Chapter 2

An Introduction to R

2.1 Downloading and Installing R

The instructions for obtaining R largely depend on the user's hardware and operating system. The R Project has written an R Installation and Administration manual with complete, precise instructions about what to do, together with all sorts of additional information. The following is just a primer to get a person started.

2.1.1 Installing R

Visit one of the links below to download the latest version of R for your operating system:

Microsoft Windows: `http://cran.r-project.org/bin/windows/base/`

MacOS: `http://cran.r-project.org/bin/macosx/`

Linux: `http://cran.r-project.org/bin/linux/`

On Microsoft Windows, click the `R-x.y.z.exe` installer to start installation. When it asks for "Customized startup options", specify Yes. In the next window, be sure to select the SDI (single document interface) option; this is useful later when we discuss three dimensional plots with the `rgl` package [1].

Installing R on a USB drive (Windows) With this option you can use R portably and without administrative privileges. There is an entry in the R for Windows FAQ about this. Here is the procedure I use:

1. Download the Windows installer above and start installation as usual. When it asks *where* to install, navigate to the top-level directory of the USB drive instead of the default `C` drive.

2. When it asks whether to modify the Windows registry, uncheck the box; we do NOT want to tamper with the registry.

3. After installation, change the name of the folder from `R-x.y.z` to just plain `R`. (Even quicker: do this in step 1.)

4. Download the following shortcut to the top-level directory of the USB drive, right beside the R folder, not inside the folder.

http://ipsur.r-forge.r-project.org/book/download/R.exe

Use the downloaded shortcut to run R.

Steps 3 and 4 are not required but save you the trouble of navigating to the R-x.y.z/bin directory to double-click Rgui.exe every time you want to run the program. It is useless to create your own shortcut to Rgui.exe. Windows does not allow shortcuts to have relative paths; they always have a drive letter associated with them. So if you make your own shortcut and plug your USB drive into some *other* machine that happens to assign your drive a different letter, then your shortcut will no longer be pointing to the right place.

2.1.2 Installing and Loading Add-on Packages

There are *base* packages (which come with R automatically), and *contributed* packages (which must be downloaded for installation). For example, on the version of R being used for this document the default base packages loaded at startup are

```
> getOption("defaultPackages")
[1] "datasets"  "utils"     "grDevices" "graphics"  "stats"     "methods"
```

The base packages are maintained by a select group of volunteers, called "R Core". In addition to the base packages, there are literally thousands of additional contributed packages written by individuals all over the world. These are stored worldwide on mirrors of the Comprehensive R Archive Network, or CRAN for short. Given an active Internet connection, anybody is free to download and install these packages and even inspect the source code.

To install a package named foo, open up R and type install.packages("foo"). To install foo and additionally install all of the other packages on which foo depends, instead type install.packages("foo", depends = TRUE).

The general command install.packages() will (on most operating systems) open a window containing a huge list of available packages; simply choose one or more to install.

No matter how many packages are installed onto the system, each one must first be loaded for use with the library function. For instance, the foreign package [18] contains all sorts of functions needed to import data sets into R from other software such as SPSS, SAS, *etc.*. But none of those functions will be available until the command library(foreign) is issued.

Type library() at the command prompt (described below) to see a list of all available packages in your library.

For complete, precise information regarding installation of R and add-on packages, see the R Installation and Administration manual, http://cran.r-project.org/manuals.html.

2.2 Communicating with R

One line at a time This is the most basic method and is the first one that beginners will use.

RGui (Microsoft® Windows)

Terminal

Emacs/ESS, XEmacs

JGR

Multiple lines at a time For longer programs (called *scripts*) there is too much code to write all at once at the command prompt. Furthermore, for longer scripts it is convenient to be able to only modify a certain piece of the script and run it again in R. Programs called *script editors* are specially designed to aid the communication and code writing process. They have all sorts of helpful features including R syntax highlighting, automatic code completion, delimiter matching, and dynamic help on the R functions as they are being written. Even more, they often have all of the text editing features of programs like Microsoft® Word. Lastly, most script editors are fully customizable in the sense that the user can customize the appearance of the interface to choose what colors to display, when to display them, and how to display them.

