BASICS OF R
A Primer

Don Edwards

Department of Statistics
University of South Carolina
Columbia, SC 29208
edwards@stat.sc.edu

September 2004
Edited July 2007, August 2009, September 2011

TABLE OF CONTENTS

1. INTRODUCTION

R is a shareware implementation of the S language, which was developed at Bell Labs in the 1970s and '80s. As such it has many similarities with Splus®, another implementation of S now for sale as a commercial software package for data analysis, distributed by TIBCO Software Inc. Learning R is essentially equivalent to learning Splus. Most commands/programs written in R (Splus) run with little or no modification in Splus (R). R is free, though; to download it, go to

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

Note also that additional R documentation is available through this link; in particular, under the Manuals link, there is a good quick-start "An Introduction to R" manual.

This primer is aimed at an individual with minimal R (or Splus) experience, to introduce the structure and syntax of the language, to provide some basic tools for manipulating data, and to introduce many other topics which deserve further study. It assumes you are using the Windows implementation of R, version 2.7 or above. Implementations for Unix, Linux, and Macintosh are also available, and supposedly do not differ dramatically in syntax or operation from the Windows implementation. Any suggestions for improvements to this primer can be sent to Don Edwards (edwards@stat.sc.edu), David Hitchcock (hitchcock@stat.sc.edu) or John Grego (grego@stat.sc.edu).

R is a powerful language, but it has a fairly steep learning curve. It is case sensitive, therefore unforgiving of careless mistakes, and its error messages leave a bit to be desired. On the other hand, it is very easy to study R objects as they are created; this is a great help in understanding coding mistakes. Patience and a lot of regular use are keys to learning R. As the cover cartoon suggests, it is like a musical instrument - if you can get past the learning phase, working with R can be enjoyable.

In this primer, R objects, operators, etc. are written in Courier New font (`like this`), the default font used in the R command window. Names of syntactically rigid commands, keywords, and built-in function names in R are written in boldface to distinguish them from created objects, whose names would often be user-provided. Commands that are meant to be demonstrated are highlighted in red. We also often use silly generic names like `my.object` when referring to user-named objects. Names can be essentially any length, but should start with a character and contain no blanks. If a name consists of multiple words, usually we connect them with periods or underscores. Try not to use names that might be natural choices for an existing built-in function, since objects could be hidden ("masked") by other objects with the same name in a different R directory. Some names to avoid, specifically, are **`"plot", "t", "c", "df", "T",`** and **`"F"`**, though these should only cause warning messages or confusion unless you use them as names for functions you create. Choose names meaningfully, since created objects tend to collect in the workspace at an alarming rate, and if they are not named well, and/or if you do not clean up your workspace regularly, you'll forget what they are.

To start R, of course you double-click on the R icon on the desktop, or under Programs in the Start menu. Start R now; an "Rgui" (graphical user interface) window labeled "R console" should open, and at the bottom you should see the R command prompt "**>**". When you type a command, press Enter or Return to execute it. You can page through recent commands using the up- and down-arrow keys. You can edit these commands using left- and right-arrow keys (you will find that the mouse does not work). When you want to quit R, execute the quit function:

```
> q()
```

2. OBJECTS, MODES, ASSIGNMENTS

R is an "object oriented" interactive language. I am sure all the CSCE majors reading this can give me a better definition of the term "object oriented" (and I hope you will), but in my non-CSCE way of thinking it means that data in R are organized into specific structures called "objects", of different types, and these objects share similar qualities, or "attributes". Functions in R are programs which do work, creating new objects from existing ones, making graphical displays, and so on (though functions are also objects). Try to keep track of your objects' types as you work, because many functions accept only certain types of objects as arguments, or they do different work depending on what kind of object(s) you use as argument(s). For example, the following call to the **`plot`** function:

```
> plot(my.object)
```

will have different consequences depending on whether `my.object` is a vector, a matrix, a factor, etc; these terms will be explained below. When you get comfortable with R, you can (and should) write your own functions as the need arises. You can also call (e.g.) Fortran and C functions from R.

The most common object types for our purposes, in order of increasing complexity, are vectors, factors, matrices, data frames, lists, and functions. Each of these will be discussed in some detail now.

2.1 Vectors (and data modes, and assignments)

Vectors are strings of data values, all of the same "mode". The major modes are: numeric (=double-precision floating point), character, and logical (a datum of mode logical has either the value TRUE or the value FALSE, sometimes abbreviated as T and F). Some other modes are integer, complex, and NULL. R has a number of built-in data sets we can play with; to access these, we use the **`data`** function (The command **`data()`** will list all data sets in the R library `datasets`; we will learn an alternate approach in Chapter 4). Let's start with a data set with descriptives on the fifty United States: enter

```
> data(state)
```

Let's now look at one of the vectors in this data set:

```
> state.name
```

and R will list out this vector's values on the screen. To save space when printing, a vector's values are printed in rows across the screen, with the index number for the first

element of each row provided in brackets at the beginning of each row. The double-quotes on each value tell us this vector is of mode character; if we weren't sure, we could ask (try it):

```
> is.character(state.name)
> is(state.name)
```

The second command identifies all data types by which `state.name` could be classified. We can access an individual component of the vector by specifying the position in square brackets like so:

```
> state.name[40]
```

The above three statements are actually R expressions that produce vectors of mode logical, character, and character, respectively. If you want to save the object resulting from an expression for later use, assign it a name, e.g.

```
> my.state=state.name[40]
```

The preferred method for assignment was formerly the "<" character followed by the "-" character (an arrow that pointed to the left). You can use the underscore character "_" in place of these two characters if you like, but some are partial to the arrow because it is much more descriptive of what actually happens in an assignment (the right-hand-side is evaluated, and then stored under the name provided on the left-hand-side). In recent times, the use of the equals sign ("=") has generally superceded both "<-" and "_", and will be used for the remainder of this tutorial (though this assignment method is often frowned upon by serious programmers).

If there is already an object in your workspace with the name `my.state`, this statement will replace it—without warning. Don't worry about replacing or damaging built-in R functions or data; they do not reside in your workspace, so you can't change them. Look at your new object `my.state` now by entering its name at the prompt.

If you would like to both save *and* display an object (not available in earlier versions of R), enclose the assignment statement in parentheses:

```
> (my.state=state.name[40])
```

Any given object usually has associated attributes, which are descriptive information about the object. With vectors, one possible attribute is a **`names`** vector. For example, let's load another built-in data set:

```
> data(precip)
```

and now look at it:

```
> precip
```

This is a numeric vector of average annual precipitation amounts in 70 U.S. cities. Each value in the vector has an associated name (the city), which you see printed above the value. If you want to access the names themselves, you can do so with the **`names`** function:

```
> names(precip)
```

The result of this command is a character vector, which you could store separately and work with if necessary. Names can be very useful, e.g. in scatter plots, or to help remember if a vector has been sorted. The names can also be created or reassigned by

using the **names** function on the left-hand side of an assignment (and including a character vector on the right-hand side); see Chapter 5 for a demonstration.

The **length()** function for vectors is fairly self-explanatory, but often useful in, e.g., function writing. Try it by entering

```
> length(precip)
```

The **c()** function combines values (or vectors, or lists) and is a simple way to create short vectors by typing in their values. Try this:

```
> blastoff = c(5,4,3,2,1)
```

Now look at the vector blastoff by entering its name at the prompt.

```
> blastoff
```

Another way to generate this particular vector is with the integer sequence operator "**:**". Try each one of these:

```
> 1:100
> 3:6
> 10:(-100)
> 5:1
```

We can combine /nest several functions, arithmetic operators, etc. in expressions; for example:

```
> seq.length=length(seq(-10,10,0.1))
```

applies the **length()** function to the vector returned by **seq()**; the vector itself is not saved. Look at `seq.length` now.

Note the vectors in R differ from vectors in matrix algebra in that they have no orientation: an R vector is neither a row vector nor a column vector. This can cause problems in matrix arithmetic if you're not careful.

2.2 Factors (and coercion)

A factor object is superficially similar to a character vector. Let's look at one:

```
> state.region
```

These are regions of the U.S. corresponding to each of the 50 states already seen. You see no quotes around each element, which is a clue that `state.region` is not a character vector. To see its attributes, type

```
> attributes(state.region)
```

And you should see vectors called **levels** (the four unique values that occur in the factor) and **class**, which tells you that `state.region` is indeed a factor object. Factor objects are required for ANOVA or similar analyses. Also, most data input functions will by default read any column having non-numeric data as a factor object. If you want to change a factor object into a character vector, for example to use it as plotting symbols on a graph, you can "coerce" it to be character with the **as.character()** function:

```
> regions.cvec=as.character(state.region)
```

This creates the character vector `regions.cvec` (look at it). Note that if `state.region` was something that could not easily be changed into a character object, this statement might produce an indecipherable error message and abort, or produce a NULL object. There are many coercion functions for forcing objects of one class/mode to a different class/mode: **`as.numeric()`, `as.vector()`, `as.matrix()`, `as.data.frame()`,** and so on.

2.3 Matrices

A matrix is a two-dimensional array of values having the same data mode. The rows are the first dimension; columns are the second dimension. Here's a built-in numeric matrix:

```
> state.x77
```

This is a 50x8 (50 rows, 8 columns) numeric matrix - the words you see on screen are not part of the matrix's values – they are row and/or column names. To see this matrix's attributes, type

```
> attributes(state.x77)
```

and you see three vectors displayed on screen. The first is the matrix's dimensions (a two-element vector "dim"). You can see it or conjure it up directly with the **`dim()`** function:

```
> dim(state.x77)
```

Any matrix has row and column names (though they may only be numbers), but these are referred to as the **`dimnames`** of the matrix – the rows are the first dimension of the matrix, so **`dimnames(`**`state.x77`**`)[[1]]`** are the row names of this matrix. The columns are the second dimension, so **`dimnames(`**`state.x77`**`)[[2]]`** are the column names. These are character vectors. Look at the column names of the matrix `state.x77` now.

