y <- m * x + b
y
```

```
##  [1]  2.8  3.6  4.4  5.2  6.0  6.8  7.6  8.4  9.2 10.0
```

If the number of rows in the smaller vector is not a multiple of the larger vector (often indicative of an error) R will return a warning.

```
x <- 1:10
m <- 0.8
b <- c(2, 1, 1)
y <- m * x + b
```

```
## Warning in m * x + b: longer object length is not a multiple
of shorter object length
```

```
y
```

```
##  [1]  2.8  2.6  3.4  5.2  5.0  5.8  7.6  7.4  8.2 10.0
```

Some arithmetic operators that are available in R include:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	exponentiation
`%/%`	integer division
`%%`	modulo (remainder)
`log (a)`	natural log of a
`log10 (a)`	base 10 log of a
`exp (a)`	e^a
`sine (a)`	sine of a
`cos (a)`	cosine of a
`tan (a)`	tangent of a
`sqrt (a)`	square root of a

Some simple functions that are useful for vector math include:

`min`	minimum value of a set of numbers
`max`	maximum of a set of numbers
`pmin`	parallel minima (compares multiple vectors "row-by-row")
`pmax`	parallel maxima
`sum`	sum of all elements
`length`	length of a vector (or number of columns in a data frame)
`nrow`	number of rows in a vector of data frame
`ncol`	number of columns
`mean`	arithmetic mean
`sd`	standard deviation
`rnorm`	generates a vector of normally-distributed random numbers
`signif, ceiling, floor`	rounding

Many, many other functions are available.

R also has a few built in constants, including `pi`.

```
pi
```

```
## [1] 3.141593
```

Parentheses can be used to control the order of operations, as in any other programming language.

```
7 - 2 * 4
```

```
## [1] -1
```

is different from:

```
(7 - 2) * 4
```

```
## [1] 20
```

and

```
10^1:5
```

```
## [1] 10  9  8  7  6  5
```

is different from:

```
10^(1:5)
```

```
## [1] 1e+01 1e+02 1e+03 1e+04 1e+05
```

Many functions in R are capable of accepting vectors (or even data frames, arrays, and lists) as input for single arguments, and returning an object with the same structure. These vectorised functions make vector manipulations very efficient. Examples of such functions include `log`, `sin`, and `sqrt`, For example:

```
x <- 1:10
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

or

```
sqrt(1:10)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

The previous expressions are also equivalent to:

```
sqrt(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

But they are not the same as the following, where all the numbers are interpreted as individual values for multiple arguments.

```
sqrt(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
## Error in sqrt(1, 2, 3, 4, 5, 6, 7, 8, 9, 10): 10 arguments
passed to 'sqrt' which requires 1
```

There are also some functions designed for making vectorised operations on lists, matrices, and arrays: these include `apply` and `lapply`.

2.3.3 *Matrices and Arrays*

Arrays are multi-dimensional collections of elements and matrices are simply two-dimensional arrays. `R` has several operators and functions for carrying out operations on arrays, and matrices in particular (e.g., matrix multiplication).

To generate a matrix, the `matrix` function can be used. For example:

```
X <- matrix(1:15, nrow = 5, ncol = 3)
X
```

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

Note that the filling order is by column by default (i.e., each column is filled before moving onto the next one). The "unpacking" order is the same:

```
as.vector(X)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

If, for any reason, you want to change the filling order, you can use the `by row` argument:

```
X <- matrix(1:15, nrow = 5, ncol = 3, byrow = T)
X
```

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

A similar function is available for higher-order arrays, called `array`. Here is an example with a three-dimensional array:

```
Y <- array(1:30, dim = c(5, 3, 2))
Y
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   16   21   26
## [2,]   17   22   27
## [3,]   18   23   28
## [4,]   19   24   29
## [5,]   20   25   30
```

Arithmetic with matrices and arrays that have the same dimensions is straightforward, and is done on an element-by-element basis. This true for all the arithmetic operators listed in earlier sections.

```
Z <- matrix(1, nrow = 5, ncol = 3)
Z
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
## [4,]    1    1    1
## [5,]    1    1    1
```

```
X + Z
```

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

This does not work when dimensions do not match:

```
Z <- matrix(1, nrow = 3, ncol = 3)
X + Z
```

```
## Error in X + Z: non-conformable arrays
```

For mixed vector/array arithmetic, vectors are recycled if needed.

```
Z
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

```
x <- 1:9
Z + x
```

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

```
y <- 1:3
Z     + y
```

```
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    3    3    3
## [3,]    4    4    4
```

R also has operators for matrix algebra. The operator `%*%` carries out matrix multiplication, and the function `solve` can invert matrices.

```
X <- matrix(c(1, 2.5, 6, 7.5, 4.9, 5.6, 9.9, 7.8, 9.3), nrow = 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]  1.0  7.5  9.9
## [2,]  2.5  4.9  7.8
## [3,]  6.0  5.6  9.3
```

```
solve(X)
```

```
##              [,1]       [,2]       [,3]
## [1,]  0.07253886 -0.5492228  0.3834197
## [2,]  0.90385723 -1.9228555  0.6505469
## [3,] -0.59105738  1.5121858 -0.5315678
```

2.3.4 *Exercises*

1. Generate a vector of numbers that contains the sequence 1, 2, 3, ... 10 (try to use the least amount of code possible to do this). Assign this vector to the variable x, and then carry out the following vector arithmetic.

   (a) $log_{10}x$
   (b) lnx
   (c) $\frac{\sqrt{x}}{2-x}$

2. Use an appropriate function to generate a vector of 100 numbers that go from 0 to 2π, with a constant interval. Assuming this fist vector is called x, create a new vector that contains $sine(2x - 0.5\pi)$. Determine the minimum and maximum of $sine(2x - 0.5\pi)$. Does this match with what you expect?
3. Create 5 vectors, each containing 10 random numbers. Give each vector a different name. Create a new vector where the 1st element contains the sum of the 1st elements in your original 5 vectors, the 2nd element contains the sum of the 2nd elements, etc. Determine the mean of this new vector.
4. Create the following matrix using the least amount of code:

```
##       [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
```

5. If you are bored, try this. Given the following set of linear equations:

$$27.2x + 32y - 10.8z = 401.2$$
$$x - 1.48y = 0$$
$$409.1x + 13.5z = 2.83$$

Solve for x, y, and z.

2.4 Data Frames, Data Import, and Data Export

As described above, a data frame is a type of data structure in R with rows and columns, where different columns contain data with different modes. A data frame is probably the most common data structure that you will use for storing soil information data sets. Recall from before the data frame that we created.

```
dat <- data.frame(profile_id = c("Chromosol", "Vertosol", "Sodosol"),
FID = c("a1", "a10", "a11"), easting = c(337859, 344059, 347034),
northing = c(6372415, 6376715, 6372740), visted = c(TRUE, FALSE, TRUE))
dat
```

```
##   profile_id FID easting northing visted
## 1  Chromosol  a1  337859  6372415   TRUE
## 2   Vertosol a10  344059  6376715  FALSE
## 3    Sodosol a11  347034  6372740   TRUE
```

We can quickly assess the different modes of a data frame (or any other object for that matter) by using the `str`(structure) function.

```
str(dat)
```

```
## 'data.frame': 3 obs. of  5 variables:
```

```
##  $ profile_id: Factor w/ 3 levels "Chromosol","Sodosol",..:
##  $ FID       : Factor w/ 3 levels "a1","a10","a11": 1 2 3
##  $ easting   : num  337859 344059 347034
##  $ northing  : num  6372415 6376715 6372740
##  $ visted    : logi  TRUE FALSE TRUE
```

The `str` is probably one of the most used `R` functions. It is great for exploring the format and contents of any object created or imported.

2.4.1 Reading Data from Files

The easiest way to create a data frame is to read in data from a file—this is done using the function `read.table`, which works with ASCII text files. Data can be read in from other files as well, using different functions, but `read.table` is the most commonly used approach. `R` is very flexible in how it reads in data from text files.

Note that the column labels in the header have to be compatible with `R`'s variable naming convention, or else `R` will make some changes as they are read in (or will not read in the data correctly). So lets import some real soil data. These soil data are some chemical and physical properties from a collection of soil profiles sampled at various locations in New South Wales, Australia

```
soil.data <- read.table("USYD_soil1.txt", header = TRUE, sep = ",")
```

```
## Warning in file(file, "rt"): cannot open file
'USYD_soil1.txt': No such file or directory
```

```
## Error in file(file, "rt"): cannot open the connection
```

```
str(soil.data)
```

```
## Error in str(soil.data): object 'soil.data' not found
```

```
head(soil.data)
```

```
## Error in head(soil.data): object 'soil.data' not found
```

However, you may find that an error occurs, saying something like that the file does not exist. This is true as it has not been provided to you. Rather, to use this data you will need to load up the previously installed `ithir` package.

```
library(ithir)
data(USYD_soil1)
soil.data <- USYD_soil1
str(soil.data)
```

```
## 'data.frame': 166 obs. of  16 variables:
```

```
##  $ PROFILE     : int  1 1 1 1 1 1 2 2 2 2 ...
##  $ Landclass   : Factor w/ 4 levels "Cropping","Forest",..: 4 4 4 ...
##  $ Upper.Depth : num  0 0.02 0.05 0.1 0.2 0.7 0 0.02 0.05 0.1 ...
##  $ Lower.Depth : num  0.02 0.05 0.1 0.2 0.3 0.8 0.02 0.05 0.1 0.2 ...
##  $ clay        : int  8 8 8 8 NA 57 9 9 9 NA ...
##  $ silt        : int  9 9 10 10 10 8 10 10 10 10 ...
##  $ sand        : int  83 83 82 83 79 36 81 80 80 81 ...
##  $ pH_CaCl2    : num  6.35 6.34 4.76 4.51 4.64 6.49 5.91 5.94 5.63 4.22 ...
##  $ Total_Carbon: num  1.07 0.98 0.73 0.39 0.23 0.35 1.14 1.14 1.01 0.48 ...
##  $ EC          : num  0.168 0.137 0.072 0.034 NA 0.059 0.123 0.101...
##  $ ESP         : num  0.3 0.5 0.9 0.2 0.9 0.3 0.3 0.6 NA NA ...
##  $ ExchNa      : num  0.01 0.02 0.02 0 0.02 0.04 0.01 0.02 NA NA ...
##  $ ExchK       : num  0.71 0.47 0.52 0.38 0.43 0.46 0.7 0.56 NA NA ...
##  $ ExchCa      : num  3.17 3.5 1.34 1.03 1.5 9.13 2.92 3.2 NA NA ...
##  $ ExchMg      : num  0.59 0.6 0.22 0.22 0.5 5.02 0.51 0.5 NA NA ...
##  $ CEC         : num  5.29 3.7 2.86 2.92 2.6 ...
```

```
head(soil.data)
```

```
##   PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 1       1 native pasture        0.00        0.02    8    9   83      6.35
## 2       1 native pasture        0.02        0.05    8    9   83      6.34
## 3       1 native pasture        0.05        0.10    8   10   82      4.76
## 4       1 native pasture        0.10        0.20    8   10   83      4.51
## 5       1 native pasture        0.20        0.30   NA   10   79      4.64
## 6       1 native pasture        0.70        0.80   57    8   36      6.49
##   Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg   CEC
## 1         1.07 0.168 0.3   0.01  0.71   3.17   0.59  5.29
## 2         0.98 0.137 0.5   0.02  0.47   3.50   0.60  3.70
## 3         0.73 0.072 0.9   0.02  0.52   1.34   0.22  2.86
## 4         0.39 0.034 0.2   0.00  0.38   1.03   0.22  2.92
## 5         0.23     NA 0.9   0.02  0.43   1.50   0.50  2.60
## 6         0.35 0.059 0.3   0.04  0.46   9.13   5.02 14.96
```

When we import a file into R, it is good habit to look at its structure (`str`) to ensure the data is as it should be. As can be seen, this data set (frame) has 166 observations, and 15 columns. The `head` function is also a useful exploratory function, which simply allows us to print out the data frame, but only the first 6 rows of it (good for checking data frame integrity). Note that you must specify `header=TRUE`, or else R will interpret the row of labels as data. If the file you are loading is not in the directory that R is working in (the working directory, which can be checked with `getwd()` and changed with `setwd(file = "filename")`). When setting the working directory (`setwd()`), you can include the file path, but note that the path should have forward, not backward slashes (or double backward slashes, if you prefer).

The column separator function `sep` (an argument of `read.table`) lets you tell R, where the column breaks or delimiters occur. In the `soil.data` object, we specify that the data is comma separated. If you do not specify a field separator, R assumes that any spaces or tabs separate the data in your text file. However, any character data that contain spaces must be surrounded by quotes (otherwise, R interprets the data on either side of the white spaces as different elements).

Besides comma separators, tab separators are also common `sep="\t"`, so is `sep="."` (full stop separator). Two consecutive separators will be interpreted as a missing value. Conversely, with the default options, you need to explicitly identify missing values in your data file with `NA` (or any other character, as long as you tell `R` what it is with the `na.strings` argument).

For some field separators, there are alternate functions that can be used with the default arguments, e.g., `read.csv`, which is identical to `read.table`, except default arguments differ. Also, `R` does not care what the name of your file is, or what its extension is, as long as it is an ASCII text file. A few other handy bits of information for using `read.table` follow. You can include comments at the end of rows in your data file—just precede them with a `#`. Also `R` will recognize `NaN`, `Inf`, and `-Inf` in input files.