R Editor (Windows): In Microsoft® Windows, RGui has its own built-in script editor, called R Editor. From the console window, select File ▸ New Script. A script window opens, and the lines of code can be written in the window. When satisfied with the code, the user highlights all of the commands and presses Ctrl+R. The commands are automatically run at once in R and the output is shown. To save the script for later, click File ▸ Save as... in R Editor. The script can be reopened later with File ▸ Open Script... in RGui. Note that R Editor does not have the fancy syntax highlighting that the others do.

RWinEdt: This option is coordinated with WinEdt for LaTeX and has additional features such as code highlighting, remote sourcing, and a ton of other things. However, one first needs to download and install a shareware version of another program, WinEdt, which is only free for a while – pop-up windows will eventually appear that ask for a registration code. RWinEdt is nevertheless a very fine choice if you already own WinEdt or are planning to purchase it in the near future.

Tinn-R/Sciviews-K: This one is completely free and has all of the above mentioned options and more. It is simple enough to use that the user can virtually begin working with it immediately after installation. But Tinn-R proper is only available for Microsoft® Windows operating systems. If you are on MacOS or Linux, a comparable alternative is Sci-Views - Komodo Edit.

Emacs/ESS: Emacs is an all purpose text editor. It can do absolutely anything with respect to modifying, searching, editing, and manipulating, text. And if Emacs can't do it, then you can write a program that extends Emacs to do it. Once such extension is called *ESS*, which stands for *E*macs *S*peaks *S*tatistics. With ESS a person can speak to R, do all of the tricks that the other script editors offer, and much, much, more. Please see the following for installation details, documentation, reference cards, and a whole lot more:

`http://ess.r-project.org`

Fair warning: if you want to try Emacs and if you grew up with Microsoft® Windows or Macintosh, then you are going to need to relearn everything you thought you knew about computers your whole life. (Or, since Emacs is completely customizable, you can reconfigure Emacs to behave the way you want.) I have personally experienced this transformation and I will never go back.

JGR (read "Jaguar"): This one has the bells and whistles of RGui plus it is based on Java, so it works on multiple operating systems. It has its own script editor like R Editor but with additional features such as syntax highlighting and code-completion. If you do not use Microsoft® Windows (or even if you do) you definitely want to check out this one.

Kate, Bluefish, *etc*. There are literally dozens of other text editors available, many of them free, and each has its own (dis)advantages. I only have mentioned the ones with which I have had substantial personal experience and have enjoyed at some point. Play around, and let me know what you find.

Graphical User Interfaces (GUIs) By the word "GUI" I mean an interface in which the user communicates with R by way of points-and-clicks in a menu of some sort. Again, there are many, many options and I only mention ones that I have used and enjoyed. Some of the other more popular script editors can be downloaded from the R-Project website at `http://www.sciviews.org/_r`
On the left side of the screen (under **Projects**) there are several choices available.

R Commander provides a point-and-click interface to many basic statistical tasks. It is called the "Commander" because every time one makes a selection from the menus, the code corresponding to the task is listed in the output window. One can take this code, copy-and-paste it to a text file, then re-run it again at a later time without the R Commander's assistance. It is well suited for the introductory level. `Rcmdr` also allows for user-contributed "Plugins" which are separate packages on `CRAN` that add extra functionality to the `Rcmdr` package. The plugins are typically named with the prefix `RcmdrPlugin` to make them easy to identify in the `CRAN` package list. One such plugin is the `RcmdrPlugin.IPSUR` package which accompanies this text.

Poor Man's GUI is an alternative to the `Rcmdr` which is based on GTk instead of Tcl/Tk. It has been a while since I used it but I remember liking it very much when I did. One thing that stood out was that the user could drag-and-drop data sets for plots. See here for more information: `http://wiener.math.csi.cuny.edu/pmg/`.