Be careful with the **`length()`** function when its argument is a matrix; **`length(`**`my.matrix`**`)`** is the total number of elements in the matrix, not its row or column dimension. R has a rich collection of matrix/linear algebra functions; for a sampling, see the Arithmetic section of this handout.

You can reference individual matrix elements by providing row and column indices in brackets, e.g. the element in row 3, column 5:

```
> state.x77[3,5]
```

You can also reference entire rows and columns, e.g. row 3:

```
> state.x77[3,]
```

or column 5:

```
> state.x77[,5]
```

or columns 3 and 5:

```
> state.x77[,c(3,5)]
```

or exclude column 5 with a "- sign:

```
> state.x77[,-5]
```

Note that the results of the above statements are not necessarily matrix objects, though; they may be vectors. You can also reference elements, or entire rows or columns, using dimension names; e.g.

```
> state.x77[,"Murder"]
```

for the fifth column. More on referencing and extraction of elements later!

2.4 Data Frames

A data frame is similar to a matrix, but its columns may be of different modes (though data must be of the same mode within each column). It is analogous to a SAS data set. Some very important functions will operate only on data frames. Let's create a data frame using the **`data.frame()`** function:

```
> state.dfr =
   data.frame(state.name,state.region,state.abb,state.x77)
```

This creates a data frame from the existing vectors of equal lengths but varying modes `state.name`, `state.region`, and `state.abb`, and the numeric matrix `state.x77`. Type the new data frame's name to look at it:

```
> state.dfr
```

When objects get very big, it's easier to inspect them via plots, descriptives like **`dim()`**, or by looking at attributes:

```
> attributes(state.dfr)
```

You can obtain a summary of the variables in a data set in the obvious way:

```
> summary(state.dfr)
```

The printed information summary for each variable depends on the format of the variable.

Notice that the column names of a data frame are just called "**`names`**", and the row names are called "**`row.names`**". These can be conjured up as working vectors using the **`names()`** or **`row.names()`** functions:

```
> names(state.dfr)
> row.names(state.dfr)
```

The **`names()`** function actually identifies individual cells in a matrix, and not its columns! The individual elements, rows, or columns of a data frame can be referenced and/or extracted using the square-bracket indices (numbers or names) as we did for matrices. Alternately, columns can be referenced using two-level names connected with a dollar sign like `dfrname`**`$`**`colname`. Using the column name alone will not usually work (unless you attach the data frame first – see "Managing your Objects" in Chapter 4). Try the following two statements:

```
> Population
> state.dfr$Population
```

Of course, you can still access the factor object `state.region` because it still exists in the workspace (separately from the data frame `state.dfr`). Notice something else:

```
> is.character(state.dfr$state.name)
```

This is now false! It was a character vector a moment ago, wasn't it? Now try

```
> is.factor(state.dfr$state.name)
```

The **data.frame()** function coerced the character vectors to be factor objects while creating state.dfr! To prevent this from occurring, you may use the **I()** function, which tells R to leave the object "as is":

```
> state.dfr= data.frame( I(state.name), state.region,
     I(state.abb), state.x77)
```

2.5 Lists

Lists are like glued-together strings of objects of possibly very different structures, lengths, and modes. We have already seen some lists – the attributes of any object are a list. Let's extract this list from our new data frame `state.dfr`:

```
> attlist.state.dfr=attributes(state.dfr)
> attlist.state.dfr
```

The individual elements of any list can be accessed/extracted by multiple level names using dollar signs, like `listname$elementname`. Alternatively, the first element of a list can be referred to using double square brackets: `listname[[1]]`, the second element by `listname[[2]]`, and so on. Beginning users in R have a hard time distinguishing when to use single brackets and double brackets—remember what we said about R's learning curve! The elements of a list can themselves be lists, in which case you might use a double-dollar-sign reference like this:

```
> list1$list2$myvec
```

You might use this if you wanted to reference the vector `myvec`, an element of the list `list2`, which in turn is an element of the list `list1`.

Most of the more sophisticated computational and data analytical functions return lists of objects as their basic output. For example, the **eigen()** function: if A is a nonnegative definite matrix,

```
> elistA = eigen(A)
```

is a list with elements `elistA$values` (a vector of eigenvalues of A) and `elistA$vectors` (a matrix of eigenvectors of A).

2.6 Functions

Most work in R is accomplished by functions. Functions can be very strange and mysterious beasts; to fully understand what a function does, you may need to read (and reread, and reread) its help file (see below), and probably experiment with the function.

Typically, you pass a function some objects (its arguments), and from these it creates and returns one (1) new object (which may be several created quantities concatenated into a list). There are exceptions, though – for example, the **q()** function to quit R usually receives no arguments, and many plotting functions create only graphs. If any object is returned from a function, it will not be saved unless it is assigned a name. Any intermediate calculations the function performs are not saved after the function terminates, period.

A full, unabbreviated function call, with assignment of an object name `new.object` to the function's output, usually looks somewhat like this:

```
> new.object = fctname(arg1=object1, arg2=object2,…)
```

The objects passed might include numeric or character constants, names of objects in the workspace, or evaluable expressions. Some or all of the arguments may be optional – optional arguments, if unspecified, will have default values supplied by the function. These default values may be expressions involving other arguments.

As an example, consider the function **`seq()`** to generate evenly spaced sequences. It has no required arguments, but calling **`seq()`** alone generates a sequence vector from 1 to 1 by 1, not usually very useful. The most important arguments to **`seq()`** are **`from, to, by,`** and **`length`**. The arguments **`from`** and **`to`** specify the endpoints of the sequence (which will be decreasing if **`from`**'s value is larger than **`to`**'s). Usually, either the argument **`by`**, which specifies the incremental spacing in the sequence values, or **`length`**, which specifies the length of the sequence vector, is also specified (but not both, since the value of **`by`** will determine **`length`** and vice versa). Each of the following will generate (but not save) a vector (-2.0, -1.5, -1.0,…,1.5, 2.0). Try them:

```
> seq(from=-2.0, to=2.0, by=0.5)
> seq(from=-2.0, to=2.0, length=9)
```

Abbreviated function calls omit most of the "arg=" junk and just list the argument values, in order:

```
> seq(-2.0, 2.0, 0.5)
> seq(-2.0, 2.0, length=9)
```

Notice that in the second call we had to say **`length=`**`9` because **`length`** is not the third argument in the argument list for **`seq`**. Try that last command again without the words "**`length=`**". Can you see the errors waiting to happen here?

As was previously mentioned, functions are sometimes generic, which means that they do different work depending on what kind of object(s) are passed to them. A simple example is the **`diag`** function in linear algebra. If `A` is a matrix, **`diag(`**`A`**`)`** returns a vector which is the main diagonal of `A`. If `A` is a vector, **`diag(`**`A`**`)`** returns a square matrix with the vector `A`'s values on the main diagonal.

2.7 Other Object Types

Also worth mentioning are so-called "ordered" objects, which are like factor objects except that the levels of the factor are naturally ordered (like Freshman, Sophomore, Junior, etc). There are also time series objects, which are like numeric vectors but with some special attributes which are helpful / needed for time series analysis. Other important types of objects include arrays, which are generalizations of matrices to more than 2 dimensions. There are lots of other object types!

3. GETTING HELP

Perhaps the most important skill to be developed early is reading help files. If you know the function name or topic which you are curious about, enter

```
> help(topic)
```

or

```
> ?topic
```

Here, `topic` is usually a function name, though entering a built-in data set's name gives you background information ("metadata") on the data set. Let's look up information about the function **rnorm**, which generates normally distributed pseudo-random variables:

```
> help(rnorm)
```

Each function help file starts with a general Description of what the function does, followed by lines showing the function's arguments. In this case, we see

```
rnorm(n, mean=0, sd=1)
```

Arguments which are required are listed first, with no "=" signs. In this case, rnorm will generate a vector of length n whose elements are realizations of normally distributed pseudo random variables with expected value specified by the argument "**mean**" and standard deviation by the argument "**sd**". We must specify the length of the vector, **n**, or the function terminates with an error message. If we do not specify values for the mean and/or sd, they are taken to be 0 and 1 respectively by default. Try it: the command below generates a vector of 30 pseudo-Normal random variable values with mean 100 and standard deviation 10, and constructs a Normal quantile plot of these (the vector is not saved). If the values indeed come from a Normal distribution, the plot should approximate a straight line.

```
> qqnorm(rnorm(30,100,10))
```

The helpfile section titled Value explains what sort of object (if any) is returned by the function, in this case a numeric vector. There may be a Details section that gives some explanation as to how the function accomplishes its work; there may also be References. One of the most important parts of the help file is the See Also section, which provides cross-references to other functions which are related to the one whose help file you're reading. Often the function you're reading about may not do exactly what you want, but one of the cross-referenced functions does. At the very bottom of the help file there are usually some Examples; these are often less helpful than they could be.

Certain important help files are hard to find because they have non-intuitive keywords. For example: `help(Logic)`, `help(Syntax)`, `help(Comparison)`; most people wouldn't guess to capitalize the first letter for these important help topics, but R is case sensitive, so if you type `help(logic)`, it will reply that there is no documentation for this topic. In addition, some words, particularly operators, need to be enclosed in quotes to obtain help (`help("if")`, `help("%*%")`). Typing

```
> help.start()
```

links you to R's documentation web-page with its links to manuals and FAQs. The information here is too general to help with a specific request.
You can also type, e.g.,

```
>help.search("logic")
```

This command will generate a list of related R functions and the libraries containing them; perhaps the function you're looking for will be in this list. Similarly,

```
>apropos("logic")
```

will provide a list of all R commands that contain the string "logic". Note that **`apropros()`** suffers from the same shortcoming as a regular **`help()`** search—it's case sensitive.