Probably the easiest approach to handling missing values is to indicate their presence with `NA` in the text file. `R` will automatically recognize these as missing values. Alternatively, just leave the missing values as they are (blank spaces), and let `R` do the rest.

```
which(is.na(soil.data$CEC))
```

```
## [1]   9  10  45  63 115
```

```
soil.data[8:11, ]
```

```
##    PROFILE        Landclass Upper.Depth Lower.Depth clay silt sand
## 8        2 improved pasture        0.02        0.05    9   10   80
## 9        2 improved pasture        0.05        0.10    9   10   80
## 10       2 improved pasture        0.10        0.20   NA   10   81
## 11       2 improved pasture        0.38        0.50   36    8   56
##    pH_CaCl2 Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg CEC
## 8      5.94         1.14 0.101 0.6   0.02  0.56    3.2    0.5 4.0
## 9      5.63         1.01 0.026  NA     NA    NA     NA     NA  NA
## 10     4.22         0.48 0.042  NA     NA    NA     NA     NA  NA
## 11     6.48         0.18 0.053 0.4   0.04  0.49    6.0    2.3 8.6
```

In most cases, it makes sense to put your data into a text file for reading into `R`. This can be done in various ways. Data download from the internet are often in text files to begin with. Data can be entered directly into a text file using a text editor. For data that are in spreadsheet program such as Excel or JMP, there are facilities available for saving these tabular frames to text files for reading into `R`.

This all may seem confusing, but it is really not that bad. Your best bet is to play around with the different options, find one that you like, and stick with it. Lastly, data frames can also be edited interactively in `R` using the `edit` function. This is really only useful for small data sets, and the function is not supported by RStudio (you could try with using Tinn-R instead if you want to explore using this function)

```
soil.data <- edit(soil.data)
```

2.4.2 Creating Data Frames Manually

Data frames can be made manually using the `data.frame` function:

```
soil <- c("Chromosol", "Vertosol", "Organosol", "Anthroposol")
carbon <- c(2.1, 2.9, 5.5, 0.2)
dat <- data.frame(soil.type = soil, soil.OC = carbon)
dat
```

```
##      soil.type soil.OC
## 1   Chromosol     2.1
## 2    Vertosol     2.9
## 3   Organosol     5.5
## 4 Anthroposol     0.2
```

While this approach is not an efficient way to enter data that could be read in directly, it can be very handy for some applications, such as the creation of customized summary tables. Note that column names are specified using an equal sign. It is also possible to specify (or change, or check) column names for an existing data frame using the function `names`.

```
names(dat) <- c("soil", "SOC")
dat
```

```
##           soil SOC
## 1   Chromosol 2.1
## 2    Vertosol 2.9
## 3   Organosol 5.5
## 4 Anthroposol 0.2
```

Row names can be specified in the `data.frame` function with the `row.names` argument.

```
dat <- data.frame(soil.type = soil, soil.OC = carbon,
row.names = c("Ch", "Ve", "Or", "An"))
dat
```

```
##       soil.type soil.OC
## Ch   Chromosol     2.1
## Ve    Vertosol     2.9
## Or   Organosol     5.5
## An Anthroposol     0.2
```

Specifying row names can be useful if you want to index data, which will be covered later. Row names can also be specified for an existing data frame with the `rownames` function (not to be confused with the `row.names` argument).

2.4.3 Working with Data Frames

So what do you do with data in R once it is in a data frame? Commonly, the data in a data frame will be used in some type of analysis or plotting procedure. It is usually necessary to be able to select and identify specific columns (i.e., vectors) within data frames. There are two ways to specify a given column of data within a data frame. The first is to use the $ notation. To see what the column names are, we can use the `names` function. Using our `soil.data` set:

```
names(soil.data)
```

```
##  [1] "PROFILE"       "Landclass"    "Upper.Depth"  "Lower.Depth"
##  [5] "clay"          "silt"         "sand"         "pH_CaCl2"
##  [9] "Total_Carbon" "EC"           "ESP"          "ExchNa"
## [13] "ExchK"         "ExchCa"       "ExchMg"       "CEC"
```

The $ just uses a $ between the data frame and column name to specify a particular column. Say we want to look at the `ESP` column, which is the acronym for exchangeable sodium percentage.

```
soil.data$ESP
```

```
##   [1]  0.3  0.5  0.9  0.2  0.9  0.3  0.3  0.6   NA   NA  0.4  0.9  0.2  0.1
##  [15]   NA  0.4  0.5  0.7  0.2  0.1   NA  0.2  0.3   NA  0.8  0.6  0.8  0.9
##  [29]  0.9  1.1  0.5  0.6  1.1  0.2  0.6  1.1  0.4  0.3  0.5  1.0  2.2  0.1
##  [43]  0.1  0.1   NA  0.1  0.1  0.4   NA  0.2  0.1  0.3  0.4  0.1  0.4  0.1
##  [57]  0.1   NA  0.1  0.3   NA  0.1   NA  0.2  1.8  2.6  0.2 13.0  0.0  0.1
##  [71]  0.3  0.1  0.1  0.3  0.0  0.3  0.6  0.9  0.4   NA   NA  2.4  0.2  0.3
##  [85]  0.2  0.0  0.1  0.4   NA  0.3  0.3  0.2  0.3  0.6  0.3  0.2  0.7  0.3
##  [99]  0.4  1.0  7.9  6.1  5.7  5.2  4.7  2.9  5.8  7.2  9.6   NA 17.4   NA
## [113] 11.1  6.4   NA  4.0 12.1 21.2  2.2  1.9   NA  4.0 13.2  0.9  0.8  0.5
## [127]  0.2  0.4   NA  1.2  0.6  0.2  1.0  0.4  0.6  0.1  0.4   NA  0.7  0.5
## [141]  0.7  0.9  4.8  3.8  4.9  6.2 10.4 16.4  2.7   NA  1.2  0.5  1.9  2.0
## [155]  2.1  1.9  1.8  3.5  7.7  2.7  1.8  0.8  0.5  0.5  0.3  0.9
```

Although it is handy to think of data frame columns to have a vertical orientation, this orientation is not present when they are printed individually—instead, elements are printed from left to right, and then top to bottom. The expression `soil.data$ESP` could be used just as you would for any other vector. For example:

```
mean(soil.data$ESP)
```

```
## [1] NA
```

R can not calculate the mean because of the `NA` values in the vector. Lets remove them first using the `na.omit` function.

```
mean(na.omit(soil.data$ESP))
```

```
## [1] 1.99863
```

The second option for working with individual columns within a data frame is to use the commands `attach` and `detach`. Both of these functions take a data frame as an argument. `attach`ing a data frame puts all the columns within the that data frame into R's search path, and they can be called by using their names alone without the `$` notation.

```
attach(soil.data)
ESP
```

```
##   [1]  0.3  0.5  0.9  0.2  0.9  0.3  0.3  0.6   NA   NA  0.4  0.9  0.2  0.1
##  [15]   NA  0.4  0.5  0.7  0.2  0.1   NA  0.2  0.3   NA  0.8  0.6  0.8  0.9
##  [29]  0.9  1.1  0.5  0.6  1.1  0.2  0.6  1.1  0.4  0.3  0.5  1.0  2.2  0.1
##  [43]  0.1  0.1   NA  0.1  0.1  0.4   NA  0.2  0.1  0.3  0.4  0.1  0.4  0.1
##  [57]  0.1   NA  0.1  0.3   NA  0.1   NA  0.2  1.8  2.6  0.2 13.0  0.0  0.1
##  [71]  0.3  0.1  0.1  0.3  0.0  0.3  0.6  0.9  0.4   NA   NA  2.4  0.2  0.3
##  [85]  0.2  0.0  0.1  0.4   NA  0.3  0.3  0.2  0.3  0.6  0.3  0.2  0.7  0.3
##  [99]  0.4  1.0  7.9  6.1  5.7  5.2  4.7  2.9  5.8  7.2  9.6   NA 17.4   NA
## [113] 11.1  6.4   NA  4.0 12.1 21.2  2.2  1.9   NA  4.0 13.2  0.9  0.8  0.5
## [127]  0.2  0.4   NA  1.2  0.6  0.2  1.0  0.4  0.6  0.1  0.4   NA  0.7  0.5
## [141]  0.7  0.9  4.8  3.8  4.9  6.2 10.4 16.4  2.7   NA  1.2  0.5  1.9  2.0
## [155]  2.1  1.9  1.8  3.5  7.7  2.7  1.8  0.8  0.5  0.5  0.3  0.9
```

Note that when you are done using the individual columns, it is good practice to `detach` your data frame. Once the data frame is `detach`ed, R will no longer know what you mean when you specify the name of the column alone:

```
detach(soil.data)
ESP
```

```
## Error in eval(expr, envir, enclos): object 'ESP' not found
```

If you modify a variable that is part of an attached data frame, the data within the data frame remain unchanged; you are actually working with a copy of the data frame.

Another option (for selecting particular columns) is to use the square braces `[]` to specify the column you want. Using the square braces to select the `ESP` column from our data set you would use:

```
soil.data[, 10]
```

Here you are specifying the column in the tenth position, which as you should check is the `ESP` column. To use the square braces the row position precedes to comma, and the column position proceeds to comma. By leaving a blank space in front of the comma, we are essentially instruction R to print out the whole column. You may be able to surmise that it is also possible to subset a selection of columns quite efficiently with this square brace method. We will use the square braces more a little later on.

The `$` notation can also be used to add columns to a data frame. For example, if we want to express our `Upper.Depth` and `Lower.Depth` columns in *cm* rather than *m* we could do the following.

```
soil.data$Upper <- soil.data$Upper.Depth * 100
soil.data$Lower <- soil.data$Lower.Depth * 100
head(soil.data)
```

```
##   PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 1       1 native pasture        0.00        0.02    8    9   83      6.35
## 2       1 native pasture        0.02        0.05    8    9   83      6.34
## 3       1 native pasture        0.05        0.10    8   10   82      4.76
## 4       1 native pasture        0.10        0.20    8   10   83      4.51
## 5       1 native pasture        0.20        0.30   NA   10   79      4.64
## 6       1 native pasture        0.70        0.80   57    8   36      6.49
##   Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg   CEC Upper Lower
## 1         1.07 0.168 0.3   0.01  0.71   3.17   0.59  5.29     0     2
## 2         0.98 0.137 0.5   0.02  0.47   3.50   0.60  3.70     2     5
## 3         0.73 0.072 0.9   0.02  0.52   1.34   0.22  2.86     5    10
## 4         0.39 0.034 0.2   0.00  0.38   1.03   0.22  2.92    10    20
## 5         0.23    NA 0.9   0.02  0.43   1.50   0.50  2.60    20    30
## 6         0.35 0.059 0.3   0.04  0.46   9.13   5.02 14.96    70    80
```

Many data frames that contain real data will have some missing observations. R has several tools for working with these observations. For starters, the `na.omit` function can be used for removing NAs from a vector. Working again with the `ESP` column of our `soil.data` set:

```
soil.data$ESP
```

```
##   [1]  0.3  0.5  0.9  0.2  0.9  0.3  0.3  0.6   NA   NA  0.4  0.9  0.2  0.1
##  [15]   NA  0.4  0.5  0.7  0.2  0.1   NA  0.2  0.3   NA  0.8  0.6  0.8  0.9
##  [29]  0.9  1.1  0.5  0.6  1.1  0.2  0.6  1.1  0.4  0.3  0.5  1.0  2.2  0.1
##  [43]  0.1  0.1   NA  0.1  0.1  0.4   NA  0.2  0.1  0.3  0.4  0.1  0.4  0.1
##  [57]  0.1   NA  0.1  0.3   NA  0.1   NA  0.2  1.8  2.6  0.2 13.0  0.0  0.1
##  [71]  0.3  0.1  0.1  0.3  0.0  0.3  0.6  0.9  0.4   NA   NA  2.4  0.2  0.3
##  [85]  0.2  0.0  0.1  0.4   NA  0.3  0.3  0.2  0.3  0.6  0.3  0.2  0.7  0.3
##  [99]  0.4  1.0  7.9  6.1  5.7  5.2  4.7  2.9  5.8  7.2  9.6   NA 17.4   NA
## [113] 11.1  6.4   NA  4.0 12.1 21.2  2.2  1.9   NA  4.0 13.2  0.9  0.8  0.5
## [127]  0.2  0.4   NA  1.2  0.6  0.2  1.0  0.4  0.6  0.1  0.4   NA  0.7  0.5
## [141]  0.7  0.9  4.8  3.8  4.9  6.2 10.4 16.4  2.7   NA  1.2  0.5  1.9  2.0
## [155]  2.1  1.9  1.8  3.5  7.7  2.7  1.8  0.8  0.5  0.5  0.3  0.9
```

```
na.omit(soil.data$ESP)
```

```
##   [1]  0.3  0.5  0.9  0.2  0.9  0.3  0.3  0.6  0.4  0.9  0.2  0.1  0.4  0.5
##  [15]  0.7  0.2  0.1  0.2  0.3  0.8  0.6  0.8  0.9  0.9  1.1  0.5  0.6  1.1
##  [29]  0.2  0.6  1.1  0.4  0.3  0.5  1.0  2.2  0.1  0.1  0.1  0.1  0.1  0.4
##  [43]  0.2  0.1  0.3  0.4  0.1  0.4  0.1  0.1  0.1  0.3  0.1  0.2  1.8  2.6
##  [57]  0.2 13.0  0.0  0.1  0.3  0.1  0.1  0.3  0.0  0.3  0.6  0.9  0.4  2.4
##  [71]  0.2  0.3  0.2  0.0  0.1  0.4  0.3  0.3  0.2  0.3  0.6  0.3  0.2  0.7
##  [85]  0.3  0.4  1.0  7.9  6.1  5.7  5.2  4.7  2.9  5.8  7.2  9.6 17.4 11.1
##  [99]  6.4  4.0 12.1 21.2  2.2  1.9  4.0 13.2  0.9  0.8  0.5  0.2  0.4  1.2
## [113]  0.6  0.2  1.0  0.4  0.6  0.1  0.4  0.7  0.5  0.7  0.9  4.8  3.8  4.9
## [127]  6.2 10.4 16.4  2.7  1.2  0.5  1.9  2.0  2.1  1.9  1.8  3.5  7.7  2.7
## [141]  1.8  0.8  0.5  0.5  0.3  0.9
## attr(,"na.action")
##  [1]   9  10  15  21  24  45  49  58  61  63  80  81  89 110 112 115 121
## [18] 129 138 150
## attr(,"class")
## [1] "omit"
```

Although the result does contain more than just the non-NA values, only the non-NA values will be used in subsequent operations. Note that the result of na.omit contains more information than just the non-NA values. This function can also be applied to complete data frames. In this case, any row with an NA is removed (so be careful with its usage).

```
soil.data.cleaned <- na.omit(soil.data)
```

It is often necessary to identify NAs present in a data structure. The is.na function can be used for this—it can also be negated using the "!" character.

```
is.na(soil.data$ESP)
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
##  [12] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
##  [23] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [45]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
##  [56] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78] FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [89]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [111] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
## [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [155] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [166] FALSE
```

2.4.4 Writing Data to Files

With R, it is easy to write data to files. The function write.table is usually the best function for this purpose. Given only a data frame and a file name, this function will write the data contained in the data frame to a text file. There are a number of arguments that can be controlled with this function as shown below (also look at the help file).
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ", eol = "\", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE, qmethod = c("escape", "double"), fileEncoding = "")

The important ones (or most frequently used) are the column separator sep argument, and whether of not you want to keep the column and row names (col.names and row.names respectively). For example, if we want to write soil.data to text file ("file name.txt"), retaining the column names, not retaining row names, and having a tab delimited column separator, we would use:

```
write.table(soil.data, file = "file name.txt", col.names = TRUE,
row.names = FALSE, sep = "\t")
```

Setting the `append` argument to `TRUE` lets you add data to a file that already exists.