Rattle is a data mining toolkit which was designed to manage/analyze very large data sets, but it provides enough other general functionality to merit mention here. See [91] for more information.

Deducer is relatively new and shows promise from what I have seen, but I have not actually used it in the classroom yet.

2.3 Basic R Operations and Concepts

The R developers have written an introductory document entitled "An Introduction to R". There is a sample session included which shows what basic interaction with R looks like. I recommend that all new users of R read that document, but bear in mind that there are concepts mentioned which will be unfamiliar to the beginner.

Below are some of the most basic operations that can be done with R. Almost every book about R begins with a section like the one below; look around to see all sorts of things that can be done at this most basic level.

2.3.1 Arithmetic

```
> 2 + 3        # add
[1] 5
```

```
> 4 * 5 / 6   # multiply and divide
[1] 3.333333
> 7^8         # 7 to the 8th power
[1] 5764801
```

Notice the comment character #. Anything typed after a # symbol is ignored by R. We know that 20/6 is a repeating decimal, but the above example shows only 7 digits. We can change the number of digits displayed with `options`:

```
> options(digits = 16)
> 10/3                # see more digits
[1] 3.333333333333333
> sqrt(2)             # square root
[1] 1.414213562373095
> exp(1)              # Euler's constant, e
[1] 2.718281828459045
> pi
[1] 3.141592653589793
> options(digits = 7)  # back to default
```

Note that it is possible to set `digits` up to 22, but setting them over 16 is not recommended (the extra significant digits are not necessarily reliable). Above notice the `sqrt` function for square roots and the `exp` function for powers of e, Euler's number.

2.3.2 Assignment, Object names, and Data types

It is often convenient to assign numbers and values to variables (objects) to be used later. The proper way to assign values to a variable is with the <- operator (with a space on either side). The = symbol works too, but it is recommended by the R masters to reserve = for specifying arguments to functions (discussed later). In this book we will follow their advice and use <- for assignment. Once a variable is assigned, its value can be printed by simply entering the variable name by itself.

```
> x <- 7*41/pi   # don't see the calculated value
> x              # take a look
[1] 91.35494
```

When choosing a variable name you can use letters, numbers, dots ".", or underscore "_" characters. You cannot use mathematical operators, and a leading dot may not be followed by a number. Examples of valid names are: `x`, `x1`, `y.value`, and `y_hat`. (More precisely, the set of allowable characters in object names depends on one's particular system and locale; see An Introduction to R for more discussion on this.)

Objects can be of many *types*, *modes*, and *classes*. At this level, it is not necessary to investigate all of the intricacies of the respective types, but there are some with which you need to become familiar:

integer: the values 0, ±1, ±2, . . . ; these are represented exactly by R.

double: real numbers (rational and irrational); these numbers are not represented exactly (save integers or fractions with a denominator that is a multiple of 2, see [85]).

character: elements that are wrapped with pairs of " or ';

logical: includes TRUE, FALSE, and NA (which are reserved words); the NA stands for "not available", *i.e.*, a missing value.

You can determine an object's type with the typeof function. In addition to the above, there is the complex data type:

```
> sqrt(-1)              # isn't defined
[1] NaN
> sqrt(-1+0i)           # is defined
[1] 0+1i
> sqrt(as.complex(-1))  # same thing
[1] 0+1i
> (0 + 1i)^2            # should be -1
[1] -1+0i
> typeof((0 + 1i)^2)
[1] "complex"
```

Note that you can just type (1i)^2 to get the same answer. The NaN stands for "not a number"; it is represented internally as double.

2.3.3 Vectors

All of this time we have been manipulating vectors of length 1. Now let us move to vectors with multiple entries.

Entering data vectors

1. c: If you would like to enter the data 74,31,95,61,76,34,23,54,96 into R, you may create a data vector with the c function (which is short for *concatenate*).

   ```
   > x <- c(74, 31, 95, 61, 76, 34, 23, 54, 96)
   > x
   [1] 74 31 95 61 76 34 23 54 96
   ```

   The elements of a vector are usually coerced by R to the the most general type of any of the elements, so if you do c(1, "2") then the result will be c("1", "2").