Previously, if you typed a function name without parentheses, R would display the object itself: the function code, line for line, on the screen. It will still do this for functions you yourself have written, but R's functions are now listed as Internal documents as though the code is now proprietary.

4. MANAGING YOUR OBJECTS

If you are continuing with R from an earlier section of this primer, please quit at this time, without saving your workspace image, and restart R to get back to an empty workspace.

The objects you will work with reside in a number of directories; when you refer to an object, R tries to find it by searching a path of directories in a given order. To see the first few current search path directories in order, type

```
> search()
```

The first directory, **.GlobalEnv**, is the current workspace. Any object which you create and assign a name to will be stored there. The other directories shown at this time in the search list are built-in directories containing built-in R functions, data sets, and so on.

To see a list of the workspace contents, type

```
> ls()
```

If you see the word **character(0)**, this means your workspace is currently empty; this should be the case if you just started up R for this section and did not save your workspace from a previous session. Let's load a few things, so we have something to work with:

```
> data(state)
> data(mtcars)
```

Now use **ls()** to check the workspace contents again. You can see a higher level directory's contents using the **objects()** function, e.g.

```
> objects(6)
```

will display the object names of the 6th directory of the search path. You may also refer to any directory in the search path by its name, e.g.

```
> objects(package:base)
```

You can modify your search path for the current interactive session. For example,

```
> attach(mtcars)
```

attaches the data frame **mtcars** in position 2 of the search path, which (like all positions above the workspace) is read-only. You can also attach lists or previously created Rdata directories (using a full file specification, in quotes) which may contain data objects and/or functions you created in earlier work sessions; we'll do that later. Now, look at the objects you loaded into position 2. Now that **mtcars** is attached, you can access its columns (variables) without clunky two-level $-names. When you want to remove `mtcars` from the search list (don't do it now), type

```
> detach(mtcars)
```

or

```
> detach(2)
```

or, simply quit R.

Before going any further, we should clean up our workspace a bit, since it is dangerous to have the data frame **mtcars** both in the workspace and in the search list (you can change the copy in your workspace, but the older version would still be attached as a

read-only data frame). We can remove it from the workspace using the **rm()** function; while we're at it, let's remove some other things, too:

```
> rm(mtcars, state.area, state.division, state.center)
```

Now use **ls()** and **search()** again to verify that **mtcars** is not in the workspace, but is still attached in the search path. Use **rm()** often to keep your workspace free of old objects which you will never use again. Note that **rm()** removes objects one-by-one and does not use wildcards; because of this, cleaning up a workspace can be tedious if you have allowed objects to accumulate.

We can save the contents of the current workspace to use as a workspace in a later session, or to attach it as a read-only directory in another session. To do this, go to the File menu and choose "Save Workspace". Change directories to find a folder where you would like to store this new workspace, for example, in a previously-created stat517 folder "C:/My Documents/classes/stat517". Then give the workspace a name, e.g. "testing123" and click Save. Note that the workspace is saved as an "R images" file, which has extension ".Rdata". In the folder where you stored it, it will henceforth be visible with an R logo. Its full file specification in this example is C:/My Documents/classes/stat517/testing123.Rdata

When you quit an R session, R will prompt you to save the workspace, even if you have just saved it as a named workspace. You should answer "No" if you have just saved the workspace under a full name; otherwise, an additional workspace labeled ".Rdata" will be saved as well.

If you want to use a saved workspace as your workspace in a future session, there are several ways to accomplish this. The easiest way is to launch R directly at that location by double-clicking the R logo next to the file you want to use as a workspace. If instead you choose to launch R in some other way, go to File, then Load Workspace, and then find the workspace you want to use. A third way is to load the previously-created workspace into position 1 with the **attach()** function; in the above example, this is accomplished by

```
> attach("C:/My Documents/classes
      /stat517/testing123.Rdata",pos=1)
```

This (apparently) replaces the current workspace with the specified one. Note that only .Rdata files can be loaded in position 1 (no data frames or lists).

If in the future you want to use the objects in a previously created workspace as read-only objects (for example, if they are functions that you want to use but not change), use the above command with **pos=**2 (or with no position specified), and it will be attached as read-only in position 2.

It is usually a bad idea to keep the same function or data object in more than one directory. If you change one of the copies, you may forget to change the other, and at some later date you will be referring to the object and not know which of the two you're working with. Unfortunately, it seems that R will not warn you about this when you create, say, a function in your workspace when there is a function of the same name in

another directory of the search path (Splus would provide such a warning). It is also a bad idea to have two functions of the same name in the search path; when you call the function, R will use the first function of that name it finds in the search path. Likewise, if you use a data-object name that is also being used for a built-in function, e.g. "**`plot`**", whenever you call the function `plot` you may see a message like this one:

```
     Looking for object "plot" of mode function.  Ignored
one of mode numeric…
```

5. GETTING DATA INTO R

5.1 Creating Data

First, there are a number of handy functions for creating vectors and matrices in R. We have already seen the **`c()`** and **`seq()`** functions, and the integer sequence "`:`" infix operator. Also useful is the **`rep()`** function, which creates a vector by repeating a single value a specified number of times..try these:

```
> rep(1,10)
> rep("Gamecocks Rule",1000)
```

We can easily generate vectors of pseudo-random numbers in R; we have already examined the **`rnorm()`** function for generating Normal random variates; there are a number of others, including **`runif()`**, **`rchisq()`**, **`rbin()`**, etc.; enter

```
> help.search("distribution")
```

to see a partial list.

Another useful data-creating function is the **`matrix()`** function. The command

```
> constmat = matrix(value,r,c)
```

creates and stores a matrix with r rows and c columns, with every entry equal to `value` (which may be of any mode). Examples:

```
> zeromat = matrix(0,5,3)
> onesvec.col = matrix(1,10,1)
```

Note the difference between `onesvec.col` and `rep(1,10)`; the former is a "column vector" in the matrix arithmetic sense – a matrix with only one column.

The matrix function can also reshape long vectors into matrices. For example, suppose `bigvec` is a vector of length 200, and we would like to reshape it into a 100 row, 2 column matrix. An important question: should the vector's elements be written into the matrix a row at a time, or a column at a time? The default is to write by columns, but if the desire is to write by rows, add the option `byrow=T`:

```
> bigvec = 1:200
> mat.bycols = matrix(bigvec,100,2)
> mat.bycols
> mat.byrows = matrix(bigvec,100,2,byrow=T)
> mat.byrows
```

5.2 The `read.table()` and `read.delim()` functions

More often than not, a data analyst will need to import data which is in roughly columnar form into R as a text file, perhaps created by some other software. If each data record consists of the same number of items, on a single line, the function **`read.table()`** for creating a data frame from the text file has long been highly recommended. This section will discuss **`read.table()`** in detail, but **`read.delim()`**, which has similar

features, is often more robust and should be used first when handling tab-delimited or comma-delimited (see **`read.csv()`** as well) data sets. Note that we will need missing values in the text file to be represented by some unambiguous missing value code, and values on each line separated by some well-defined delimiter (usually blanks, tabs, or commas). Items need not be in exactly the same columns on different lines. It is recommended that you read (and reread) the help file for **`read.table()`** carefully, to find out exactly what its capabilities and limitations are.

To use **`read.table()`** most effectively, let the first item of each record be a character variable to be used for row names; any names that have embedded blanks should be enclosed in quotes. Then, add a row at the top of the text file with the column names you desire, but no name for the row names column. If each record's data values are separated by white space or tabs, and missing values are represented by the R missing value code NA (which stands for "not available"), then the command

```
> my.dfr = read.table("filespec")
```

should create a data frame called `my.dfr` with the desired column and row names. Important note: the file specification `filespec` should use double *forward* slashes wherever a normal file specification would use single *back* slashes, e.g. `E://myfile.txt` for a text file `myfile` stored on a thumb drive. If the file specified by `filespec` is not in the directory where the current workspace resides, then the specification must include the full directory path to the file. This problem can be avoided entirely by changing the default directory; select *File* then *Change dir…* from the R toolbar, and then choose the directory that contains, e.g., `myfile.txt`. The usual default directory would be a subdirectory with the set of R program files—it is generally convenient to change this directory to your personal workspace at the start of each R session.

For example, suppose the text file below, `brainbod.txt` , which lists typical body weights (kg) and brain weights (g) for 15 terrestrial mammals, exists in the same folder where R was opened. Note there are three items in every row, the first being a unique name for the row. Since the first record has one less item, these are presumed to be the **`names`** for the data frame to be created. Note the missing value in row 6, column 2.

	bodywt	brainwt
afele	6654.00	5712.00
cow	465.00	423.00
donkey	187.00	419.00
man	62.00	1320.00
graywolf	36.33	119.50
redfox	4.24	NA
narmadillo	3.50	10.80
echidna	3.00	25.00
phalanger	1.62	11.40
guineapig	1.04	5.50
eurhedghog	0.79	3.50

chinchilla	0.43	64.00
ghamster	0.12	1.00
snmole	0.06	1.00
lbbat	0.01	0.25

The command

```
> brainbod = read.table("brainbod.txt")
```

will create a data frame with two columns, names `bodywt` and `brainwt`, and **`row.names`** present. Note that if there had been any character data other than the row names in the text file, their columns would have been converted into factor objects. To preserve character columns as character objects, add the option **`as.is=`**`T` to the **`read.table()`** call. It is a highly recommended exercise to copy the above table, paste it into a text file, and try to read it into R as a data frame as described above.