The `write.table` function can not be used with all data structures in `R` (like lists for example). However, it can be used for such things as vectors and matrices.

2.4.5 *Exercises*

1. Using the `soil.data` object, determine the minimum and maximum soil pH (`PH_CaCl2`) in the data frame. Next add a new column to the dataframe that contains the log_10 of soil carbon `Total_Carbon`.
2. Create a new data frame that contains the mean SOC, pH, and clay of the data set. Write out the summary to a new file using the default options. Finally, try changing the separator to a tab and write to a new file.
3. There are a number of `NA` values in the data set. We want to remove them. Could this be done in one step i.e., delete every row that contains an `NA`? Is this appropriate? How would you go about ensuring that no data is lost? Can you do this? or perhaps—do this!

2.5 Graphics: The Basics

2.5.1 *Introduction to the `Plot` Function*

It is easy to produce publication-quality graphics in `R`. There are many excellent `R` packages at your finger tips to do this; some of which include `lattice` and `ggplot2` (see the help files and documentation for these). While in the course of this book we will revert to using these "high end" plotting packages, some fundamentals of plotting need to bedded down. Therefore in this section we will focus on the simplest plots—those which can be produced using the `plot` function, which is a base function that come with `R`. This function produces a plot as a side effect, but the type of plot produced depends on the type of data submitted. The basic plot arguments, as given in the help file for `plot.default` are:

```
plot(x, y = NULL, type = ''p'',  xlim = NULL, ylim = NULL, log =
'''', main = NULL, sub = NULL, xlab = NULL, ylab = NULL, ann =
par(''ann''), axes = TRUE, frame.plot = axes, panel.first = NULL,
panel.last = NULL, asp = NA, ...)
```

To plot a single vector, all we need to do is supply that vector as the only argument to the function. This plot is shown in Fig. 2.3.

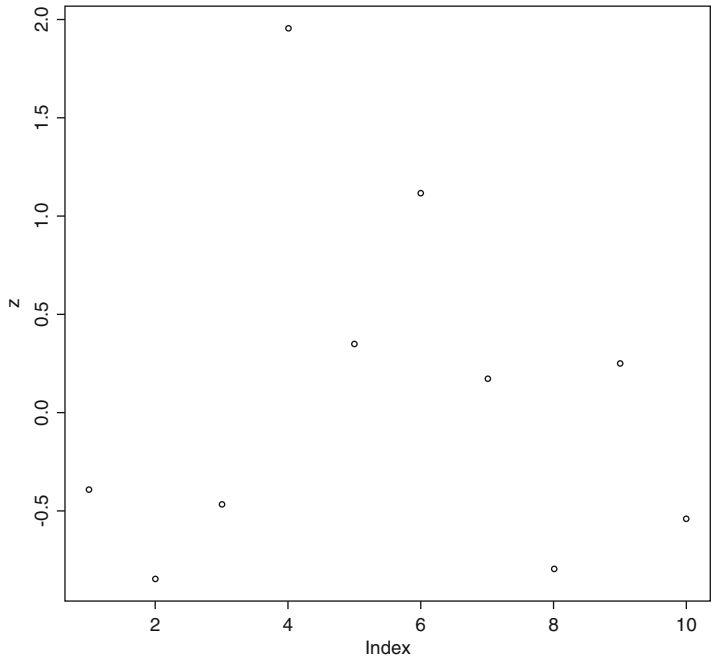

Fig. 2.3 Your first plot

```
z <- rnorm(10)
plot(z)
```

In this case, R simply plots the data in the order they occur in the vector. To plot one variable versus another, just specify the two vectors for the first two arguments. (see Fig. 2.4)

```
x <- -15:15
y <- x^2
plot(x, y)
```

And this is all it takes to generate plots in R, as long as you like the default settings. Of course, the default settings generally will not be sufficient for publication- or presentation-quality graphics. Fortunately, plots in R are very flexible. The table below shows some of the more common arguments to the `plot` function, and some of the common settings. For many more arguments, see the help file for `par` or consult some online materials where http://www.statmethods.net/graphs/ is a useful starting point.

Use of some of the arguments in Table 2.1 is shown in the following example (Fig. 2.5).

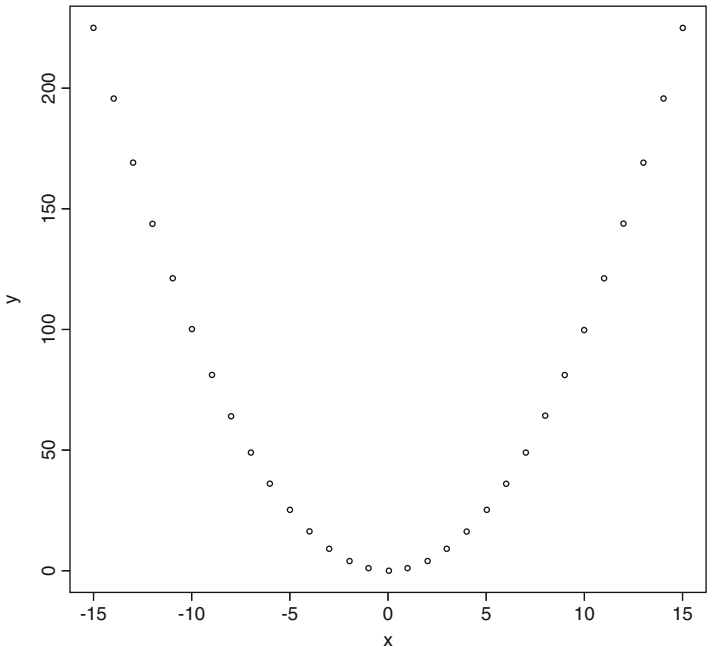

Fig. 2.4 Your second plot

```
plot(x, y, type = "o", xlim = c(-20, 20), ylim = c(-10, 300), pch = 21,
col = "red", bg = "yellow", xlab = "The X variable", ylab = "X squared")
```

The `plot` function is effectively vectorised. It accepts vectors for the first two arguments (which specify the x and y position of your observations), but can also accept vectors for some of the other arguments, including `pch` or `col`. Among other things, this provides an easy way to produce a reference plot demonstrating R's plotting symbols and lines. If you use R regularly, you may want to print a copy out (or make your own)—see Fig. 2.6.

```
plot(1:25, rep(1, 25), pch = 1:25, ylim = c(0, 10),
xlab = "", ylab = "",
axes = FALSE) text(1:25, 1.8, as.character(1:25), cex = 0.7)
text(12.5, 2.5, "Default", cex = 0.9)
points(1:25, rep(4, 25), pch = 1:25, col = "blue")
text(1:25, 4.8, as.character(1:25), cex = 0.7, col = "blue")
text(12.5, 5.5, "Blue", cex = 0.9, col = "blue")
points(1:25, rep(7, 25), pch = 1:25, col = "blue", bg = "red")
text(1:25, 7.8, as.character(1:25), cex = 0.7, col = "blue")
text(10, 8.5, "Blue", cex = 0.9, col = "blue")
text(15, 8.5, "Red", cex = 0.9, col = "red")
box()
```

Table 2.1 Some of the more commonly used `plot` arguments

Argument	Common options	Additional information

`col`	`"red"`
	`"blue"`
	`1`
	through
	`657`
		Colour of plotting symbols and lines. Type `colors()` to get list. You can also mix your own colours. See "color specification" in the help file for `par` or http://research.stowers-institute.org/efg/R/Color/Chart/

`bg`	`"red"`
	`"blue"`
	many more
		Colour of fill for some plotting symbols (see below)

`las`	`0`
	`1`
	`2`
	`3`
		Rotation of numeric axis labels

`main`	`Any character string,` `e.g.,"plot 1"`	Adds a main title at the top of the plot

`log`	`"x"`
	`"y"`
	`"xy"`
		For making logarithmic scaled axes

`lty`	`0`
	`1` or `"solid"`
	`2` or `"dashed"`
	`3` or `"dashed"`
	through
	`6`
		Line types

`pch`	`0`
	through
	`25`
		Plotting symbols. See below for symbols. Can also use any single character, e.g., "v", or "X" etc

`type`	`"p"` for points `"l"` for line `"b"` for both `"o"` for over `"n"` for none	`"n"` can be handy for setting up a plot that you later add data to
`xlab, ylab`	Any character string, e.g.,"soil depth"	For specifying axis labels

`xlim, ylim`	Any two element vector,
	e.g., `c(0-100)`
	`c(-10-10)`
	`c(55-0)`
		List higher value first to reverse axis

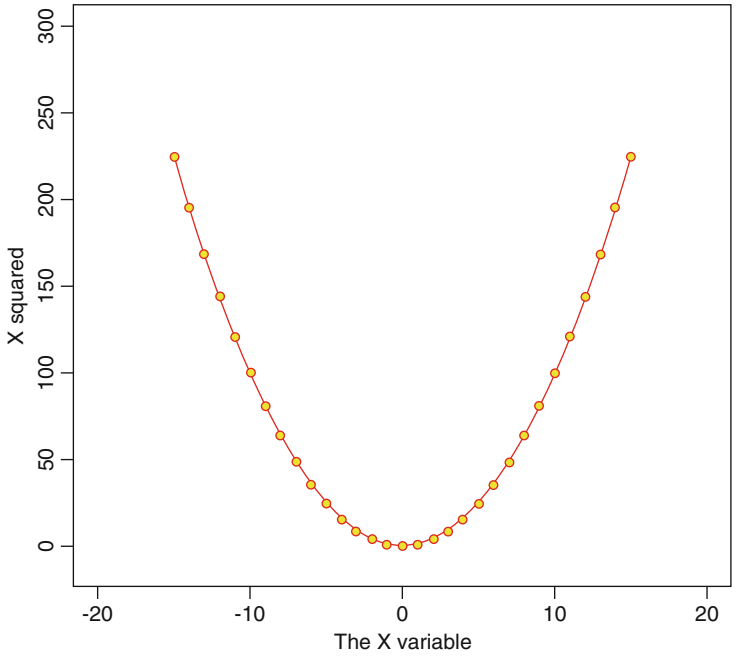

Fig. 2.5 Your first plot using some of the `plot` arguments

2.5.2 Exercises

1. Produce a data frame with two columns: x, which ranges from -2π to 2π and has a small interval between values (for plotting), and cosine(x). Plot the cosine(x) vs. x as a line. Repeat, but try some different line types or colours.
2. Read in the data from the `ithir` package called `"USYD_dIndex"`, which contains some observed soil drainage characteristics based on some defined soil colour and drainage index (first column). In the second column is a corresponding prediction which was made by a soil spatial prediction function. Plot the observed drainage index (`DI_observed`) vs. the predicted drainage index (`DI_predicted`). Ensure your plot has appropriate axis limits and labels, and a heading. Try a few plotting symbols and colours. Add some informative text somewhere. If you feel inspired, draw a line of concordance i.e., a 1:1 line on the plot.