2. scan: This method is useful when the data are stored somewhere else. For instance, you may type x <- scan() at the command prompt and R will display 1: to indicate that it is waiting for the first data value. Type a value and press Enter, at which point R will display 2:, and so forth. Note that entering an empty line stops the scan. This method is especially handy when you have a column of values, say, stored in a text file or spreadsheet. You may copy and paste them all at the 1: prompt, and R will store all of the values instantly in the vector x.

3. repeated data; regular patterns: the seq function will generate all sorts of sequences of numbers. It has the arguments from, to, by, and length.out which can be set in concert with one another. We will do a couple of examples to show you how it works.

```
> seq(from = 1, to = 5)
[1] 1 2 3 4 5
> seq(from = 2, by = -0.1, length.out = 4)
[1] 2.0 1.9 1.8 1.7
```

Note that we can get the first line much quicker with the colon operator :

```
> 1:5
[1] 1 2 3 4 5
```

The vector LETTERS has the 26 letters of the English alphabet in uppercase and letters has all of them in lowercase.

Indexing data vectors Sometimes we do not want the whole vector, but just a piece of it. We can access the intermediate parts with the [] operator. Observe (with x defined above)

```
> x[1]
[1] 74
> x[2:4]
[1] 31 95 61
> x[c(1, 3, 4, 8)]
[1] 74 95 61 54
> x[-c(1, 3, 4, 8)]
[1] 31 76 34 23 96
```

Notice that we used the minus sign to specify those elements that we do *not* want.

```
> LETTERS[1:5]
[1] "A" "B" "C" "D" "E"
> letters[-(6:24)]
[1] "a" "b" "c" "d" "e" "y" "z"
```

2.3.4 Functions and Expressions

A function takes arguments as input and returns an object as output. There are functions to do all sorts of things. We show some examples below.

```
> x <- 1:5
> sum(x)
[1] 15
> length(x)
[1] 5
> min(x)
[1] 1
> mean(x)      # sample mean
[1] 3
> sd(x)        # sample standard deviation
[1] 1.581139
```

It will not be long before the user starts to wonder how a particular function is doing its job, and since R is open-source, anybody is free to look under the hood of a function to see how things are calculated. For detailed instructions see the article "Accessing the Sources" by Uwe Ligges [60]. In short:

1. Type the name of the function without any parentheses or arguments. If you are lucky then the code for the entire function will be printed, right there looking at you. For instance, suppose that we would like to see how the `intersect` function works:

   ```
   > intersect
   function (x, y)
   {
       y <- as.vector(y)
       unique(y[match(as.vector(x), y, 0L)])
   }
   <environment: namespace:base>
   ```

2. If instead it shows `UseMethod("`*something*`")` then you will need to choose the *class* of the object to be inputted and next look at the *method* that will be *dispatched* to the object. For instance, typing `rev` says

   ```
   > rev
   function (x)
   UseMethod("rev")
   <environment: namespace:base>
   ```

The output is telling us that there are multiple methods associated with the `rev` function. To see what these are, type

```
> methods(rev)

[1] rev.default     rev.dendrogram*

   Non-visible functions are asterisked
```

Now we learn that there are two different `rev(x)` functions, only one of which being chosen at each call depending on what `x` is. There is one for `dendrogram` objects and a `default` method for everything else. Simply type the name to see what each method does. For example, the `default` method can be viewed with

```
> rev.default

function (x)
if (length(x)) x[length(x):1L] else x
<environment: namespace:base>
```

3. Some functions are hidden by a *namespace* (see An Introduction to R [85]), and are not visible on the first try. For example, if we try to look at the code for `wilcox.test` (see Chapter 15) we get the following:

```
> wilcox.test

function (x, ...)
UseMethod("wilcox.test")
<environment: namespace:stats>

> methods(wilcox.test)

[1] wilcox.test.default* wilcox.test.formula*

   Non-visible functions are asterisked
```

If we were to try `wilcox.test.default` we would get a "not found" error, because it is hidden behind the namespace for the package `stats` (shown in the last line when we tried `wilcox.test`). In cases like these we prefix the package name to the front of the function name with three colons; the command `stats:::wilcox.test.default` will show the source code, omitted here for brevity.