Usually, missing values in a raw text file are not represented by the R missing value code NA. If this is the case, there is an option **`na.strings`** in **`read.table`** that allows specification of a different missing value character, e.g. the naked decimal " ." in SAS: **`na.strings=". "`**. Note there is a blank before the decimal in this specification, to make sure that decimals imbedded in numbers are not interpreted as missing values – this specification assumes that numeric values less than 1 are represented in the text file using leading zeros, i.e. "0.XXX". It may be obvious from this that one must BE VERY CAREFUL in reading large, complex data sets (in any language, actually). When there is a great deal of data, anything that can go wrong, will go wrong. There will probably be "typos", for example. Always check the results very carefully, and be patient.

Suppose the first entry of every row of the text file is a numeric value, not to be used as a row name. If, as above, column names have been typed as the first row of the file (and now every record has the same number of items as there are column names), try

```
> my.dfr = read.table("filespec", header=T)
```

This will generate row numbers as default row names. Or, you can add the option **`row.names`**`=charvec.row` in the call to **`read.table()`**, where `charvec.row` is a workspace character vector you would like to attach to the data frame as row names. Of course, you can also change row names easily after the fact with a command like

```
> row.names(my.dfr) = charvec.row
```

If for some reason the (column) **`names`** of the data frame cannot be included as the first row of the source text file, default **`names V1, V2,`** etc. will be generated; to avoid this, you can use the option **`col.names`**`=charvec.col` in the call to **`read.table()`**. Or, change them after the fact with something like

```
> names(my.dfr) = charvec.col
```

5.3 The `scan()` function

In rare cases of very complex data to be read into R, the **data.frame()** function may not have the needed flexibility. In these instances, the **scan()** function, more flexible but more difficult to use, may fill the bill. The help file is difficult to read, but worth the effort(s) if you will be inputting unwieldy data. Useful optional arguments to **scan()** are the **what, multi.line,** and **widths** arguments. For example, suppose we must read a data set with several thousand records, where each record has 50 values, some numeric, some character, in variable-width fields, and each record is written in several lines. After a careful search-and-change in the text file to change missing values in the data to NA, the following generic call should work:

```
> my.dfr = as.data.frame(scan("filespec",
+ what=list(name1=0,name2="",name3=0,……,name50=""),
+ multi.line=T))
```

Here, the "+" signs are supplied by R as a continuation, whenever you type a statement which reaches beyond the end of the command window. The **what** argument in this case supplies a model for a single dummy record of the text file, with variable names: `name1=0` means the first value in each record of the file is numeric; `name2=""` means the second value is character, and so on. Since the object assigned to **what** is (in this case) a list, **scan()** creates a list from the text file with element names `name1, name2`, and so on. The **as.data.frame** function converts this list into a data frame; the list element names will become the names for the data frame `my.dfr`. The **multi.line=**`T` option causes **scan** to ignore line breaks in the text file.

If the data in the text file are in fixed-column format, in which case there are often no missing values represented in the data, the widths argument would probably be needed. To use it, include **widths=c**`(3,4,2,…)` anywhere in the scan argument list, where the values inside the `c()` are of course the field widths for each data record. You may also want to include the option **strip.white**=T so that character variables' values will not include leading or trailing blanks.

Interestingly, **scan()** is also taught as the simplest method for entering data, as it can readily be used to read a single column from keyboard input or a text file as an R vector object.

6. GETTING RESULTS OUT OF R

This can be surprisingly annoying. The most common situation seems to be the need to output a matrix or data frame in R to a text file. A command like the following used to be the standard approach:

```
> write(t(mymat),"mymat.txt", ncol=ncol(mymat))
```

Here, `mymat` is an existing matrix or data frame and "`mymat.txt`" is the desired name for the text file to be created; it will be created in the current directory unless you specify a different directory path as part of the file specification. The "**t**" is the transpose operator, which was a necessary annoyance; the transpose of `mymat` is saved otherwise. If the values of `mymat` are floating point numeric values, you may want to substitute, e.g. **round(t(**`mymat`**),digits=2)** in place of **t(**`mymat`**)** to round values to two decimal places, or **signif(t(**`mymat`**),digits=6)** to write only the first 6 significant digits. The **ncol** argument is needed in **write**; if you leave it off, **write** will simply write 5 values per line.

R has recently implemented **write.table()**, which does *not* transpose a matrix or data frame, and also does not require a **ncol** statement. It saves row and column names by default; for something as simple as our matrix object mymat, we will likely want to use the following form:

```
>write.table(mymat,"mymat.txt",row.names=F,col.names=F)
```

Another common situation is one in which a user wants a text file representation of a function. This can be accomplished easily using the **fix()** function:

```
> fix(myfct)
```

This opens a text editor (Notepad) window to allow one to modify the function code – simply go to File, then Save As, and type the desired name for the text representation. Then go back to File and Exit. If you would like to do this for several functions / objects without taking the time to deal with them individually, the **dump** function will get everything out quickly, though you may not like what you get:

```
> dump(c("object1","object2",…),"miscstuff.txt")
```

Often, especially in debugging a function, you will want to print messages to the screen or an output file. The **print()** function will print any character or numeric object (by default) to the screen, or to a textfile if a file specification is given. To combine character and numeric objects into one message, use **paste()** to glue them together into a character string:

```
> print(paste("Now starting iteration number",iter))
```

Here, `iter` would probably be an integer-mode counter inside a loop; **paste()** converts the numeric value to character, binding it with the first character string.

Other functions designed to get results out of R include **dput()**, **sink()**, **pdf()**, and **postscript()**—for plots and other graphics.

7. ARITHMETIC

For all details, reading the **help(Arithmetic)**, **help(Math)**, and **help(Syntax)** files is recommended, along with experimentation with the operators discussed here. The infix operators for addition (+), subtraction (-), multiplication (*), division (/), exponentiation (^), and modulo arithmetic on integer objects (%%) are most common, and will be discussed here, though standard "calculator" functions and extension (found under **help(Math))** will also receive frequent use. Operator precedence for infix operators is fairly standard, and is spelled out in the help files; use parentheses liberally to be safe. The general use of these infix operators is

```
> new.object = (object1 op object2)
```

where **op** is an arithmetic infix operator and `object1, object2` are integer, numeric or complex objects such as vectors, matrices, or arrays. If these have the same length or dimensions, the result stored in `new.object` has the length or dimensions produced by elementwise application of the operator to the two objects. If `object1, object2` are not of the same length or dimensions, the smaller one is "recycled as necessary". This can be a nice feature, but can lead to major errors without error messages if one is not careful. For example, in the expression

```
> new.object = (object1 ^ object2)
```

if `object1` is a matrix and `object2` is a scalar (a vector of length 1), then `new.object` is a matrix of the same dimensions as `object1` obtained by exponentiating all of its elements to the power given by `object2`. If `object2` is a vector of length 2, elements of the matrix are alternately exponentiated by the first or second element of the vector.

That having been said, a little discussion regarding matrix arithmetic is appropriate. The **t()** function transposes a matrix (exchanges rows and columns); though vectors in R have no orientation, **t()** converts an R vector to a row vector. The command

```
> colvec = as.matrix(myvec)
```

coerces the R vector `myvec` to be what humans call a column vector. The **matrix** and **diag** functions for creating matrices have already been mentioned; if the argument to **diag()** is a positive integer, an identity matrix of that size is created. Try the command **diag(10).** If `A` and `B` are matrices and the usual matrix product `AB` is defined (if and only if number of columns of `A` = number of rows of `B`), then this product is obtained by `A%*%B`. If `b` is a vector such that the system of equations `Ax=b` is well defined and has a solution `x`, then **solve(A,b)** will provide one. If `b` is a matrix, the solution is obtained by solving the systems corresponding to each column of `b`. The inverse of a nonsingular matrix is obtained by

```
> Ainv = solve(A)
```

The functions **nrow()**, **ncol()**, and **dim()** produce simple numeric objects describing the size of a matrix supplied as their argument. The functions **rbind()** and **cbind()** concatenate matrices by rows or by columns, respectively. These also can form matrices from vectors, if all vectors have the same length. e.g.

```
> newmat = cbind(vec1,vec2,vec3)
```

is a matrix with **`length`**`(vec1)` rows and three columns. For other functions relevant to matrix arithmetic, please refer to the **`help()`** files. Most matrix algebra operations, expressions, etc. work with purely numeric data frames as well. If not, try (carefully) coercing the data frame to be a matrix using **`as.matrix()`**.

8. LOGICAL OBJECTS AND CONDITONAL EXECUTION

An object of mode logical is a vector, matrix, array etc. whose elements consist entirely of **TRUE** or **FALSE** values (sometimes abbreviated just as **T** and **F**). To see one, load some data

```
> data(mtcars)
```

and try

```
> guzzlers = (mtcars$mpg < 20)
> names(guzzlers) = row.names(mtcars)
> guzzlers
```

The first of these three commands creates the logical vector `guzzlers`, with length equal to the length of `mtcars$mpg`, whose value is **TRUE** if the corresponding car (row) in **mtcars** gets less than 20 miles per gallon of fuel, and **FALSE** otherwise. The other commands attach names to the vector `guzzlers` and display the named vector on screen.

Logical objects are invaluable for subsetting data (Ch. 9) and for conditional execution of statements using **if()** clauses. Most logical objects are generated with a statement like

```
> logical.obj = object1 comparison.op object2
```

where `object1` and `object2` are usually numeric constants, vectors, matrices, or arrays (though character objects can be used for alphabetical comparisons) and **comparison.op** is one of the following logical infix operators:

> < >= <= == !=

The latter two operators above are strict equality and non-equality, respectively; note that "**=**" is not a comparison operator. Be sure to leave a space between "**<**" and a negative sign in comparisons, or use parentheses to separate these. As in arithmetic expressions, if `object1` and `object2` are the same length or dimensions, the comparison is done elementwise and the created logical object is of the same length or dimensions. If `object1` and `object2` are not the same length or dimensions, the smaller object is "recycled as necessary", generating a logical object whose length or dimensions match the larger of `object1` and `object2`. In particular, if `object2` (e.g.) is a scalar, the command compares every element of `object1` with that scalar and returns a collection of **TRUE/FALSE** results with the same size and structure as `object1`.