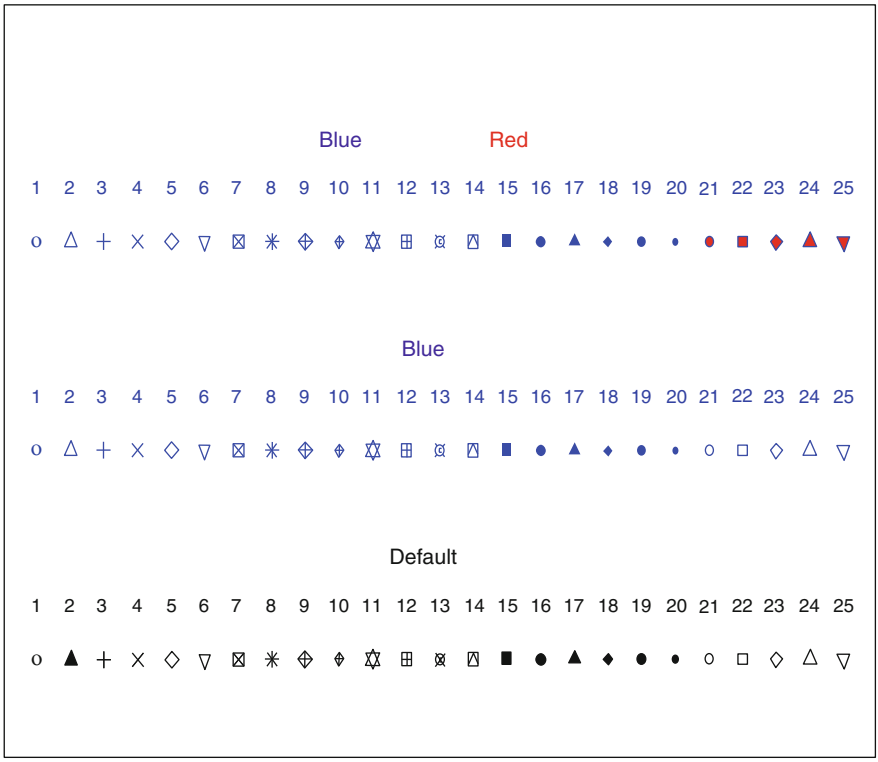

Fig. 2.6 Illustration of some of the `plot` arguments and symbols

2.6 Manipulating Data

2.6.1 Modes, Classes, Attributes, Length, and Coercion

As described before, the mode of an object describes the type of data that it contains. In R, mode is an object attribute. All objects have at least two attributes: mode and length, but may objects have more.

```
x <- 1:10
mode(x)
```

```
## [1] "numeric"
```

```
length(x)
```

```
## [1] 10
```

It is often necessary to change the mode of a data structure, e.g., to have your data displayed differently, or to apply a function that only works with a particular

type of data structure. In R this is called coercion. There are many functions in R that have the structure as.something that change the mode of a submitted object to "something". For example, say you want to treat numeric data as character data.

```
x <- 1:10
as.character(x)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Or, you may want to turn a matrix into a data frame.

```
X <- matrix(1:30, nrow = 3)
as.data.frame(X)
```

```
##   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
## 1  1  4  7 10 13 16 19 22 25  28
## 2  2  5  8 11 14 17 20 23 26  29
## 3  3  6  9 12 15 18 21 24 27  30
```

If you are unsure of whether or not a coercion function exists, give it a try—two other common examples are as.numeric and as.vector.

Attributes are important internally for determining how objects should be handled by various functions. In particular, the class attribute determines how a particular object will be handled by a given function. For example, output from a linear regression has the class "lm" and will be handled differently by the print function than will a data frame, which has the class "data.frame". The utility of this object-orientated approach will become more apparent later on.

It is often necessary to know the length of an object. Of course, length can mean different things. Three useful functions for this are nrow, NROW, and length.

The function nrow will return the number of rows in a two-dimensional data structure.

```
X <- matrix(1:30, nrow = 3)
X
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    4    7   10   13   16   19   22   25    28
## [2,]    2    5    8   11   14   17   20   23   26    29
## [3,]    3    6    9   12   15   18   21   24   27    30
```

```
nrow(X)
```

```
## [1] 3
```

The vertical analog is ncol.

```
ncol(X)
```

```
## [1] 10
```

You can get both of these at once with the dim function.

```
dim(X)
```

```
## [1]  3 10
```

For a vector, use the function NROW or length.

```
x <- 1:10
NROW(x)
```

```
## [1] 10
```

The value returned from the function length depends on the type of data structure you submit, but for most data structures, it is the total number of elements.

```
length(X)
```

```
## [1] 30
```

```
length(x)
```

```
## [1] 10
```

2.6.2 *Indexing, Sub-setting, Sorting, and Locating Data*

Sub-setting and indexing are ways to select specific parts of the data structure (such as specific rows within a data frame) within R. Indexing (also know as sub-scripting) is done using the square braces in R:

```
v1 <- c(5, 1, 3, 8)
v1
```

```
## [1] 5 1 3 8
```

```
v1[3]
```

```
## [1] 3
```

R is very flexible in terms of what can be selected or excluded. For example, the following returns the 1st through 3rd observation:

```
v1[1:3]
```

```
## [1] 5 1 3
```

While this returns all but the 4th observation:

```
v1[-4]
```

```
## [1] 5 1 3
```

This bracket notation can also be used with relational constraints. For example, if we want only those observations that are <5.0:

```
v1[v1 < 5]
```

```
## [1] 1 3
```

This may seem confusing, but if we evaluate each piece separately, it becomes more clear:

```
v1 < 5
```

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

```
v1[c(FALSE, TRUE, TRUE, FALSE)]
```

```
## [1] 1 3
```

While we are on the topic of subscripts, we should noted that, unlike some other programming languages, the size of a vector in `R` is not limited by its initial assignment. This is true for other data structures as well. To increase the size of a vector, just assign a value to a position that does not currently exist:

```
length(v1)
```

```
## [1] 4
```

```
v1[8] <- 10
length(v1)
```

```
## [1] 8
```

```
v1
```

```
## [1]  5  1  3  8 NA NA NA 10
```

Indexing can be applied to other data structures in a similar manner as shown above. For data frames and matrices, however, we are now working with two dimensions. In specifying indices, row numbers are given first. We will use our `soil.data` set to illustrate the following few examples:

```
library(ithir)
data(USYD_soil1)
soil.data <- USYD_soil1
dim(soil.data)
```

```
## [1] 166  16
```

```
str(soil.data)
```

```
## 'data.frame': 166 obs. of  16 variables:
##  $ PROFILE    : int  1 1 1 1 1 1 2 2 2 2 ...
##  $ Landclass  : Factor w/ 4 levels "Cropping","Forest",..: 4 4 4 ...
```

```
##  $ Upper.Depth : num  0 0.02 0.05 0.1 0.2 0.7 0 0.02 0.05 0.1 ...
##  $ Lower.Depth : num  0.02 0.05 0.1 0.2 0.3 0.8 0.02 0.05 0.1 0.2 ...
##  $ clay        : int  8 8 8 8 NA 57 9 9 9 NA ...
##  $ silt        : int  9 9 10 10 10 8 10 10 10 10 ...
##  $ sand        : int  83 83 82 83 79 36 81 80 80 81 ...
##  $ pH_CaCl2    : num  6.35 6.34 4.76 4.51 4.64 6.49 5.91 ...
##  $ Total_Carbon: num  1.07 0.98 0.73 0.39 0.23 0.35 1.14 ...
##  $ EC          : num  0.168 0.137 0.072 0.034 NA 0.059 0.123 ...
##  $ ESP         : num  0.3 0.5 0.9 0.2 0.9 0.3 0.3 0.6 NA NA ...
##  $ ExchNa      : num  0.01 0.02 0.02 0 0.02 0.04 0.01 0.02 NA NA ...
##  $ ExchK       : num  0.71 0.47 0.52 0.38 0.43 0.46 0.7 0.56 NA NA ...
##  $ ExchCa      : num  3.17 3.5 1.34 1.03 1.5 9.13 2.92 3.2 NA NA ...
##  $ ExchMg      : num  0.59 0.6 0.22 0.22 0.5 5.02 0.51 0.5 NA NA ...
##  $ CEC         : num  5.29 3.7 2.86 2.92 2.6 ...
```

If we want to subset out only the first 5 rows, and the first 2 columns:

```
soil.data[1:5, 1:2]
```

```
##   PROFILE      Landclass
## 1       1 native pasture
## 2       1 native pasture
## 3       1 native pasture
## 4       1 native pasture
## 5       1 native pasture
```

If an index is left out, R returns all values in that dimension (you need to include the comma).

```
soil.data[1:2, ]
```

```
##   PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 1       1 native pasture        0.00        0.02    8    9   83     6.35
## 2       1 native pasture        0.02        0.05    8    9   83     6.34
##   Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg  CEC
## 1         1.07 0.168 0.3   0.01  0.71   3.17   0.59 5.29
## 2         0.98 0.137 0.5   0.02  0.47   3.50   0.60 3.70
```

You can also specify row or column names directly within the brackets—this can be very handy when column order may change in future versions of your code.

```
soil.data[1:5, "Total_Carbon"]
```

```
## [1] 1.07 0.98 0.73 0.39 0.23
```

You can also specify multiple column names using the c function.

```
soil.data[1:5, c("Total_Carbon", "CEC")]
```

```
##   Total_Carbon  CEC
## 1         1.07 5.29
## 2         0.98 3.70
## 3         0.73 2.86
```

```
## 4          0.39 2.92
## 5          0.23 2.60
```

Relational constraints can also be used in indexes. Lets subset out the soil observations that are extremely sodic i.e an ESP greater than 10 %.

```
na.omit(soil.data[soil.data$ESP > 10, ])
```

```
##     PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 68       12         Forest        0.02        0.05   22   13   64     4.65
## 111      19 native pasture        0.16        0.26   32   13   56     6.10
## 113      20       Cropping        0.00        0.02    9    9   81     4.64
## 117      20       Cropping        0.30        0.40   37    7   56     6.30
## 118      20       Cropping        0.70        0.80   20   14   67     7.17
## 123      21       Cropping        0.25        0.35   25   16   59     5.05
## 147      26       Cropping        0.15        0.24   51    8   42     6.27
## 148      26       Cropping        0.70        0.80   50    9   40     7.81
##     Total_Carbon    EC  ESP ExchNa ExchK ExchCa ExchMg   CEC
## 68          1.49 0.499 13.0   1.00  0.74   2.17   3.76  6.85
## 111         0.50 0.223 17.4   2.62  0.30   3.74   8.37 11.97
## 113         1.08 0.301 11.1   0.31  0.87   1.01   0.63  3.02
## 117         0.32 0.214 12.1   1.66  0.27   4.19   7.61 12.44
## 118         0.09 0.292 21.2   2.88  0.33   2.86   7.50 10.23
## 123         0.25 0.073 13.2   0.95  0.30   2.00   3.92  5.29
## 147         0.63 0.134 10.4   2.09  0.74   5.09  12.23 14.29
## 148         0.84 0.820 16.4   4.93  0.91   7.52  16.72 23.34
```

While indexing can clearly be used to create a subset of data that meet certain criteria, the `subset` function is often easier and shorter to use for data frames. Sub-setting is used to select a subset of a vector, data frame, or matrix that meets a certain criterion (or criteria). To return what was given in the last example.

```
subset(soil.data, ESP > 10)
```

```
##     PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 68       12         Forest        0.02        0.05   22   13   64     4.65
## 111      19 native pasture        0.16        0.26   32   13   56     6.10
## 113      20       Cropping        0.00        0.02    9    9   81     4.64
## 117      20       Cropping        0.30        0.40   37    7   56     6.30
## 118      20       Cropping        0.70        0.80   20   14   67     7.17
## 123      21       Cropping        0.25        0.35   25   16   59     5.05
## 147      26       Cropping        0.15        0.24   51    8   42     6.27
## 148      26       Cropping        0.70        0.80   50    9   40     7.81
##     Total_Carbon    EC  ESP ExchNa ExchK ExchCa ExchMg   CEC
## 68          1.49 0.499 13.0   1.00  0.74   2.17   3.76  6.85
## 111         0.50 0.223 17.4   2.62  0.30   3.74   8.37 11.97
## 113         1.08 0.301 11.1   0.31  0.87   1.01   0.63  3.02
## 117         0.32 0.214 12.1   1.66  0.27   4.19   7.61 12.44
## 118         0.09 0.292 21.2   2.88  0.33   2.86   7.50 10.23
## 123         0.25 0.073 13.2   0.95  0.30   2.00   3.92  5.29
## 147         0.63 0.134 10.4   2.09  0.74   5.09  12.23 14.29
## 148         0.84 0.820 16.4   4.93  0.91   7.52  16.72 23.34
```

Note that the `$` notation does not need to be used in the `subset` function, As with indexing multiple constraints can also be used:

```
subset(soil.data, ESP > 10 & Lower.Depth > 0.3)
```

```
##     PROFILE Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 117      20  Cropping        0.30        0.40   37    7   56      6.30
## 118      20  Cropping        0.70        0.80   20   14   67      7.17
## 123      21  Cropping        0.25        0.35   25   16   59      5.05
## 148      26  Cropping        0.70        0.80   50    9   40      7.81
##     Total_Carbon    EC  ESP ExchNa ExchK ExchCa ExchMg   CEC
## 117         0.32 0.214 12.1   1.66  0.27   4.19   7.61 12.44
## 118         0.09 0.292 21.2   2.88  0.33   2.86   7.50 10.23
## 123         0.25 0.073 13.2   0.95  0.30   2.00   3.92  5.29
## 148         0.84 0.820 16.4   4.93  0.91   7.52  16.72 23.34
```

In some cases you may want to select observations that include any one value out of a set of possibilities. Say we only want those observations where `Landclass` is `native pasture` or `forest`. We could use:

```
subset(soil.data, Landclass == "Forest" | Landclass == "native pasture")
```

But, this is an easier way (we are using the `head` function just to limit the number of outputted rows. So try it without the `head` function).

```
head(subset(soil.data, Landclass %in% c("Forest", "native pasture")))
```

```
##   PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 1       1 native pasture        0.00        0.02    8    9   83      6.35
## 2       1 native pasture        0.02        0.05    8    9   83      6.34
## 3       1 native pasture        0.05        0.10    8   10   82      4.76
## 4       1 native pasture        0.10        0.20    8   10   83      4.51
## 5       1 native pasture        0.20        0.30   NA   10   79      4.64
## 6       1 native pasture        0.70        0.80   57    8   36      6.49
##   Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg   CEC
## 1         1.07 0.168 0.3   0.01  0.71   3.17   0.59  5.29
## 2         0.98 0.137 0.5   0.02  0.47   3.50   0.60  3.70
## 3         0.73 0.072 0.9   0.02  0.52   1.34   0.22  2.86
## 4         0.39 0.034 0.2   0.00  0.38   1.03   0.22  2.92
## 5         0.23    NA 0.9   0.02  0.43   1.50   0.50  2.60
## 6         0.35 0.059 0.3   0.04  0.46   9.13   5.02 14.96
```

Both of the above methods produce the same result, so it just comes down to a matter of efficiency.

Indexing matrices and arrays follows what we have just covered. For example:

```
X <- matrix(1:30, nrow = 3)
X
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    4    7   10   13   16   19   22   25    28
## [2,]    2    5    8   11   14   17   20   23   26    29
## [3,]    3    6    9   12   15   18   21   24   27    30
```

```
X[3, 8]
```

```
## [1] 24
```

```
X[, 3]
```

```
## [1] 7 8 9
```

```
Y <- array(1:90, dim = c(3, 10, 3))
Y[3, 1, 1]
```

```
## [1] 3
```

Indexing is a little trickier for lists—you need to use double square braces, `[[i]]`, to specify an element within a list. Of course, if the element within the list has multiple elements, you could use indexing to select specific elements within it.