4. If it shows `.Internal(`*something*`)` or `.Primitive("`*something*`")`, then it will be necessary to download the source code of R (which is *not* a binary version with an `.exe` extension) and search inside the code there. See Ligges [60] for more discussion on this. An example is `exp`:

```
> exp

function (x)  .Primitive("exp")
```

Be warned that most of the `.Internal` functions are written in other computer languages which the beginner may not understand, at least initially.

2.4 Getting Help

When you are using R, it will not take long before you find yourself needing help. Fortunately, R has extensive help resources and you should immediately become familiar with them. Begin by clicking Help on Rgui. The following options are available.

- **Console**: gives useful shortcuts, for instance, Ctrl+L, to clear the R console screen.
- **FAQ on R**: frequently asked questions concerning general R operation.
- **FAQ on R for Windows**: frequently asked questions about R, tailored to the Microsoft Windows operating system.
- **Manuals**: technical manuals about all features of the R system including installation, the complete language definition, and add-on packages.
- **R functions (text)...**: use this if you know the *exact* name of the function you want to know more about, for example, `mean` or `plot`. Typing `mean` in the window is equivalent to typing `help("mean")` at the command line, or more simply, `?mean`. Note that this method only works if the function of interest is contained in a package that is already loaded into the search path with `library`.
- **HTML Help**: use this to browse the manuals with point-and-click links. It also has a Search Engine & Keywords for searching the help page titles, with point-and-click links for the search results. This is possibly the best help method for beginners. It can be started from the command line with the command `help.start()`.
- **Search help...**: use this if you do not know the exact name of the function of interest, or if the function is in a package that has not been loaded yet. For example, you may enter `plo` and a text window will return listing all the help files with an alias, concept, or title matching '`plo`' using regular expression matching; it is equivalent to typing `help.search("plo")` at the command line. The advantage is that you do not need to know the exact name of the function; the disadvantage is that you cannot point-and-click the results. Therefore, one may wish to use the HTML Help search engine instead. An equivalent way is `??plo` at the command line.
- **search.r-project.org...**: this will search for words in help lists and email archives of the R Project. It can be very useful for finding other questions that other users have asked.
- **Apropos...**: use this for more sophisticated partial name matching of functions. See `?apropos` for details.

On the help pages for a function there are sometimes "Examples" listed at the bottom of the page, which will work if copy-pasted at the command line (unless marked otherwise). The `example` function will run the code automatically, skipping the intermediate step. For instance, we may try `example(mean)` to see a few examples of how the `mean` function works.

2.4.1 R Help Mailing Lists

There are several mailing lists associated with R, and there is a huge community of people that read and answer questions related to R. See here `http://www.r-project.org/mail.html`

for an idea of what is available. Particularly pay attention to the bottom of the page which lists several special interest groups (SIGs) related to R.

Bear in mind that R is free software, which means that it was written by volunteers, and the people that frequent the mailing lists are also volunteers who are not paid by customer support fees. Consequently, if you want to use the mailing lists for free advice then you must adhere to some basic etiquette, or else you may not get a reply, or even worse, you may receive a reply which is a bit less cordial than you are used to. Below are a few considerations:

1. Read the FAQ (`http://cran.r-project.org/faqs.html`). Note that there are different FAQs for different operating systems. You should read these now, even without a question at the moment, to learn a lot about the idiosyncrasies of R.

2. Search the archives. Even if your question is not a FAQ, there is a very high likelihood that your question has been asked before on the mailing list. If you want to know about topic `foo`, then you can do `RSiteSearch("foo")` to search the mailing list archives (and the online help) for it.