Logical objects can be combined in a number of ways. If `logical.obj` is a logical object, then `!logical.obj` is its complement, i.e. an object of the same size and structure as `logical.obj` with **TRUE** and **FALSE** reversed in each position. If `logical.obj1` and `logical.obj2` are two logical objects, then

```
> logical.obj1 & logical.obj2
```

(**&** is the "and" operator) evaluates to **TRUE** in every component where the compared elements of `logical.obj1` and `logical.obj2` are both **TRUE** (if these objects are not of the same length or dimensions, the smaller one is used cyclically, cha cha cha). In contrast,

```
> logical.obj1 | logical.obj2
```

(**|** is the "or" operator) evaluates to **TRUE** in every component where at least one of the compared elements of `logical.obj1` and `logical.obj2` is **TRUE**. There are also "sequential-and" and "sequential-or" operators **&&** and **||** which have similar results but can be more efficient since the second comparison is not carried out if the first comparison is tested and found to be false.

There are several important functions which use and/or return logical arguments. We have already seen the query functions **is.matrix**, **is.character**, etc., which return a scalar **TRUE** or **FALSE**. Other important functions involving logical objects include **all(), any(), all.equal()** and **identical().** For example,

```
> all(logical.obj)
```

evaluates to a scalar **TRUE** if and only if every component of `logical.obj` is **TRUE**. Similarly,

```
> any(logical.obj)
```

evaluates to a scalar **TRUE** if and only if at least one component of `logical.obj` is **TRUE**. If you would like comparisons involving missing values to be ignored, add the argument **na.rm=T** inside the parentheses.

The function **identical()** compares any two R objects and returns a scalar **TRUE** if and only if they are identical in every respect (including structure, mode, etc). For example,

```
> identical(1., as.integer(1))
```

is **FALSE** because the second object is not a floating point numeric value; **identical()** will also signal false when comparing a matrix and a data frame, even if their values are identical when compared componentwise.

The operator **==** and the function **identical()** may signal **FALSE** when comparing numeric objects which are equal up to roundoff error (i.e. equal up to machine precision); this can be undesirable. The function **all.equal()** compares two objects testing for "near equality" - the tolerance for judging equality of numeric objects can be adjusted (see the **help()** file). If it finds no "real" differences, it returns a scalar **TRUE**; otherwise, it returns a character vector describing the differences found.

One can use logical expressions in an **if()** clause to place a condition on execution of statements or groups of statements. A generic **if()** clause might look like this:

```
if(logical.condition) {
          statement1
          statement2
          } else    {
          statement3
          statement4
          }
```

Here, if `logical.condition` is `TRUE`, the statements in the first group of brackets are executed; otherwise, the statements in the second group of brackets are executed. If there is only one statement in a group, the brackets are not necessary. The **else** clause

is not required if action is only to be taken when `logical.condition` is `TRUE`. When using the **`else`** clause, always make sure the **`else`** statement appears on the same line as the right-hand brace (as shown above). Otherwise, R completes execution of the **`if`** statement, and can't interpret **`else`** correctly.

The `logical.condition` should evaluate to a scalar `TRUE` or `FALSE`; if it evaluates to a logical vector or matrix, only the first element will be checked to decide if the condition holds (mercifully, with a warning message). For example, if you would like to execute an **`if()`** clause only if all elements of the numeric vector `testvec` are zero (within machine epsilon), these are all bad ways to do it:

```
> if(test.vec == 0) …                (condition evaluates to a vector)
> if(all(test.vec == 0)) …           (tiny roundoff error causes FALSE )
> if(all.equal(test.vec,0)) …        (all.equal can return char. vec.)
```

The recommended technique for this sort of conditional execution is to nest **`all.equal`** with **`identical`**:

```
> if(identical(all.equal(test.vec,0), TRUE)) …
```

The **`ifelse()`** function conducts mass-production conditional execution over a vector or array, making it one of the most commonly used functions – it is absolutely invaluable to avoid looping. You may be familiar with it from using it in Excel. A typical call looks like

```
> new.object=ifelse(logical.obj,expression1,expression2)
```

where `logical.obj` can be a vector or matrix of **`TRUE`**s and **`FALSE`**s, or an expression that evaluates to one. The created `new.object` is an object of the same length or dimensions as `logical.obj` with the result of `expression1` found wherever `logical.obj` is **`TRUE`**; otherwise the result of `expression2`. If the expressions do not generate objects of the same size as `logical.obj`, they are repeated cyclically. Here is a useful example, a command to replace missing values in a numeric object `my.object` with zeros:

```
> my.object=ifelse(is.na(my.object),0,my.object)
```

Here is another example which creates a "truncated" mpg vector, replacing mpg values greater than 20 with the value 20:

```
> mmp.truncated =ifelse(mtcars$mpg>20,20,mtcars$mpg)
```

Logical objects can also be used in arithmetic expressions: the numeric value 1 replaces **`TRUE`** and 0 replaces **`FALSE`**. For all details on logical objects and operators, reading the **`help(Syntax)`** and **`help(Logic`**) files is highly recommended!

9. SUBSETTING, SORTING, AND SO ON…

In Chapter 2: OBJECTS, MODES, ASSIGNMENTS, we introduced a few simple ways to subset vectors and matrices (and data frames, and arrays…) using brackets `[ ]` with names or index numbers inserted to extract a vector element or a row or column from a matrix. This technique can be generalized considerably. Suppose we want to select some of the elements of an existing vector `oldvec`, and suppose `index.vec` is a vector giving the indices (numeric positions) of desired elements to extract. The following statement

```
> newvec = oldvec[index.vec]
```

Will extract and store the desired elements in `newvec` in the order that they occur in `index.vec`. For example, load the state data set again:

```
> data(state)
```

and extract the last five states in reverse alphabetical order:

```
> tryit = state.name[50:46]
> tryit
```

A minus sign on an index indicates that this element is *not* to be extracted from the vector. For example,

```
> state.name[-(31:40)]
```

is all the states *except* those numbered 31 through 40 (so, e.g. South Carolina, number 40, would not be included in the extracted vector).

These indexing tricks can be used to easily drop rows or columns, or extract submatrices from larger matrices, e.g.

```
> newmat = bigmat[-1,]
```

is everything but the first row of `bigmat`,

```
> newmat = bigmat[,2:10]
```

is columns 2 through 10 of `bigmat`, and

```
> newmat = bigmat[1:3,1:3]
```

is the upper-left 3x3 submatrix of `bigmat`.

We can also use names to extract certain elements of named vectors, matrices, and data frames, though this is less handy when we want to extract more than one index. For example, to study data for only the Carolinas and Georgia, we might subset the matrix `state.x77` selecting three rows as follows:

```
> tristate.region = state.x77[c("South Carolina",
            "Georgia", "North Carolina"),]
> tristate.region
```

Though cumbersome, using names instead of index numbers can be a safer way to extract values when there are missing values in the data. For example, to do certain calculations, missing values will need to be removed, but after doing this the remaining data items may not be numbered in the way you expect!

A third way to extract elements, rows, and columns is through the use of logical vectors. The command

```
> newvec = oldvec[logical.vec]
```
will extract only the elements of `oldvec` for which `logical.vec` is `TRUE`, in the order in which `TRUE`s are encountered. For example,
```
> newvec = oldvec[oldvec>0]
```
extracts the positive elements of `oldvec` in the order they appear and stores them in `newvec`, which may be shorter in length than `oldvec`. Let's try to extract the rows of `state.x77` corresponding to the defined level `SOUTH` in `state.region`:
```
> south.region = state.x77[state.region=="South",]
> south.region
```
Notice that when you print the object `state.region` to the screen, the levels `South, North Central`, etc. do not appear in quotes (`state.region` is not a character vector – it's a factor), but the above logical statement does not work properly without quotes.

A number of functions can be useful in generating the information needed to generate index vectors, or to directly subset vectors and matrices. See for example the help files for **`match()`**, **`split()`**, **`sample()`**, **`unique()`**, **`cut()`**, **`table()`**, **`category()`**, **`tabulate()`**, **`grep()`**, and **`charmatch()`**.

The index-vector extraction technique is actually the basis for sorting vectors, matrices, and data frames via the **`order()`** function: let
```
> ord = order(sortvec)
```
Then `ord` is an index vector which maps the elements of `sortvec` to their locations in a sorted version (i.e., `ord` is a permutation vector). This is easier to understand with an example:
```
> ord = order(state.area)
> ord
```
and we see:
```
 [1] 39  8  7 11 30 21 29 45 20 48 40 19 14 17 46 35 …
```
which means that the smallest state in area is the state in position 39 in **`state.area`** (i.e., Rhode Island), the second smallest state is the state in position 8 (i.e., Delaware), and so on. Since these are indices, we can use the created `ord` vector to create sorted versions of objects. Try this:
```
> state.name[ord]
```
and we find out Rhode Island is indeed the smallest state in area. If you want to list the objects in reverse order of area, try
```
> ord = rev(order(state.area))
> state.name[ord]
```
and we discover that it is indeed true that California is the third largest state behind Alaska and Texas. Instead of using the **`rev()`** function, we could have done this:
```
> ord = order(-state.area)
```

We can use the **order()** function to sort an entire matrix or data frame by the elements of a given column or other vector of the same length. To sort the rows of **state.x77** in order of decreasing population, we could try this:

```
> Population = state.x77[,"Population"]
> ord = rev(order(Population))
> state.x77.popsorted = state.x77[ord,]
> state.x77.popsorted
```

The function **order()** accepts any number of vector arguments of the same length, of possibly different modes, as long as each vector can be ordered by its values. If several vectors are passed to **order()**, the output index vector is the permutation required to perform a nested sort. For example,

```
> ord = order(as.character(state.region),
                          (-Population))
```

is the permutation vector required to sort first by region (alphabetically), and then within region by decreasing Population. Try it:

```
> state.dfr =
     data.frame(state.name,state.region,state.x77)
> ord = order(as.character(state.region),
                    (-Population))
> state.dfr=state.dfr[ord,]
> state.dfr
```

Note that we need the **as.character()** function here, since the factor object **state.region** has an implied order by levels which may not be alphabetical.