```
list.1 <- list(1:10, X, Y)
list.1[[1]]
```

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

It is also possible to use double, triple, etc. indexing with all types of data structures. `R` evaluates the expression from left to right. As a simple example, lets extract the element on the third row of the second column of the second element of `list.1`:

```
list.1[[2]][3, 2]
```

```
## [1] 6
```

An easy way to divide data into groups is to use the `split` function. This function will divide a data structure (typically a vector or a data frame) into one subset for each level of the variable you would like to split by. The subsets are stored together in a list. Here we split our `soil.data` set into the separate or individual soil profile (splitting by the `PROFILE` column—note output is not shown here for sake of brevity).

```
soil.data.split <- split(soil.data, soil.data$PROFILE)
```

If you apply `split` to individual vectors, the resulting list can be used directly in some plotting or summarizing functions to give you results for each separate group. (There are usually other ways to arrive at this type of result). The `split` function can also be handy for manipulating and analyzing data by some grouping variable, as we will see later.

It is often necessary to sort data. For a single vector, this is done with the function `sort`.

```
x <- rnorm(5)
x
```

```
## [1] -1.1915609  1.3808311  0.9079993  0.2527144 -1.5155076
```

```
y <- sort(x)
y
```

```
## [1] -1.5155076 -1.1915609  0.2527144  0.9079993  1.3808311
```

But what if you want to sort an entire data frame by one column? In this case it is necessary to use the function `order`, in combination with indexing.

```
head(soil.data[order(soil.data$clay), ])
```

```
##     PROFILE        Landclass Upper.Depth Lower.Depth clay silt sand
## 116      20         Cropping        0.10        0.30    5   11   85
## 1         1   native pasture        0.00        0.02    8    9   83
## 2         1   native pasture        0.02        0.05    8    9   83
## 3         1   native pasture        0.05        0.10    8   10   82
## 4         1   native pasture        0.10        0.20    8   10   83
## 7         2 improved pasture        0.00        0.02    9   10   81
##     pH_CaCl2 Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg  CEC
## 116      5.33         0.24 0.033 4.0   0.10  0.18   1.40   0.70 1.90
## 1        6.35         1.07 0.168 0.3   0.01  0.71   3.17   0.59 5.29
## 2        6.34         0.98 0.137 0.5   0.02  0.47   3.50   0.60 3.70
## 3        4.76         0.73 0.072 0.9   0.02  0.52   1.34   0.22 2.86
## 4        4.51         0.39 0.034 0.2   0.00  0.38   1.03   0.22 2.92
## 7        5.91         1.14 0.123 0.3   0.01  0.70   2.92   0.51 3.59
```

The function `order` returns a vector that contains the row positions of the ranked data:

```
order(soil.data$clay)
```

```
##   [1] 116   1   2   3   4   7   8   9  16 113 114 115  14  31  32  61  62
##  [18]  64  45 126 149 150  37  38  39 101 102 103 107 108 151  34  42  44
##  [35]  77  13  43  46  50 109 125  78 144  35  51  79  96 110 119 120 121
##  [52] 127 134  67  97 130 135 145 161  36  98 136 140 146 160 162  17  52
##  [69]  83 104 118 131 139 141  65  84  85  68  69  89  99  53 156  72 123
##  [86] 137  54  56  80  95 105  57  90  74 132  58  81  91 128  20  55 111
## [103] 142  86 163 106 154  11 129  18  87 112 117  25  75  21 155  66  92
## [120]  22  26 152 164 133 138  47  71  41  59  93  60 148  24 147  12 158
## [137] 165   6  82 166  48  88 159  28  29  94  76 100  40   5  10  15  19
## [154]  23  27  30  33  49  63  70  73 122 124 143 153 157
```

The previous discussion in this section showed how to isolate data that meet certain criteria from a data structure. But sometimes it is important to know where data resides in its original data structure. But sometimes it is important to know where data resides in its original data structure. To functions that are handy for locating data within an `R` data structure are `match` and `which`. The `match` function will tell you where specific values reside in a data structure, while the `which` function will return the locations of values that meet certain criteria.

```
match(c(25.85, 11.45, 9.23), soil.data$CEC)
```

```
## [1] 41 59 18
```

and to check the result...

```
soil.data[c(41, 59, 18), ]
```

```
##    PROFILE        Landclass Upper.Depth Lower.Depth clay silt sand
## 41       7         Cropping         0.7         0.8   47   11   42
## 59      10 improved pasture         0.1         0.2   47   12   42
## 18       3           Forest         0.7         0.8   37    9   54
##    pH_CaCl2 Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg   CEC
## 41     6.70         0.23 0.063 2.2   0.61  0.53  14.22  12.95 25.85
## 59     5.94         0.70 0.039 0.1   0.01  0.74   6.76   2.61 11.45
## 18     6.05         0.17 0.088 0.7   0.06  0.33   6.15   2.35  9.23
```

Note that the `match` function matches the first observation only (this makes it difficult to use when there are multiple observations of the same value). This function is vectorised. The `match` function is useful for finding the location of the unique values, such as the maximum.

```
match(max(soil.data$CEC, na.rm = TRUE), soil.data$CEC)
```

```
## [1] 95
```

Note the call to the `na.rm` argument in the `max` function as a means to overlook the presence of `NA` values. So what is the maximum CEC value in our `soil.data` set.

```
soil.data$CEC[95]
```

```
## [1] 28.21
```

The `which` function, on the other hand, will return all locations that meet the criteria.

```
which(soil.data$ESP > 5)
```

```
##  [1]  68 101 102 103 104 107 108 109 111 113 114 117 118 123 146 147 148
## [18] 159
```

Of course, you can specify multiple constraints.

```
which(soil.data$ESP > 5 & soil.data$clay > 30)
```

```
## [1] 111 117 147 148 159
```

The `which` function can also be useful for locating missing values.

```
which(is.na(soil.data$ESP))
```

```
##  [1]   9  10  15  21  24  45  49  58  61  63  80  81  89 110 112 115 121
## [18] 129 138 150
```

```
soil.data$ESP[c(which(is.na(soil.data$ESP)))]
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

2.6.3 Factors

For many analyses, it is important to distinguish between quantitative (i.e., continuous) and categorical (i.e., discrete) variables. Categorical data are called factors in R. Internally, factors are stored as numeric data (as a check with `mode` will tell you), but they are handled as categorical data in statistical analyses. Factors are a class of data in R. R automatically recognizes non-numerical data as factors when the data are read in, but if numerical data are to be used as a factor (or if character data are generated within R and not read in, conversion to a factor must be specified explicitly. In R, the function `factor` does this.

```
a <- c(rep(0, 4), rep(1, 4))
a
```

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

```
a <- factor(a)
a
```

```
## [1] 0 0 0 0 1 1 1 1
## Levels: 0 1
```

The levels that R assigns to your factor are by default the unique values given in your original vector. This is often fine, but you may want to assign more meaningful levels. For example, say you have a vector that contains soil drainage class categories.

```
soil.drainage<- c("well drained", "imperfectly drained", "poorly drained",
"poorly drained", "well drained", "poorly drained")
```

If you designate this as a factor, the default levels will be sorted alphabetically.

```
soil.drainage1 <- factor(soil.drainage)
soil.drainage1
```

```
## [1] well drained        imperfectly drained poorly drained
## [4] poorly drained      well drained        poorly drained
## Levels: imperfectly drained poorly drained well drained
```

```
as.numeric(soil.drainage1)
```

```
## [1] 3 1 2 2 3 2
```

If you specify `levels` as an argument of the `factor` function, you can control the order of the levels.

```
soil.drainage2 <- factor(soil.drainage, levels = c("well drained",
"imperfectly drained", "poorly drained"))
as.numeric(soil.drainage2)
```

```
## [1] 1 2 3 3 1 3
```

This can be useful for obtaining a logical order in statistical output or summaries.

2.6.4 Combining Data

Data frames (or vectors or matrices) often need to be combined for analysis or plotting. Three R functions that are very useful for combining data are `rbind` and `cbind`. The function `rbind` simply "stacks" objects on top of each other to make a new object ("row bind"). The function `cbind` ("column bind") carries out an analogous operation with columns of data.

```
soil.info1 <- data.frame(soil = c("Vertosol", "Hydrosol", "Sodosol"),
response = 1:3)
soil.info1
```

```
##       soil response
## 1 Vertosol        1
## 2 Hydrosol        2
## 3  Sodosol        3
```

```
soil.info2 <- data.frame(soil = c("Chromosol", "Dermosol", "Tenosol"),
response = 4:6)
soil.info2
```

```
##        soil response
## 1 Chromosol        4
## 2  Dermosol        5
## 3   Tenosol        6
```

```
soil.info <- rbind(soil.info1, soil.info2)
soil.info
```

```
##        soil response
## 1  Vertosol        1
## 2  Hydrosol        2
## 3   Sodosol        3
## 4 Chromosol        4
## 5  Dermosol        5
## 6   Tenosol        6
```

```
a.column <- c(2.5, 3.2, 1.2, 2.1, 2, 0.5)
soil.info3 <- cbind(soil.info, SOC = a.column)
soil.info3
```

```
##        soil response SOC
## 1  Vertosol        1 2.5
## 2  Hydrosol        2 3.2
## 3   Sodosol        3 1.2
## 4 Chromosol        4 2.1
## 5  Dermosol        5 2.0
## 6   Tenosol        6 0.5
```

2.6.5 Exercises

1. Using the `soil.data` set, return the following:
   (a) The first 10 rows of the columns `clay`, `Total_Carbon`, and `ExchK`
   (b) The column `CEC` for the `Forest` land class
   (c) Use the `subset` function to create a new data frame that has only data for the cropping land class
   (d) How many soil profiles are there? Actually write some script to determine this rather than look at the data frame
2. Using the same data set, find the location and value of the maximum, minimum and median soil carbon (`Total_Carbon`)
3. Make a new data frame which is sorted by the upper soil depth (`Upper.Depth`). Can you sort it in decreasing order (Hint: Check the help file)
4. Make a new data frame which contains the columns `PROFILE`, `Landclass`, `Upper.Depth`, and `Lower.Depth`. Make another data frame which contains just the information regarding the exchangeable cations e.g., `ExchNa`, `ExchK`, `ExchCa`, and `ExchMg`. Make a new data frame which combines these two separate data frames together
5. Make separate data frame for each of the land classes. Now make a new data frame which combines the four separate data frames together.

2.7 Exploratory Data Analysis

2.7.1 Summary Statistics

We will again use the `soil.data` set to demonstrate calculation of summary statistics. Just for recall, lets see what is in this data frame.

```
library(ithir)
data(USYD_soil1)
soil.data <- USYD_soil1
names(soil.data)
```

```
##  [1] "PROFILE"      "Landclass"    "Upper.Depth"  "Lower.Depth"
##  [5] "clay"         "silt"         "sand"         "pH_CaCl2"
##  [9] "Total_Carbon" "EC"           "ESP"          "ExchNa"
## [13] "ExchK"        "ExchCa"       "ExchMg"       "CEC"
```

Here are some useful functions (and note the usage of the `na.rm` argument) for calculation of means (`mean`), medians (`median`), standard deviations (`sd`) and variances (`var`):

```
mean(soil.data$clay, na.rm = TRUE)
```

```
## [1] 26.95302
```

```
median(soil.data$clay, na.rm = TRUE)
```

```
## [1] 21
```

```
sd(soil.data$clay, na.rm = TRUE)
```

```
## [1] 15.6996
```

```
var(soil.data$clay, na.rm = TRUE)
```

```
## [1] 246.4775
```

R has a built-in function for summarizing vectors or data frames called summary. This function is a generic function—what it returns is dependent on the type of data set to it. Applying the summary function to the first 6 columns in the soil.data set results in the following output:

```
summary(soil.data[, 1:6])
```

```
##     PROFILE                 Landclass   Upper.Depth      Lower.Depth
##  Min.   : 1.00   Cropping        :49   Min.   :0.0000   Min.   :0.0200
##  1st Qu.: 8.00   Forest          :50   1st Qu.:0.0200   1st Qu.:0.0500
##  Median :15.00   improved pasture:35   Median :0.0500   Median :0.1000
##  Mean   :14.73   native pasture  :32   Mean   :0.1816   Mean   :0.2464
##  3rd Qu.:22.00                         3rd Qu.:0.2000   3rd Qu.:0.3000
##  Max.   :29.00                         Max.   :0.7000   Max.   :0.8000
##
##       clay            silt
##  Min.   : 5.00   Min.   : 6.0
##  1st Qu.:15.00   1st Qu.:11.0
##  Median :21.00   Median :15.0
##  Mean   :26.95   Mean   :16.5
##  3rd Qu.:37.00   3rd Qu.:20.0
##  Max.   :68.00   Max.   :32.0
##  NA's   :17      NA's   :1
```

Notice the difference between numerical and categorical variables. The summary function should probably be your first stop after organizing your data, and before analyzing it—it provides an easy way to check for wildly erroneous values.