3. Do a Google search and an `RSeek.org` search.

If your question is not a FAQ, has not been asked on R-help before, and does not yield to a Google (or alternative) search, then, and only then, should you even consider writing to R-help. Below are a few additional considerations.

1. **Read the posting guide (`http://www.r-project.org/posting-guide.html`) before posting.** This will save you a lot of trouble and pain.

2. Get rid of the command prompts (>) from output. Readers of your message will take the text from your mail and copy-paste into an R session. If you make the readers' job easier then it will increase the likelihood of a response.

3. Questions are often related to a specific data set, and the best way to communicate the data is with a `dump` command. For instance, if your question involves data stored in a vector `x`, you can type `dump("x","")` at the command prompt and copy-paste the output into the body of your email message. Then the reader may easily copy-paste the message from your email into R and `x` will be available to him/her.

4. Sometimes the answer the question is related to the operating system used, the attached packages, or the exact version of R being used. The `sessionInfo()` command collects all of this information to be copy-pasted into an email (and the Posting Guide requests this information). See Appendix A for an example.

2.5 External Resources

There is a mountain of information on the Internet about R. Below are a few of the important ones.

The R Project for Statistical Computing: (`http://www.r-project.org/`) Go here first.

The Comprehensive R Archive Network: (`http://cran.r-project.org/`) This is where R is stored along with thousands of contributed packages. There are also loads of contributed information (books, tutorials, *etc*.). There are mirrors all over the world with duplicate information.

R-Forge: (http://r-forge.r-project.org/) This is another location where R packages are stored. Here you can find development code which has not yet been released to CRAN.

R Wiki: (http://wiki.r-project.org/rwiki/doku.php) There are many tips and tricks listed here. If you find a trick of your own, login and share it with the world.

Other: the R Graph Gallery (http://addictedtor.free.fr/graphiques/) and R Graphical Manual (http://bm2.genes.nig.ac.jp/RGM2/index.php) have literally thousands of graphs to peruse. RSeek (http://www.rseek.org) is a search engine based on Google specifically tailored for R queries.

2.6 Other Tips

It is unnecessary to retype commands repeatedly, since R remembers what you have recently entered on the command line. On the Microsoft® Windows RGui, to cycle through the previous commands just push the ↑ (up arrow) key. On Emacs/ESS the command is M-p (which means hold down the Alt button and press "p"). More generally, the command history() will show a whole list of recently entered commands.

- To find out what all variables are in the current work environment, use the commands objects() or ls(). These list all available objects in the workspace. If you wish to remove one or more variables, use remove(var1, var2, var3), or more simply use rm(var1, var2, var3), and to remove all objects use rm(list = ls()).
- Another use of scan is when you have a long list of numbers (separated by spaces or on different lines) already typed somewhere else, say in a text file. To enter all the data in one fell swoop, first highlight and copy the list of numbers to the Clipboard with Edit ▸ Copy (or by right-clicking and selecting Copy). Next type the x <- scan() command in the R console, and paste the numbers at the 1: prompt with Edit ▸ Paste. All of the numbers will automatically be entered into the vector x.
- The command Ctrl+l clears the screen in the Microsoft® Windows RGui. The comparable command for Emacs/ESS is
- Once you use R for awhile there may be some commands that you wish to run automatically whenever R starts. These commands may be saved in a file called Rprofile.site which is usually in the etc folder, which lives in the R home directory (which on Microsoft® Windows usually is C:\Program Files\R). Alternatively, you can make a file .Rprofile to be stored in the user's home directory, or anywhere R is invoked. This allows for multiple configurations for different projects or users. See "Customizing the Environment" of *An Introduction to R* for more details.
- When exiting R the user is given the option to "save the workspace". I recommend that beginners DO NOT save the workspace when quitting. If Yes is selected, then all of the objects and data currently in R's memory is saved in a file located in the working directory called .RData. This file is then automatically loaded the next time R starts (in which case R will say [previously saved workspace restored]). This is a valuable feature for experienced users of R, but I find that it causes more trouble than it saves with beginners.