There is also a **sort()** function that accepts only a single vector argument, returning the sorted version. It does not have the same breadth of application as the **order()** function.

10. ITERATION

Recall that R was built from the AT&T package S, which is in the public domain. The commercial package Splus, also built from S, existed long before R; its greatest weakness (besides its cost) is that it does not loop efficiently. R is reputed to be more efficient than Splus for looping, but nonetheless we will try whenever we can to avoid looping, since R has so many built-in iterative capabilities.

This will be more a matter of breaking old programming habits than anything else. Many tasks traditionally done in loops can be accomplished in R with a single statement, e.g. squaring every element of a 1000x200 matrix `A` can be accomplished by the statement `A^2` instead of two nested loops! Most standard functions operate elementwise on vectors, matrices, and arrays, e.g. **`sin(`**`A`**`)`**, **`exp(`**`A`**`)`**, etc. producing objects of like size and structure. With some thought, you can often determine a combination of functions that will do mass-production calculations without looping. For example, to generate a matrix of random numbers, generate a large vector and shape it into a matrix using the **`matrix()`** function. The **`kronecker()`** function (or the outer product operator **`%o%`**) is another useful mass-production function for matrix operations. Be creative.

Occasionally, a function cannot operate on entire matrices, or we would like to apply it repetitively, say, to the rows of a matrix. For these instances, the **`apply()`** function will allow its application somewhat more efficiently than loops:

```
> new.obj = apply(array, indices, fctname, …)
```

Here, the function whose name is given by `fctname` is to be applied to `array` (considering vectors and matrices as one- and two-dimensional arrays) over the indices specified in `indices`: rows=1, columns=2, etc. If `indices` is not mentioned, the function is applied elementwise over the entire array. The notation "…" in the argument list refers to the fact that **`apply`** will also accept as arguments any arguments that `fctname` needs.

Here are some examples: Suppose `myfct` can operate only on scalars:

```
> newmat = apply(oldmat,myfct)
```

applies `myfct` separately to every element in `oldmat`; the results are stored in the matrix `newmat`, which has the same dimensions as `oldmat`. A common task is to find the maximum of each row of a matrix; the command

```
> max(mymat)
```

will not do this, as it will find the single maximum value of the entire matrix. To find a vector containing the maximum value in each row,

```
> rowmaxes = apply(mymat,1,max)
```

Since the **`max`** function will by default deliver a missing value **`NA`** if it finds any missing values in its argument, you may want to add the **`max`** argument **`na.rm=`**`T` to the list:

```
> rowmaxes = apply(mymat,1,max,na.rm=T)
```

Now the function will produce a vector containing the row-wise maxima in `mymat`, ignoring missing values.

Another frequent task is to apply a function to groups in the data, the sort of thing we use a BY statement for in SAS. The **tapply()** function can do this to some extent:

```
> newobj = tapply(myvec,groups,fctname, …)
```

Here, `groups` is a factor, vector, or list of these, each the same length as `myvec` whose values or combinations of values specify subgroups for the elements of `myvec`. The function `fctname` is a function that can operate on these subgroups of `myvec`, and "…." again refers to optional arguments for `fctname`. An example:

```
> Income =state.x77[,"Income"]
> medinc.byreg = tapply(Income,state.region,median)
> medinc.byreg
```

This computes median state income by region. There are two other `apply`–like functions, **sapply()** and **lapply()**, which can apply functions to each element of a list.

If you MUST loop, try to keep the loop size as small as possible, and definitely try to avoid nested loops. Tacking new rows onto matrices in each loop, or new elements onto a list, tends to be slow in Splus, so possibly so in R as well. The syntax for looping uses either a **for** or **while** clause, usually something like this (e.g. to carry out a number of operations over each column of a matrix `mymat`):

```
> for (j in 1:ncol(mymat)) {
      multiple statements on, e.g. mymat[,j]
      }
```

or similarly using **while(**`logical.condition`**)** to determine entry to, and exit from, the loop. See **help(Logic)**.

11. AN INTRODUCTION TO GRAPHICS IN R

One of R's strongest virtues is its integrated graphical capabilities. This primer will hardly scratch the surface on this topic, though. For a quick tour, try

```
> demo(graphics)
> demo(image)
```

Several graphics devices (window types) are available. The command

```
> windows()
```

opens the default style device. You can open as many of these windows as you need (subject only to available RAM); graphical commands are by default sent to the "current window", usually the most recently opened one. If you make a mistake or for any other reason want to erase whatever is in the current window, entering **`frame()`** will do it. The command **`dev.off()`** will close all graphics windows. Let's plot something for the sake of discussion: a plot of 100 generated standard Normal deviates in the order they are generated:

```
> zvars=rnorm(100)
> plot(zvars)
```

If it is not already active, activate your window by clicking anywhere in its frame, or by choosing it under the Windows menu. Once a graphics window is activated, a different set of menus is available at upper left; a more limited set of menus is available by right-clicking. The most important options available there are under the File menu; they include

- Save As: this allows the graph to be saved as a graphics file in any of several standard formats. I generally save graphs in Metafile format for easy insertion into Microsoft documents later.
- Copy to the Clipboard: this is an attractive option if you will paste the graph immediately into another document. Again, I generally use Metafile format. You can also use **`pdf()`** or **`postscript()`** on the command line to save copies of graphs.
- Print: self-explanatory; the Properties button will allow for some flexibility in the printing, especially to allow printing to be done in either portrait or landscape mode.

R has standard ways of formatting graphics, e.g. the font styles, colors, box around the plot, etc, which you can modify. The current choices for an active graphics device can be viewed and/or modified with a call to the function **`par()`**, though it is difficult to sift through the lengthy list of available options. Let's look at the default values:

```
> par()
```

As you see, there are many of these graphical parameters; unfortunately their names are not in every case intuitive. To gain complete expertise in using R's graphics, one finds themselves reading (and rereading, and rereading) the help file for **`par()`**! The default choice of parameters can be changed by a call to **`par()`**, e.g. (don't do this please):

```
> par(pty="s",bty="l",lwd=2,pin=c(3,3),las=1)
```

will force all plots made on the graphics device to be square, and to have an "L" shaped box instead of a box completely around the plot, and to have line widths doubled over the default width, and to have a plot which is 3"x3" in size, and to have axis labels which are horizontal rather than perpendicular to the axes (the default). A given set of parameters can be saved as a list with a name and recalled later in R. Also, whatever choice is stored there can generally be overridden in the creation of any given plot using keywords and new values.

11.1 Single Variable Graphical Descriptives

Let's do a little exploring using the **state.dfr** data frame:

```
> data(state)
> state.dfr =
      data.frame(I(state.abb),state.region,state.x77)
> attach(state.dfr)
```

We will start with quick-n-dirty draft graphics for describing data distributions. A histogram of the state populations can be obtained easily:

```
> hist(Population)
```

The default title can be changed through the argument **main:**

```
> hist(Population,
      main="Population in 10,000s by State, 1977")
```

A fancier "smoothed" histogram or density plot is available with only a little more work:

```
> plot((density(Population)))
```

The call to **density** actually creates a list which includes a grid of x-values (for the horizontal axis) and the values of the density estimate (y) over each of these. See the help file for **density** to learn how to vary (e.g.) the bandwidth for calculating the smoothed histogram. The function **plot** notices the structure of its argument and responds by plotting points with no plotting symbols, connected by lines, to form a smooth curve. This is another good example of the fact that **plot** is a generic function. Try to construct a smoothed histogram of the **zvars** data.

A great number of other single-variable graphical descriptives are available. See the help files for **boxplot(),barplot(), pie(),** and **stem(),** for example.

11.2 Scatter Plots and Function Plots

For a simple scatter plot of two variables, specify the variables in a call to **plot()** (horizontal axis variable first). For example,

```
> plot(Illiteracy,Murder)
```

We can attempt to make the graph more attractive with some options:

```
> plot(Illiteracy,Murder,las=1,lwd=2,cex=1.2,pch=19,
      xlab="Percent Illiterate",ylab="")
> title("Murders per 100,000 vs. Percent Illiterate")
> text(2,4,"The 50 States",cex=1.2, adj=0, col=2)
```

The graphical parameters **las** and **lwd** controlling axis-label style and line widths have been mentioned already. The choice **pch=**19 calls for a solid-circle plotting symbol. The **cex** argument changes font size ("character expansion"): if less than 1, it shrinks the characters; if greater than 1, it expands them. The axis labels (or lack thereof) are specified by **xlab** and **ylab**.

The **title** and **text** commands add to the existing plot. There are a great number of other functions that add to plots: see the help files for **points(), lines(), symbols(), arrows(), segments(), mtext(), polygon(), legend(),** and **axis()**. For most of these, the first two arguments are x,y coordinates (possibly vectors of these) of the location(s) on the plot where text, points, etc. are to be added, using the current axis system set up on the active plot (note these values may at times differ from the labels shown on the axes). For example, in the **text** statement above, the values 2,4 specify that the provided text string is to be located at those coordinates; **adj=**0 means that the string will be left-justified at that spot (**adj=**0.5, the default, will center text, and **adj=**1 will right-justify text, at the supplied coordinates). The **col=**2 argument specifies a color number from the current color scheme. Alternately, you may specify a color with a character string (see help(colors)).