2.7.2 *Histograms and Box Plots*

Box plots and histograms are simple but useful ways of summarizing data. You can generate a histogram in R using the function hist.

```
hist(soil.data$clay)
```

The histogram (Fig. 2.7) can be made to look nicer, by applying some of the plotting parameters or arguments that we covered for the plot function. There are

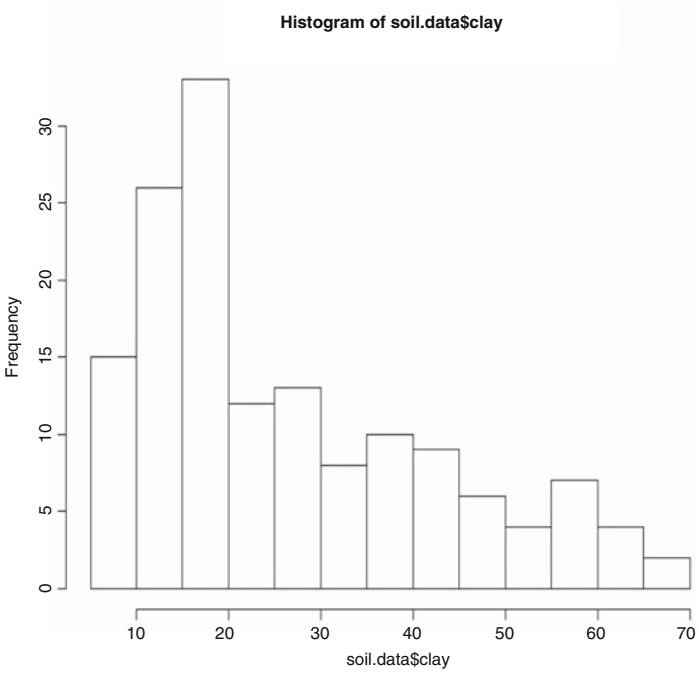

Fig. 2.7 Histogram of clay content from `soil.data`

also some additional "plotting" arguments that can be sourced in the `hist` help file. One of these is the ability to specify the number or location of breaks in the histogram.

Box plots are also a useful way to summarize data. We can use it simply, for example, summarize the clay content in the `soil.data` (Fig. 2.8).

```
boxplot(soil.data$clay)
```

By default, the heavy line shows the median, the box shows the 25th and 75th percentiles, the "whiskers" show the extreme values, and points show outliers beyond these.

Another approach is to plot a single variable by some factor. Here we will plot `Total_Carbon` by `Landclass` (Fig. 2.9).

```
boxplot(Total_Carbon ~ Landclass, data = soil.data)
```

Note the use of the tilde symbol "~" in the above command. The code `Total_Carbon~Landclass` is analogous to a model formula in this case, and simply indicates that `Total_Carbon` is described by `Landclass` and should be split up based on the category of this variable. We will see more of this character with the specification of soil spatial prediction functions later on.

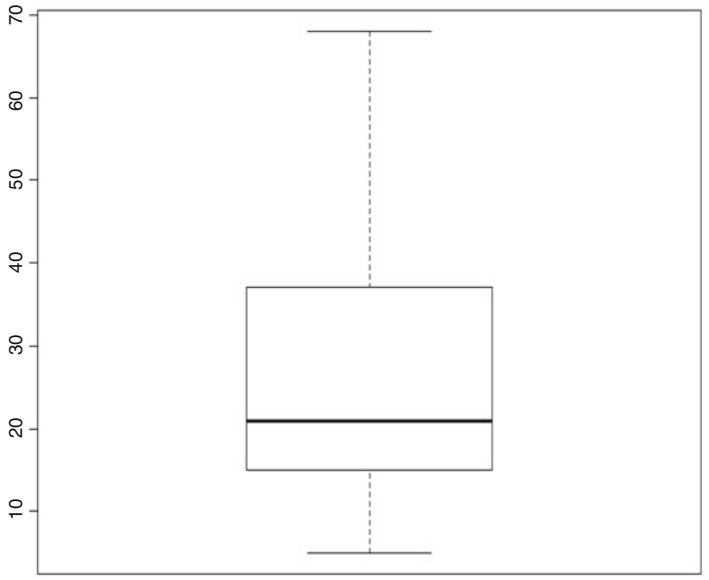

Fig. 2.8 Boxplot of clay content from `soil.data`

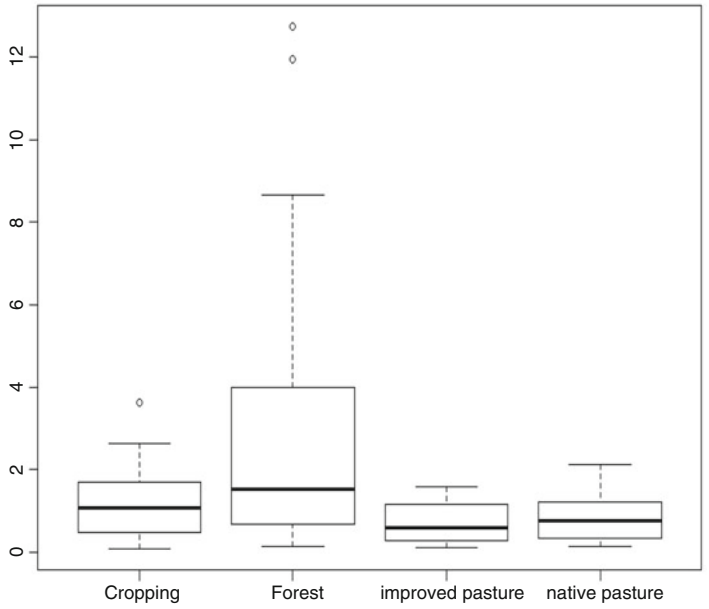

Fig. 2.9 Box plot of total carbon with respect to landclass

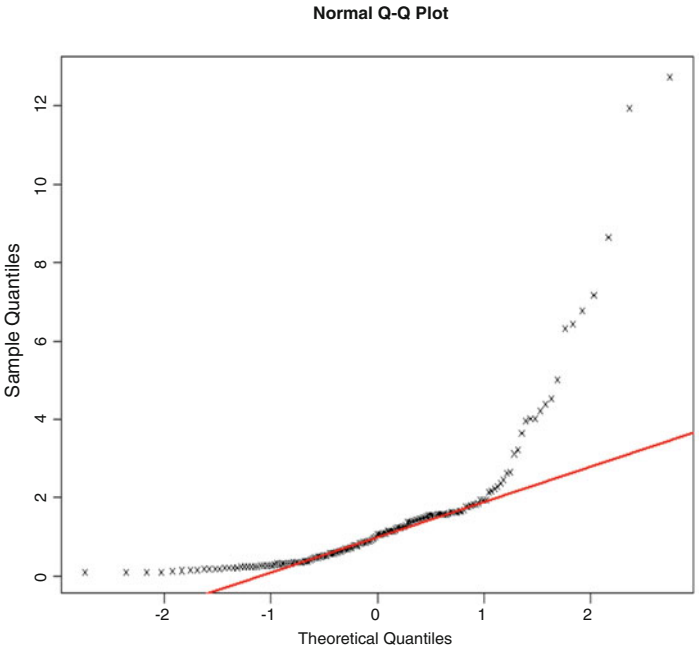

Fig. 2.10 QQ plot of total carbon in the `soil.data` set

2.7.3 Normal Quantile and Cumulative Probability Plots

One way to assess the normality of the distribution of a given variable is with a quantile-quantile plot. This plot shows data values vs. quantiles based on a normal distribution (Fig. 2.10).

```
qqnorm(soil.data$Total_Carbon, plot.it = TRUE, pch = 4, cex = 0.7)
qqline(soil.data$Total_Carbon, col = "red", lwd = 2)
```

There definitely seems to be some deviation from normality here. This is not unusual for soil carbon information. It is common (in order to proceed with statistical modelling) to perform a transformation of sorts in order to get these data to conform to a normal distribution—lets see if a log transformation works any better (Fig. 2.11).

```
qqnorm(log(soil.data$Total_Carbon), plot.it = TRUE, pch = 4, cex = 0.7)
qqline(log(soil.data$Total_Carbon), col = "red", lwd = 2)
```

Finally, another useful data exploratory tool is quantile calculations. R will return the quantiles of a given data set with the `quantile` function. Note that there are

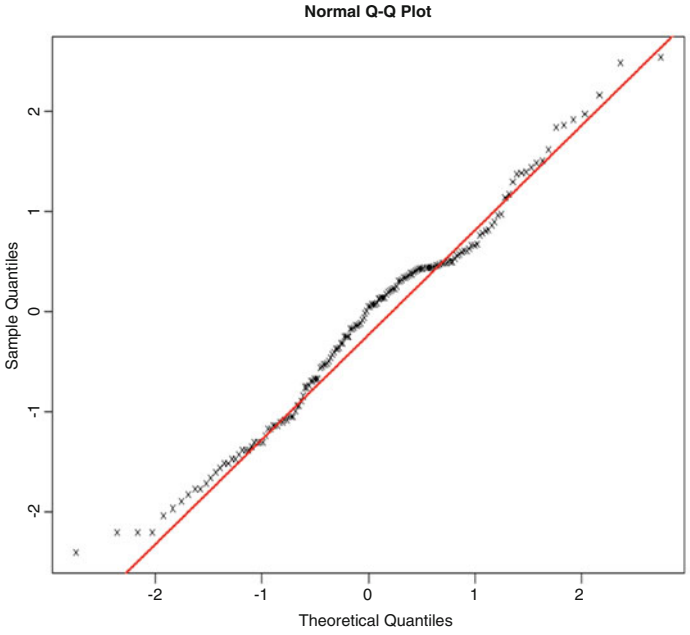

Fig. 2.11 QQ plot of log-transformed total carbon in the `soil.data` set

nine different algorithms available for doing this—you can find descriptions in the help file for `quantile`.

```
quantile(soil.data$Total_Carbon, na.rm = TRUE)
```

```
##     0%    25%    50%    75%   100%
##   0.09   0.39   1.05   1.60 12.74
```

```
quantile(soil.data$Total_Carbon, na.rm = TRUE, probs = seq(0, 1, 0.05))
```

```
##     0%     5%    10%    15%    20%    25%    30%    35%    40%    45%
##  0.090  0.170  0.230  0.270  0.328  0.390  0.502  0.604  0.730  0.866
##    50%    55%    60%    65%    70%    75%    80%    85%    90%    95%
##  1.050  1.150  1.268  1.448  1.548  1.600  1.762  2.026  2.928  4.494
##   100%
## 12.740
```

```
quantile(soil.data$Total_Carbon, na.rm = TRUE, probs = seq(0.9, 1, 0.01))
```

```
##     90%     91%     92%     93%     94%     95%     96%     97%     98%
##  2.9280  3.3208  3.9128  4.0152  4.2388  4.4940  5.5920  6.4488  7.0536
##     99%    100%
##  9.8344 12.7400
```

2.7.4 Exercises

1. Using the soil.data set firstly determine the summary statistics for each of the numerical or quantitative variables. You want to calculate things like maximum, minimum, mean, median, standard deviation, and variance. There are a couple of ways to do this. However, put all the results into a data frame and export as a text file.
2. Generate histograms and QQ plots for each of the quantitative variables. Do any need some sort of transformation so that their distribution is normal. If so, do the transformation and perform the plots again.

2.8 Linear Models: The Basics

2.8.1 The lm Function, Model Formulas, and Statistical Output

In R several classical statistical models can be implemented using one function: lm (for linear model). The lm function can be used for simple and multiple linear regression, analysis of variance (ANOVA), and analysis of covariance (ANCOVA). The help file for lm lists the following.

```
lm(formula, data, subset, weights, na.action, method
= "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
singular.ok = TRUE, contrasts = NULL, offset, ...)
```

The first argument in the lm function call (formula) is where you specify the structure of the statistical model. This approach is used in other R functions as well, such as glm, gam, and others, which we will later investigate. The most common structures of a statistical model are:

y ~ x	Simple linear regression of y on x
y ~ x + z	Multiple regression of y on x and z

There are many others, but it is these model structures that are used in one form or another for DSM.

There is some similarity between the statistical output in R and in other statistical software programs. However, by default, R usually gives only basic output. More detailed output can be retrieved with the summary function. For specific statistics, you can use "extractor" functions, such as coef or deviance. Output from the lm function is of the class lm, and both default output and specialized output from extractor functions can be assigned to objects (this is of course true for other model objects as well). This quality is very handy when writing code that uses the results of statistical models in further calculations or in compiling summaries.

2.8.2 *Linear Regression*

To demonstrate simple linear regression in R, we will again use the `soil.data` set. Here we will regress CEC content on clay.

```
summary(cbind(clay = soil.data$clay, CEC = soil.data$CEC))
```

```
##       clay            CEC
##  Min.   : 5.00   Min.   : 1.900
##  1st Qu.:15.00   1st Qu.: 5.350
##  Median :21.00   Median : 8.600
##  Mean   :26.95   Mean   : 9.515
##  3rd Qu.:37.00   3rd Qu.:12.110
##  Max.   :68.00   Max.   :28.210
##  NA's   :17      NA's   :5
```

The (alternative) hypothesis here is that clay content is a good predictor of CEC. As a start, let us have a look at what the data looks like (Fig. 2.12).

```
plot(soil.data$clay, soil.data$CEC)
```

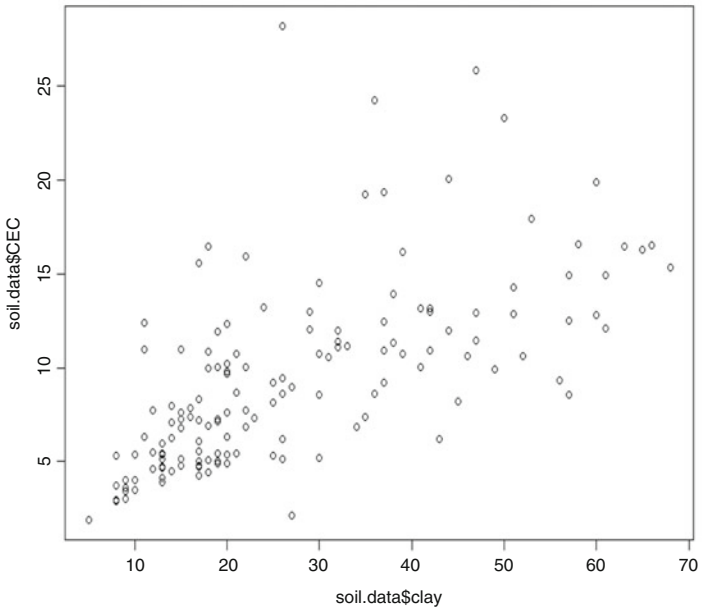

Fig. 2.12 plot of CEC against `clay` from the `soil.data` set

There appears to be some meaningful relationship. To fit a linear model, we can use the `lm` function:

```
mod.1 <- lm(CEC ~ clay, data = soil.data, y = TRUE, x = TRUE)
mod.1
```

```
##
## Call:
## lm(formula = CEC ~ clay, data = soil.data, x = TRUE, y = TRUE)
##
## Coefficients:
## (Intercept)         clay
##      3.7791       0.2053
```

R returns only the call and coefficients by default. You can get more information using the `summary` function.

```
summary(mod.1)
```

```
##
## Call:
## lm(formula = CEC ~ clay, data = soil.data, x = TRUE, y = TRUE)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.1829 -2.3369 -0.6767  1.0185 19.0924
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.77913    0.63060   5.993 1.58e-08 ***
## clay         0.20533    0.02005  10.240  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.783 on 144 degrees of freedom
##   (20 observations deleted due to missingness)
## Multiple R-squared:  0.4214,Adjusted R-squared:  0.4174
## F-statistic: 104.9 on 1 and 144 DF,  p-value: < 2.2e-16
```

Clay does appear to be a significant predictor of CEC as one would generally surmise using some basic soil science knowledge. As mentioned above, the output from the `lm` function is an object of class `lm`. These objects are lists that contain at least the following elements (you can find this list in the help file for `lm`):

`coefficients`	a named vector of model coefficients
`residuals`	model residuals (observed - predicted values)
`fitted.value`	the fitted mean values
`rank`	a numeric rank of the fitted linear model
`weights`	(only for weighted fits) the specified weights
`df.residual`	the residual degrees of freedom
`call`	the matched call

terms	the terms object used
contrasts	(only where relevant) the contrasts used
xlevels	(only where relevant) a record of the levels of the factors used in fitting
offset	the offset used (missing if none were used)
y	if requested, the response used
x	if requested, the model matrix used
model	if requested (the default), the model frame used
na.action	(where relevant) information returned by model frame on the handling of NAs

```
class(mod.1)
```

```
## [1] "lm"
```

To get at the elements listed above, you can simply index the `lm` object, i.e., call up part of the list.

```
mod.1$coefficients
```

```
## (Intercept)        clay
##   3.7791256   0.2053256
```

However, `R` has several extractor functions designed precisely for pulling data out of statistical model output. Some of the most commonly used ones are: `add1`, `alias`, `anova`, `coef`, `deviance`, `drop1`, `effects`, `family`, `formula`, `kappa`, `labels`, `plot`, `predict`, `print`, `proj`, `residuals`, `step`, `summary`, and `vcov`. For example:

```
coef(mod.1)
```

```
## (Intercept)        clay
##   3.7791256   0.2053256
```

```
head(residuals(mod.1))
```

```
##          1          2          3          4          6          7
## -0.1317300 -1.7217300 -2.5617300 -2.5017300 -0.5226822 -2.0370556
```

As mentioned before, the `summary` function is a generic function—what it does and what it returns is dependent on the class of its first argument. Here is a list of what is available from the `summary` function for this model:

```
names(summary(mod.1))
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"  "na.action"
```

To extract some of the information in the summary which is of a list structure, we would use:

```
summary(mod.1)[[4]]
```

```
##              Estimate Std. Error   t value     Pr(>|t|)
## (Intercept) 3.7791256 0.63060292  5.992877 1.577664e-08
## clay        0.2053256 0.02005048 10.240431 7.910414e-19
```

To look at the statistical summary. What is the R^2 of mod.1?

```
summary(mod.1)[["r.squared"]]
```

```
## [1] 0.4213764
```

This flexibility is useful, but makes for some redundancy in R. For many model statistics, there are three ways to get your data: an extractor function (such as coef), indexing the lm object, and indexing the summary function. The best approach is to use an extractor function whenever you can. In some cases, the summary function will return results that you can not get by indexing or using extractor functions.

Once we have fit a model in R, we can generate predicted values using the predict function.

```
head(predict(mod.1))
```

```
##         1         2         3         4         6         7
##  5.421730  5.421730  5.421730  5.421730 15.482682  5.627056
```

Lets plot the observed vs. the predict from this model (Fig. 2.13).

```
plot(mod.1$y, mod.1$fitted.values)
```

As we will see later on, the predict function works for a whole range of statistical models in R—not just lm objects. We can treat the predictions as we would any vector. For example we can add them to the above plot or put them back in the original data frame. The predict function can also give confidence and prediction intervals.

```
head(predict(mod.1, int = "conf"))
```

```
##         fit       lwr       upr
## 1  5.421730  4.437845  6.405615
## 2  5.421730  4.437845  6.405615
## 3  5.421730  4.437845  6.405615
## 4  5.421730  4.437845  6.405615
## 6 15.482682 14.152940 16.812425
## 7  5.627056  4.673657  6.580454
```

A quick way to demonstrate multiple linear regression in R, we will regress CEC on clay plus the exchangeable cations: ExchNa and ExchCa. First lets subset these data out, then get their summary statistics.

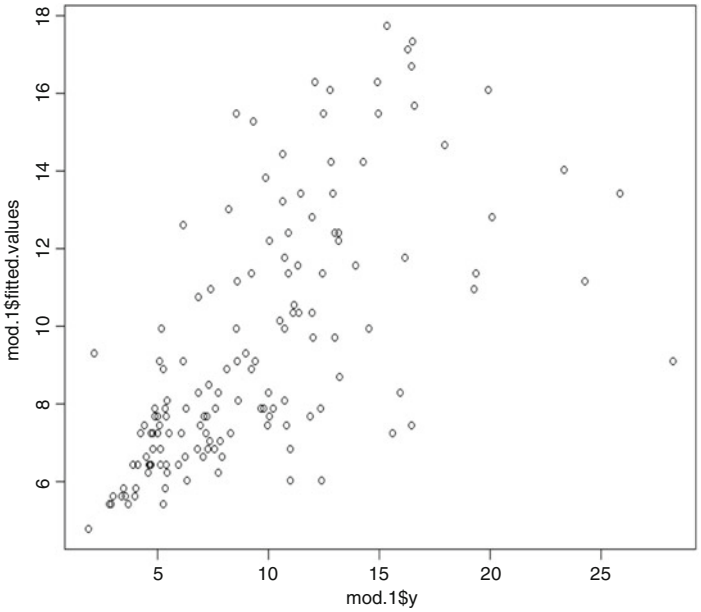

Fig. 2.13 observed vs. predicted plot of CEC from `soil.data`

```
subs.soil.data <- soil.data[, c("clay", "CEC", "ExchNa", "ExchCa")]
summary(subs.soil.data)
```

```
##       clay             CEC             ExchNa            ExchCa
##  Min.   : 5.00   Min.   : 1.900   Min.   :0.0000   Min.   : 1.010
##  1st Qu.:15.00   1st Qu.: 5.350   1st Qu.:0.0200   1st Qu.: 2.920
##  Median :21.00   Median : 8.600   Median :0.0500   Median : 4.610
##  Mean   :26.95   Mean   : 9.515   Mean   :0.2563   Mean   : 5.353
##  3rd Qu.:37.00   3rd Qu.:12.110   3rd Qu.:0.1500   3rd Qu.: 7.240
##  Max.   :68.00   Max.   :28.210   Max.   :6.8100   Max.   :25.390
##  NA's   :17      NA's   :5        NA's   :5        NA's   :5
```

A quick way to look for relationships between variables in a data frame is with the `cor` function. Note the use of the `na.omit` function.

```
cor(na.omit(subs.soil.data))
```

```
##              clay       CEC      ExchNa       ExchCa
## clay    1.0000000 0.6491351  0.17104962  0.54615778
## CEC     0.6491351 1.0000000  0.33369789  0.87570029
## ExchNa 0.1710496 0.3336979  1.00000000 -0.02055496
## ExchCa 0.5461578 0.8757003 -0.02055496  1.00000000
```

To visualize these relationships, we can use `pairs` (Fig. 2.14).

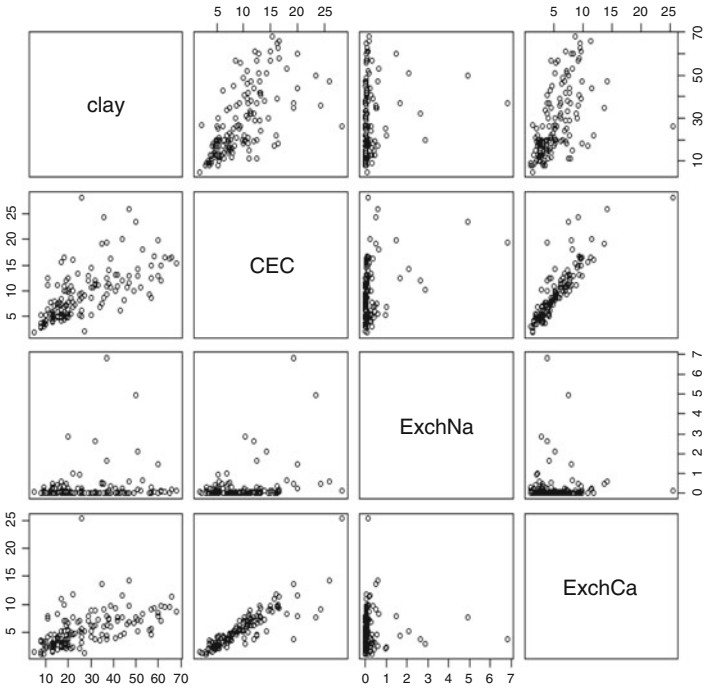

Fig. 2.14 A pairs plot of a select few soil attributes from the `soil.data` set

```
pairs(na.omit(subs.soil.data))
```

There are some interesting relationships here. Now for fitting the model:

```
mod.2 <- lm(CEC ~ clay + ExchNa + ExchCa, data = subs.soil.data)
summary(mod.2)
```

```
##
## Call:
## lm(formula = CEC ~ clay + ExchNa + ExchCa, data =
      subs.soil.data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.2008 -0.7065 -0.0470  0.6455  9.4025
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.048318   0.274264   3.822 0.000197 ***
## clay        0.050503   0.009867   5.119 9.83e-07 ***
## ExchNa      2.018149   0.163436  12.348  < 2e-16 ***
## ExchCa      1.214156   0.046940  25.866  < 2e-16 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.522 on 142 degrees of freedom
##   (20 observations deleted due to missingness)
## Multiple R-squared:  0.9076,Adjusted R-squared:  0.9057
## F-statistic: 465.1 on 3 and 142 DF,  p-value: < 2.2e-16
```

For much of the remainder of this book, we will be investigating these regression type relationships using a variety of different model types for soil spatial prediction. These fundamental modelling concepts of the `lm` will become useful as we progress ahead.

2.8.3 Exercises

1. Using the `soil.data` set firstly generate a correlation matrix of all the soil variables.
2. Choose two variables that you think would be good to regress against each other, and fit a model. Is the model any good? Can you plot the observed vs. predicted values? Can you draw a line of concordance (maybe want to consult an appropriate help file to do this). Something a bit trickier: Can you add the predictions to the data frame `soil.data` correctly.
3. Now repeat what you did for the previous question, except this time perform a multiple linear regression i.e., use more than 1 predictor variable to make a prediction of the variable you are targeting.

2.9 Advanced Work: Developing Algorithms with R

One of the advantages of using a scripting language such as R is that we can develop algorithms or script a set of commands for problem-solving. We illustrate this by developing an algorithm for generating a catena, or more correctly, a digital toposequence from a digital elevation model (DEM). To make it more interesting, we will not use any function from external libraries other than basic R functions. Before scripting, it is best to write out the algorithm or sequence of routines to use.

A toposequence can be described as a transect (not necessarily a straight line) which begins at a hilltop and ends at a valley bottom or a stream (Odgers et al. 2008). To generate a principal toposequence, one can start from the highest point in an area. If we numerically consider a rainfall as a discrete packet of "precipitation" that falls on the highest elevation pixel, then this precipitation will mostly move to its neighbor with the highest slope. As DEM is stored in matrix format, we simply

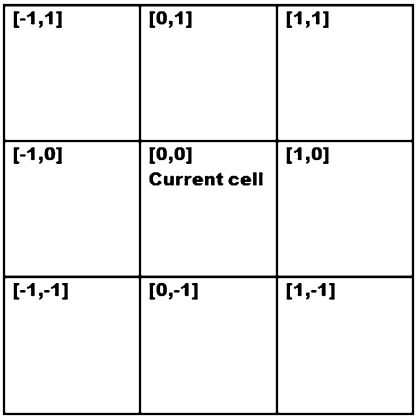

Fig. 2.15 Indexation of 3 × 3 neighborhood

look around its 3 × 3 neighbors and determine which pixel is the lowest elevation. The algorithm for principal toposequence can be written as:

1. Determine the highest point in an area.
2. Determine its 3 × 3 neighbor, and determine whether there are lower points?
3. If yes, set the lowest point as the next point in the toposequence, and then repeat step 2. If no, the toposequence has ended.

To facilitate the 3 × 3 neighbor search in R, we can code the neighbors using its relative coordinates. If the current cell is designated as [0, 0], then its left neighbor is [−1, 0], and so on. We can visualize it as follows in Fig. 2.15.