The locations of add-on text and symbols can also be determined interactively with the **locator()** function. For example, enter the following command:

```
> text(locator(1),"1977",cex=1.2, adj=0, col=2)
```

After entering this, click (with the left mouse button) on the plot at the point you would like the left-justified text string "1977" to begin.

A common task in the statistics business is to plot a smooth function f(x) versus some values x. Often this smooth curve is superimposed on a scatter plot of points. If the values in a vector xgrid have been sorted, and yhat is the corresponding vector of function values f(x), then

```
> plot(xgrid,yhat,type="l")
```

will create a basic lineplot for these values. If we are adding to an existing plot,

```
> lines(xgrid,yhat)
```

will do the job, as long as the existing plot's axes are matched up well to the values in xgrid and yhat. We can also add to an existing plot by using the **par(new=TRUE)** command; a new **plot()** statement will not create a new graphics window, but will add features to the current graphics window. The feature can be disabled by typing **par(new=FALSE)**. Some programmers prefer this approach to the use of **lines()** and **points()** commands since the **plot()** command is more versatile. Be careful though, since axis labels, etc may not line up exactly.

As a simple example, let's fit a quadratic function to the Murder vs. Illiteracy data by least squares, and superimpose the fitted curve on the scatter plot. Without getting bogged down in how to fit the regression…

```
> Illiteracy2 = Illiteracy^2
```

```
> lm(Murder ~ Illiteracy + Illiteracy2)
```

This fits the quadratic regression model and (since we have not assigned a name to the results) prints a summary on the screen. From this fitted model we can generate a grid of values for the x (Illiteracy) axis, `Illgrid`, and a vector of fitted values `Murderhat` evaluated on this grid (there are easier, but more confusing, ways to do this):

```
> Illgrid = pretty(Illiteracy,50)
> Murderhat = 1.6627+5.5642*Illgrid-0.4586*Illgrid^2
```

Finally, we can add the curve to the plot:

```
> lines(Illgrid,Murderhat,lty=2,lwd=2,col=2)
> text(0.5,14,"With Fitted Quadratic", adj=0, col=2)
```

Not really an excellent fit…maybe we should try a different model….

If you would like to generate a graph of a function, you can also use **`curve()`**. Note that **`curve()`** creates a new plotting window, so it cannot be overlaid on a scatterplot, though the **`points()`** command could be used to overlay points on a curve.

```
> curve(1.6627+5.5642*x-
0.4586*x^2,min(Illiteracy,max(Illiteracy),ylab="")
> points(Illiteracy,Murder)
```

Presentation-quality and publication-quality graphics often require a high degree of user control over the plotting package. This can be managed in R by putting your plot together piece by piece. You will probably want to write a function (see Chapter 12) to do this, since the function can be changed easily and re-called to reconstruct the graphic when your teacher (or your co-author, or the journal editor, or the Grad School office…etc) suggests changes.

The first step in a plot put together piece-by-piece is construction of an "empty plot", one which exists only to set up an axis system to locate add-ons:

```
> plot(xvec,yvec,
      type="n",xaxt="n",xlab="",yaxt="n",ylab="")
```

This statement seems to create an empty box (you can opt not to have the box, too) but sets up a coordinate system necessary to include the values contained in `xvec` and `yvec`. The default x-axis and its label are suppressed by the options **`xaxt="n", xlab=""`**; these can be added after-the-fact with (e.g.) an **`axis()`** statement. After an "empty plot" statement like the above, you can add points, curves, line segments, symbols, arrows, text strings, and so on to your heart's content.

This "empty plot" stepwise technique is how I prefer to make plots with multiple curves and/or scattered points. There is also a function **`matplot()`** that can superimpose multiple curves and/or scatter plots by plotting all columns of a matrix, but it seems less flexible than what is often needed.

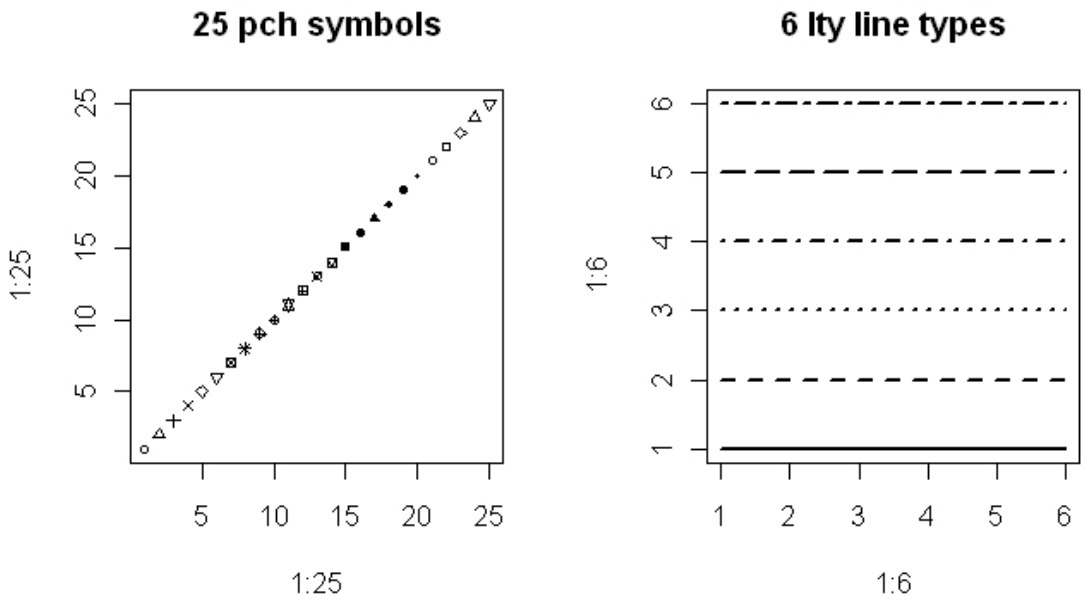
11.3 Multiple Plots on a Page

One can accomplish this with a very good degree of control using **par()** and a lot of care; I have examples. To do this very well is often a lot of work, though. The easy way to construct multiple plots an a single page is through the **mfrow** and/or **mfcol** parameters in **par()**. For example, suppose we desire two plots side-by-side on a page (i.e. in 1 row, with 2 columns). After opening a graphics window, set

```
> par(mfrow=c(1,2))
```

Then proceed with both of your plots. When R encounters the second plot statement, it assumes the first plot is finished; any additions after that point will be made to the second plot. If you would like 6 plots laid out in two rows of three plots each, use **par(mfrow=c(2,3))** or **par(mfcol=c(2,3))**. The difference between these two choices is the order in which plots are created (by row for **mfrow,** by column for **mfcol**). Below is a slightly complicated example that also demonstrates **for** loops and yields a useful souvenir. Type these commands very carefully, exactly as shown. The result is shown below the code. You may want to make a hard copy of this figure for future reference, if you have a printer handy.

```
> windows()
> par(mfrow=c(1,2),pty="s")
> plot(1:25,1:25,type="n")
> for (i in 1:25) points(i,i,pch=i,cex=0.7)
> title("25 pch symbols")
> plot(1:6,1:6,type="n")
> for (i in 1:6) lines(c(1,6),c(i,i),lty=i,lwd=2)
> title("6 lty line types")
```



Some R plotting functions naturally create multiple plots on a page. Scatterplot matrices are useful for exploratory data analysis—the **`pairs()`** command will generate all possible pairs of two-way scatterplots for the data frame or matrix listed as the first argument. The plots are automatically arranged in a *n* by *n* grid, where *n* is the number of variables in the data frame or the number of columns in the matrix. Refinements are available in the R packages ***car*** and ***lattice***.

11.4 Three-D plots

Three-D plots represent the values of a numeric matrix as a third dimension, with the row and column indices (or numeric vectors of the same lengths) representing the first two dimensions. There are at least four types in R: **`contour()`**,**`filled.contour()`**, **`persp()`** and **`image()`**. The most basic call is of the form (e.g. for **`contour`**)

```
> contour(zmatrix)
```

Let's try this for the built-in numeric matrix of approximate topographic information on the Maunga Whau volcano in Auckland, New Zealand:

```
> data(volcano)
> contour(volcano)
```

Try the **filled.contour()**, **persp()** and **image()** plots on this data, too. To modify colors used, see the help file for the **heat.colors()**, **topo.colors()**, **terrain.colors()**, and/or **palette()** functions.

Unless more meaningful values are provided, these functions use the row and column indices to determine the x- and y-values, scaling these to lie between 0 and 1. If meaningful x- and y-values are available (e.g. latitude and longitude values, or so-called "easting" and "northing" values, these can be provided in the call, something like this:

```
> contour(xvec, yvec, zmatrix)
```

All of these functions, of course, accept graphical parameters to "pretty them up"; all can be enhanced using the created x-y coordinate system and adding text, symbols, etc. The **contour** function can output a list of contour-curve coordinates which might be useful. Contours can also be added to an existing plot. The **lattice** library provides a more flexible set of options for 3-dimensional plots.

11.5 Interactive Graphics and Exploratory Data Analysis

Close all plots now by entering **graphics.off()** as many times as needed. Then reconstruct the basic plot of Murder vs. Illiteracy:

```
> plot(Illiteracy,Murder)
```

R offers some capability to interact with plots; for example, if we are curious about an unusual point on a plot, we may be able to identify it. Suppose the plot's points are given by vectors **xvec** and **yvec**, and there is another vector of labels for these points (of exactly the same length; all three vectors must be sorted in the same way – be careful!). A generic call to the **identify()** function, after the scatter plot has been constructed, looks like:

```
> weirdos = identify(xvec,yvec,labels)
```

After entering this command, using the leftmost mouse button, click on a point on the plot and its label should appear near the point. You may identify as many points as you like in this way. When finished, click the right mouse button and choose Stop. The created object, here called `weirdos`, is a vector containing the indices of the identified points, in the order identified. These indices could be used to extract these points from **xvec** and **yvec** or a data frame or matrix having the same number of rows as `labels`.