If we designate the current cell [0, 0] as *z1*, the function below will look for the lowest neighbor for pixel z1 in a DEM.

```
# function to find the lowest 3 x 3 neighbor
find_steepest <- function(dem, row_z, col_z)
{
    z1 = dem[row_z, col_z]  #elevation
    # return the elevation of the neighboring values
    dir = c(-1, 0, 1)  #neighborhood index
    nr = nrow(dem)
    nc = ncol(dem)
    pz = matrix(data = NA, nrow = 3, ncol = 3) #placeholder
         for the values
    for (i in 1:3) {
        for (j in 1:3) {
            if (i != 0 & j != 0) {
                ro <- row_z + dir[i]
                co <- col_z + dir[j]
                if (ro > 0 & co > 0 & ro < nr & co < nc) {
                  pz[i, j] = dem[ro, co]
                }
            }
```

```
        }
    }

    pz <- pz - z1  # difference of neighbors from centre value
    # find lowest value
    min_pz <- which(pz == min(pz, na.rm = TRUE), arr.ind = TRUE)
    row_min <- row_z + dir[min_pz[1]]
    col_min <- col_z + dir[min_pz[2]]
    retval <- c(row_min, col_min, min(pz, na.rm = TRUE))
    return(retval)  #return the minimum
}
```

The principal toposequence code can be implemented as follows. First we load in a small dataset called `topo_dem` from the `ithir` package.

```
library(ithir)
data(topo_dem)
str(topo_dem)
```

```
##  num [1:109, 1:110] 121 121 120 118 116 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:110] "V1" "V2" "V3" "V4" ...
```

Now we want to create a data matrix to store the result of the toposequence i.e. the row, column, and elevation values that are selected using the `find_steepest` function.

```
transect <- matrix(data = NA, nrow = 20, ncol = 3)
```

Now we want to find within that matrix that maximum elevation value and its corresponding row and column position.

```
max_elev <- which(topo_dem == max(topo_dem), arr.ind = TRUE)
row_z = max_elev[1]  # row of max_elev
col_z = max_elev[2]  # col of max_elev
z1 = topo_dem[row_z, col_z]  # max elevation

# Put values into the first entry of the transect object
t <- 1
transect[t, 1] = row_z
transect[t, 2] = col_z
transect[t, 3] = z1
lowest = FALSE
```

Below we use the `find_steepest` function. It is embedded into a `while` conditional loop, such that the routine will run until neither of the surrounding neighbors are less than the middle pixel or *z1*. We use the `break` function to stop the routine when this occurs. Essentially, upon each iteration, we use the selected *z1*

to find the lowest value pixel from it, which in turn becomes the selected *z1* and so on until the values of the neighborhood are no longer smaller than the selected *z1*.

```
# iterate down the hill until lowest point
while (lowest == FALSE) {
    result <- find_steepest(dem = topo_dem, row_z, col_z) # find
       steepest neighbor
    t <- t + 1
    row_z = result[1]
    col_z = result[2]
    z1 = topo_dem[row_z, col_z]
    transect[t, 1] = row_z
    transect[t, 2] = col_z
    transect[t, 3] = z1
    if (result[3] >= 0)
        {
            lowest == TRUE
            break
        }  # if found lowest point
}
```

Finally we can plot the transect. First lets calculate a distance relative to the top of the transect. After this we can generate a plot as in Fig. 2.16.

```
dist = sqrt((transect[1, 1] - transect[, 1])^2 + (transect[1, 2]
   - transect[, 2])^2)
plot(dist, transect[, 3], type = "l", xlab = "Distance (m)",
   ylab = "Elevation (m)")
```

So let's take this a step further and consider the idea of a random toposequence. In reality, water does not only flow in the steepest direction, water can potentially move down to any lower elevation. And, a toposequence does not necessarily start at the highest elevation either. We can generate a random toposequence (Odgers et al. 2008), where we select a random point in the landscape, then find a random path to

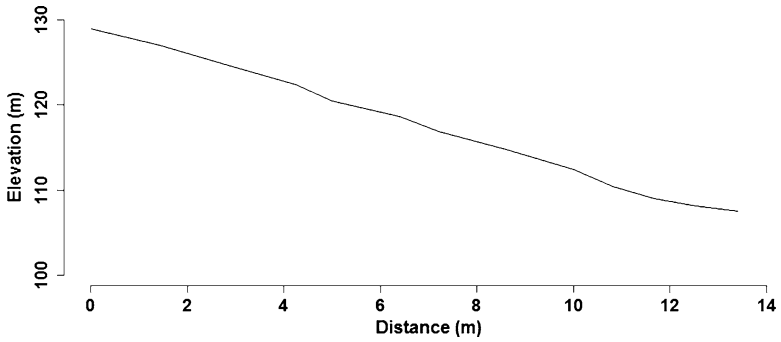

Fig. 2.16 Generated toposequence

the top and bottom of a hillslope. In addition to the downhill routine, we need an uphill routine too.

The algorithm for the random toposequence could be written as:

1. Select a random point from a DEM.
2. Travel uphill:
   - 2.1 Determine its 3×3 neighbor, and determine whether there are higher points?
   - 2.2 If yes, select randomly a higher point, add to the uphill sequence, and repeat step 2.1. If this point is the highest, the uphill sequence ended.
3. Travel downhill:
   - 3.1 Determine its 3×3 neighbor, and determine whether there are lower points?
   - 3.2 If yes, select randomly a lower point, add to the downhill sequence, and repeat step 3.1. If this point is the lowest or reached a stream, the downhill sequence ended.

From this algorithm plan, we need to specify two functions, one that allows the transect to travel uphill and another which allows it to travel downhill. For the one to travel downhill, we could use the function from before (`find_steepest`), but we want to build on that function by allowing the user to indicate whether they want a randomly selected smaller value, or whether they want to minimum every time. Subsequently the two new functions would take the following form:

```
# function to simulate water moving down the slope
# input: dem and its row &
travel_down <- function(dem, row_z, col_z, random)
# column random: TRUE use random path, FALSE for steepest path
  return:
# row,col,z-z1 of lower neighbour
{
    z1 = dem[row_z, col_z]
    # find its eight neighbour
    dir = c(-1, 0, 1)
    nr = nrow(dem)
    nc = ncol(dem)
    pz = matrix(data = NA, nrow = 3, ncol = 3)
    for (i in 1:3) {
        for (j in 1:3) {
            ro <- row_z + dir[i]
            co <- col_z + dir[j]
            if (ro > 0 & co > 0 & ro < nr & co < nc) {
                pz[i, j] = dem[ro, co]
            }
        }
    }
    pz[2, 2] = NA
    pz <- pz - z1  # difference with centre value
```

```
    min_pz <- which(pz < 0, arr.ind = TRUE)
    nlow <- nrow(min_pz)
    if (nlow == 0) {
        min_pz <- which(pz == min(pz, na.rm = TRUE),
        arr.ind = TRUE)
    } else {
        if (random) {
            # find random lower value
            ir <- sample.int(nlow, size = 1)
            min_pz <- min_pz[ir, ]
        } else {
            # find lowest value
            min_pz <- which(pz == min(pz, na.rm = TRUE),
            arr.ind = TRUE)
        }
    }
    row_min <- row_z + dir[min_pz[1]]
    col_min <- col_z + dir[min_pz[2]]
    z_min <- dem[row_min, col_min]
    retval <- c(row_min, col_min, min(pz, na.rm = TRUE))
    return(retval)
}

# function to trace water coming from up hill
# input: dem and its row &
travel_up <- function(dem, row_z, col_z, random)
# column random: TRUE use random path, FALSE for steepest path
  return:
# row,col,z-zi of higher neighbour
{
    z1 = dem[row_z, col_z]
    # find its eight neighbour
    dir = c(-1, 0, 1)
    nr = nrow(dem)
    nc = ncol(dem)
    pz = matrix(data = NA, nrow = 3, ncol = 3)
    for (i in 1:3) {
        for (j in 1:3) {
            ro <- row_z + dir[i]
            co <- col_z + dir[j]
            if (ro > 0 & co > 0 & ro < nr & co < nc) {
                pz[i, j] = dem[ro, co]
            }
        }
    }
    pz[2, 2] = NA
    pz <- pz - z1  # difference with centre value

    max_pz <- which(pz > 0, arr.ind = TRUE)  # find higher pixel
    nhi <- nrow(max_pz)
    if (nhi == 0) {
```

```
        max_pz <- which(pz == max(pz, na.rm = TRUE),
        arr.ind = TRUE)
    } else {
        if (random) {
            # find random higher value
            ir <- sample.int(nhi, size = 1)
            max_pz <- max_pz[ir, ]
        } else {
            # find highest value
            max_pz <- which(pz == max(pz, na.rm = TRUE),
            arr.ind = TRUE)
        }
    }
    row_max <- row_z + dir[max_pz[1]]
    col_max <- col_z + dir[max_pz[2]]
    retval <- c(row_max, col_max, max(pz, na.rm = TRUE))
    return(retval)
}
```

Now we can generate a random toposequence. We will use the same `topo_dem` data as before. First we select a point at random using a random selection of a row and column value. Keep in mind that the random point selected here may be different to the one you get because we are using a random number generator via the `sample.int` function.

```
nr <- nrow(topo_dem)  # no. rows in a DEM
nc <- ncol(topo_dem)  # no. cols in a DEM

# start with a random pixel as seed point
row_z1 <- sample.int(nr, 1)
col_z1 <- sample.int(nc, 1)
```

We then can use the `travel_up` function to get our transect to go up the slope.

```
# Travel uphill seed point as a starting point
t <- 1
transect_up <- matrix(data = NA, nrow = 100, ncol = 3)
row_z <- row_z1
col_z <- col_z1
z1 = topo_dem[row_z, col_z]
transect_up[t, 1] = row_z
transect_up[t, 2] = col_z
transect_up[t, 3] = z1

highest = FALSE
# iterate up the hill until highest point
while (highest == FALSE) {
    result <- travel_up(dem = topo_dem, row_z, col_z,
    random = TRUE)
    if (result[3] <= 0)
```

```
        {
            highest == TRUE
            break
        }  # if found lowest point
    t <- t + 1
    row_z = result[1]
    col_z = result[2]
    z1 = topo_dem[row_z, col_z]
    transect_up[t, 1] = row_z
    transect_up[t, 2] = col_z
    transect_up[t, 3] = z1
}
transect_up <- na.omit(transect_up)
```

Next we then use the `travel_down` function to get our transect to go down the slope from the seed point.

```
# travel downhill create a data matrix to store results
transect_down <- matrix(data = NA, nrow = 100, ncol = 3)
# starting point
row_z <- row_z1
col_z <- col_z1
z1 = topo_dem[row_z, col_z]  # a random pixel
t <- 1
transect_down[t, 1] = row_z
transect_down[t, 2] = col_z
transect_down[t, 3] = z1
lowest = FALSE

# iterate down the hill until lowest point
while (lowest == FALSE) {
    result <- travel_down(dem = topo_dem, row_z, col_z,
    random = TRUE)
    if (result[3] >= 0)
        {
            lowest == TRUE
            break
        }  # if found lowest point
    t <- t + 1
    row_z = result[1]
    col_z = result[2]
    z1 = topo_dem[row_z, col_z]
    transect_down[t, 1] = row_z
    transect_down[t, 2] = col_z
    transect_down[t, 3] = z1
}
transect_down <- na.omit(transect_down)
```

The idea then is to bind both uphill and downhill transects into a single one. Note we are using the `rbind` function for this. Furthermore, we are also using the `order` function here to re-arrange the uphill transect so that the resultant binding

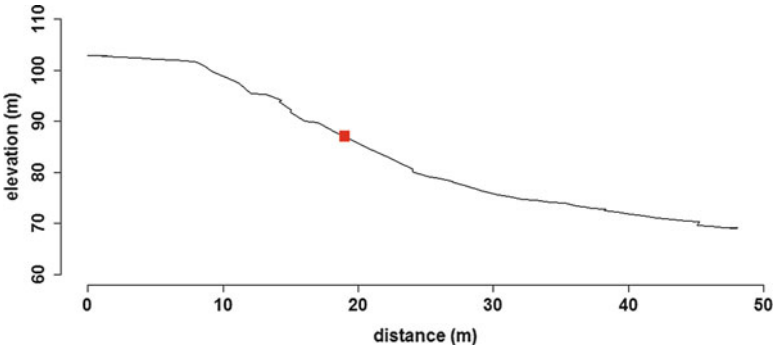

Fig. 2.17 Generated random toposequence. (*Red point* indicates the random seed point)

will be sequential from highest to lowest elevation. Finally, we then calculate the distance relative to the hilltop.

```
transect <- rbind(transect_up[order(transect_up[, 3],
decreasing = T), ],
transect_down[-1, ])
# calculate distance from hilltop
dist = sqrt((transect[1, 1] - transect[, 1])^2 +
(transect[1, 2] - transect[, 2])^2)
```

The last step is to make the plot (Fig. 2.17) of the transect. We can also add the randomly selected seed point for visualization purposes.

```
plot(dist, transect[, 3], type = "l", col = "red",
  xlim = c(0, 100), ylim = c(50, 120), xlab = "Distance (m)",
  ylab = "Elevation (m)")
points(dist[nrow(transect_up)], transect[nrow(transect_up), 3])
```

After seeing how this algorithm works, you can modify the script to take in stream networks, and make the toposequence end once it reaches the stream. You can also add "error trapping" to handle missing values, and also in case where the downhill routine ends up in a local depression. This algorithm also can be used to calculate slope length, distance to a particular landscape feature (e.g. hedges), and so on.

Reference

Odgers NP, McBratney AB, Minasny B (2008) Generation of kth-order random toposequences. Comput Geosci 34(5):479–490

http://www.springer.com/978-3-319-44325-6

Using R for Digital Soil Mapping

Malone, B.P.; Minasny, B.; McBratney, A.B.

2017, XVI, 262 p. 61 illus., 44 illus. in color., Hardcover
ISBN: 978-3-319-44325-6