For example, suppose we are curious which state has the highest murder rate. Let's find out.

```
> mypoints = identify(Illiteracy,Murder,state.abb)
```

Now click under the point with the highest murder rate. Continue clicking as long as your curiosity drives you. When finished, click the right mouse button and choose Stop. Now investigate the created object `mypoints` :

```
> mypoints
```

```
> state.abb[mypoints]
```

The capability to interactively identify points on scatter plots is one of the basic tools of EDA (Exploratory Data Analysis), credited largely to John Tukey. Another EDA tool is the scatter plot matrix, whose generic call is something like this:

```
> pairs(num.matrix)
```

Try it on four of the **yvec** variables to explore relationships between education, murder rate, and life expectancy:

```
> pairs(state.x77[,3:6])
```

Crowded as it is, this plot allows us to quickly examine every pairwise relationship.

Another graphical technique related to the scatterplot matrix is the conditioning plot or "coplot". In a coplot, a variable y would be plotted against another variable x, separately for each value of a grouping variable. It has a somewhat unusual syntax. Here is a call for the Murder vs. Illiteracy data, conditioning on state.region:

```
> coplot( Murder ~ Illiteracy | state.region)
```

A data frame containing the variables must usually be attached, or can be referenced as part of the function call using the **data=** argument.

The original S package had even more built-in interactive capabilities, and Splus has retained these, but R (as part of the base package) unfortunately has not. These include "brushing" and "spinning". In brushing, a scatter plot matrix is first constructed, and then points on any given plot are identified using the mouse. As they are identified on any one plot, the same points are highlighted on every plot. In spinning, three variables are plotted on x-y-z axes, and using the mouse the user is capable of spinning the axes to give the illusion of a three-dimensional point cloud. What actually happens is that 2-dimensional projections of the three-tuples are computed quickly and plotted, then recomputed and replotted, and the viewer "fills in" the illusion of three-dimensional form in a fashion similar to what happens with 3-D glasses. These capabilities have been removed from the base package for R, but can be added as optional packages.

12. AN INTRODUCTION TO FUNCTION WRITING

R will be most useful to you when you become comfortable writing your own functions. As an example, the next page shows a text file which is a home-written function to perform a two-sample t-test and confidence interval for a difference of population means (this is the t-procedure that assumes independent samples and uses the pooled sample variance). We will get this into R in three steps:

1. Write the function commands carefully in a text file. Note that the function begins with the word **`function`** followed by parentheses containing argument names and, if desired, their default value assignments. For example, in this case, the default value for **`alpha`** is 0.05, and the default value for **`header`** (the plot title) is a blank. Immediately following the closing parenthesis after the argument list is a left brace ("curly bracket"). All actual function statements must lie between this brace and its closing right brace. The first few lines of the function should be documentation comments, statements starting with # signs, explaining briefly what the function does, what it accepts as its input arguments, and what kind of object, if any, it returns. When you are fairly sure that the function is correctly written, select the text and copy it to the clipboard.

2. Use the **`fix()`** function to create an "empty" function with the appropriate name (I try to use verbs in function names to help find them in the workspace…or an extension like "myfct"):

```
> fix(twosamp.myfct)
```

This opens a text editor window. At this point, it should contain only the words "**`function() {}`**"; delete these (without copying them to the clipboard) and paste your own function text there in their place. Then go to File, Save, and Exit.

3. (a) If you then return to the R prompt with no error messages, your function has, as far as R can tell, no obvious syntax errors and now resides in your workspace. Check it by issuing the **`ls()`** command or typing its name at the prompt (e.g.):

```
> twosamp.myfct
```

(b) If, however, when you quit your **`fix()`** session, you receive an error message similar to this one:

```
Error in edit(name, file, editor) : An error occurred…
(etc etc some sketchy details will be provided)
use a command like
 x = edit()
 to recover
```

This means that the R syntax-checker has found fault with what you wrote - for example, there may be left parentheses or braces without matching right parentheses or braces. The original function (in this case **`twosamp.myfct`**) has *not* been placed in the workspace yet. If you have a text file containing the function as it was pasted in, it may be best to go back to that text file, modify it as best you can in response to the error messages, save it and copy it to the clipboard, and then repeat step 2 above. Alternatively, check statements line by line (if possible) to help identify the problem.

If you do not have a text file version of the function as you modified it, be very careful at this step or you will lose the work you did during the most recent **`fix()`** session. To get back to the text file you were just working on, issue the command (e.g. if the function name is `twosamp.myfct`)

```
> twosamp.myfct = edit()
```

If you make more changes, you might want to save the results as a text file, using File ⇒ Save As, in case you can't find your syntax errors during this R session.

As an exercise, at this point you should create a text file `twosamp.myfct.txt` from the code shown below, and try to get it into R as a function. Try to understand what each command of the function is accomplishing.

```
function(yvec,trtvec,alpha=0.05,header="") {
################################################
# A function to compute a two-sample t-test and confidence
# interval (equal-variance, independent samples).  yvec is
# a numeric vector containing both samples' data.  trtvec
# is a vector, same length as yvec, of treatment
# identifiers for the data in yvec.  A boxplot comparing
# the treatments' data is constructed.  Output is a one-row
# data frame reporting the results of the test and
# confidence interval
################################################
trtvec=as.factor(trtvec)
boxplot(split(yvec,trtvec))
title(header)
ybar=tapply(yvec,trtvec,mean)
varvec=tapply(yvec,trtvec,var)
nvec=table(trtvec)
error.df=nvec[1]+nvec[2]-2
pooled.var=((nvec[1]-1)*varvec[1]+(nvec[2]-
1)*varvec[2])/error.df
diff12estimate=ybar[1]-ybar[2]
stderr=sqrt(pooled.var*((1/nvec[1])+(1/nvec[2])))
tratio=diff12estimate/stderr
twosidedP=2*(1-pt(abs(tratio),error.df))
tcrit=qt(1-alpha/2,error.df)
lower=diff12estimate-tcrit*stderr
upper=diff12estimate+tcrit*stderr
out=data.frame(diff12estimate,stderr,tratio,twosidedP,lower
,upper,alpha)
out
}
```

As another exercise, open your function as though you wish to modify it further:

```
> fix(twosamp.myfct)
```

Now, remove a parenthesis or a brace somewhere, Save, and Exit. This should cause you to be in case 3b above. Try to get back to your function using the **`edit()`** command as discussed earlier, and replace the missing brace / parenthesis.

Some general rules and comments about functions:

- If the function will return anything to the calling program, it can return one and only one object, though this object may be a list. This one object is simply named in the last statement of the function, before the closing "}" brace.
- The function statements can operate on data passed as arguments (using the function's argument names, not the names of the same objects as they exist outside the function), or on objects it creates, or on objects that already existed outside the function in the workspace or search list before the function was called. Any objects created by the function will not be saved when the function terminates, unless they are passed back in the final statement. However, changes to the search list seem to be preserved even after the function terminates (e.g. if the function creates and attaches a data frame, that data frame seems to still be present when the function terminates).

To test out our new function for doing two-sample t-tests, what better data set to use than the one used by W.S. Gosset ("Student") in his 1908 article on the t-distribution? Load the data:

```
> data(sleep)
```

By entering **`help(sleep)`** we find that this is a 20-row data frame containing two columns; the second column identifies a "soporific treatment" applied to a subject. This column **`group`** is a factor object with levels "1" and "2". The first column **`extra`** gives the extra sleep in hours over some baseline amount for that subject. Before calling **`twosamp.myfct`**, check the workspace contents. Then call it as follows:

```
> results = twosamp.myfct(sleep$extra, sleep$group)
```

We see the created boxplots of the two groups' data. We can view our numerical output by naming the created object:

```
> results
```

Of course, if we had not assigned the function output to an object name, these results would have been printed on the screen without being saved in the workspace.

Once your function is into R, you are not necessarily home free. There may be syntax errors which cannot be caught until run time, and then there are always those wonderful logical errors under which the function will run just fine but produce erroneous output. R has a terrific tool for debugging functions known as the **`browser`**. To use it, simply edit your dysfunctional function with, e.g.

```
> fix(my.fct)
```

and insert the command

```
browser()
```

at a strategic spot; perhaps the spot just before the trouble seems to occur, if you have any clue. Save and quit the function; then try to run it again. This time, the **`browser`** will

stop the function at exactly the spot you inserted it, and give you a **browser** prompt, something like this:

```
Browse[1]>
```

From this prompt, you can issue commands to examine all the objects that the function has created up to this point. If you don't remember what your own buggy function looks like, enter its name and it'll appear on screen (including the **browser** command). Usually a quick inspection of objects created just before this spot and/or their attributes or dimensions, etc., or a plot or two, will reveal the problem(s). Or, you can copy and paste statements from the function code following the **browser** command and see where the crash actually occurs. At any rate, when you are finished browsing around and want to continue executing the function, enter "c" at the browser prompt,

```
Browse[1]> c
```

and the function will resume execution at the next command. Or, if you prefer, enter Q,

```
Browse[1]> Q
```

and R will quit executing the function altogether (note the capital Q). Or, enter n,

```
Browse[1]> n
```

and this will bring up the next command after the browser() command inside the function for your inspection.

When you are finished investigating your buggy function, enter Q and re-edit the function with, e.g.

```
> fix(my.fct)
```

and make the changes. You may want to remove the **browser()** command now, or move it further down the function code if you expect more errors.

Once you get your function running, don't forget to make a text file copy. You are required to give me $1 every time you do this and forget to remove the **browser()** command.