R for Programmers

Norman Matloff
University of California, Davis

December 4, 2008

IMPORTANT NOTICE: This document is no longer being maintained. My book on R programming, *The Art of R Programming*, is due out in August 2011. You are welcome to use an early draft at `http://heather.cs.ucdavis.edu/~matloff/132/NSPpart.pdf`; it was about 50% complete and contains bugs, but should be useful.

Licensing:

Contents

Prerequisites

The only real prerequisite is that you have some programming experience; you need not be an expert programmer, though experts should find the material suitable for their level too.

Occasionally there will be some remarks aimed at professional programmers, say about object-oriented programming or Python, but these remarks will not make the treatment inaccessible to those having only a moderate background in programming.

1 What Is R?

R is a scripting language for statistical data manipulation and analysis. It was inspired by, and is mostly compatible with, the statistical language S developed by AT&T. The name S, obviously standing for statistics, was an allusion to another programming language developed at AT&T with a one-letter name, C. S later was sold to a small firm, which added a GUI interface and named the result S-Plus.

R has become more popular than S/S-Plus, both because it's free and because more people are contributing to it. R is sometimes called "GNU S."

2 Why Use R for Your Statistical Work?

Why use anything else? As the Cantonese say, *yauh peng, yauh leng*—"both inexpensive and beautiful."

Its virtues:

- a public-domain implementation of the widely-regarded S statistical language; R/S is the de facto standard among professional statisticians
- comparable, and often superior, in power to commercial products in most senses
- available for Windows, Macs, Linux
- in addition to enabling statistical operations, it's a general programming language, so that you can automate your analyses and create new functions
- object-oriented and functional programming structure
- your data sets are saved between sessions, so you don't have to reload each time
- open-software nature means it's easy to get help from the user community, and lots of new functions get contributed by users, many of which are prominent statisticians

I should warn you that one submits commands to R via text, rather than mouse clicks in a Graphical User Interface (GUI). If you can't live without GUIs, you should consider using one of the free GUIs that have been developed for R, e.g. R Commander or JGR. (See Section 34.2 below.) Note that R definitely does have graphics—tons of it. But the graphics are for the output, e.g. plots, not for the input.

Though the terms *object-oriented* and *functional programming* may pique the interests of computer scientists, they are actually quite relevant to anyone who uses R.

The term *object-oriented* can be explained by example, say statistical regression. When you perform a regression analysis with other statistical packages, say SAS or SPSS, you get a mountain of output. By contrast, if you call the **lm()** regression function in R, the function returns an object containing all the results—estimated coefficients, their standard errors, residuals, etc. You then pick and choose which parts of that object to extract, as you wish.

R is *polymorphic*, which means that the same function can be applied to different types of objects, with results tailored to the different object types. Such a function is called a *generic function*.[1] Consider for

[1] In C++, this is called a *virtual function*.

instance the **plot()** function. If you apply it to a simple list of numbers, you get a simple plot of them, but if you apply it to the output of a regression analysis, you get a set of plots of various aspects of the regression output. This is nice, since it means that you, as a user, have fewer commands to remember! For instance, you know that you can use the **plot()** function on just about any object produced by R.

The fact that R is a programming language rather than a collection of discrete commands means that you can combine several commands, each one using the output of the last, with the resulting combination being quite powerful and extremely flexible. (Linux users will recognize the similarity to shell pipe commands.)

For example, consider this (compound) command

```
nrow(subset(x03,z==1))
```

First the **subset()** function would take the data frame **x03**, and cull out all those records for which the variable **z** has the value 1. The resulting new frame would be fed into **nrow()**, the function that counts the number of rows in a frame. The net effect would be to report a count of **z** = 1 in the original frame.

A common theme in R programming is the avoidance of writing explicit loops. Instead, one exploits R's functional programming and other features, which do the loops internally. They are much more efficient, which can make a huge timing difference when running R on large data sets.

3 How to Run R

R has two modes, *interactive* and *batch*. The former is the typical one used.

3.1 Interactive Mode

You start R by typing "R" on the command line in Linux or on a Mac, or in a Windows Run window. You'll get a greeting, and then the R prompt, >.

You can then execute R commands, as you'll see in the quick sample session discussed in Section 4. Or, you may have your own R code which you want to execute, say in a file **z.r**. You could issue the command

```
> source("z.r")
```

which would execute the contents of that file. Note by the way that the contents of that file may well just be a function you've written. In that case, "executing" the file would mean simply that the R interpreter reads in the function and stores the function's definition in memory. You could then execute the function itself by calling it from the R command line, e.g.

```
> f(12)
```

3.2 Running R in Batch Mode

Sometimes it's preferable to automate the process of running R. For example, we may wish to run an R script that generates a graph output file, and not have to bother with manually running R. Here's how it could be done. Consider the file **z.r**, which produces a histogram and saves it to a PDF file:

```
pdf("xh.pdf")  # set graphical output file
hist(rnorm(100))  # generate 100 N(0,1) variates and plot their histogram
dev.off()  # close the file
```

Don't worry about the details; the information in the comments (marked with #) suffices here.

We could run it automatically by simply typing

```
R CMD BATCH --vanilla < z.r
```

The **–vanilla** option tells R not to load any startup file information, and not to save any.

4 A First R Example Session (5 Minutes)

We start R from our shell command line, and get the greeting message and the > prompt:

```
R : Copyright 2005, The R Foundation for Statistical Computing
Version 2.1.1  (2005-06-20), ISBN 3-900051-07-0
...
Type `q()' to quit R.

>
```

Now let's make a simple data set, a *vector* in R parlance, consisting of the numbers 1, 2 and 4, and name it **x**:

```
> x <- c(1,2,4)
```

The standard assignment operator in R is <-. However, there are also ->, = and even the **assign()** function.

The "c" stands for "concatenate." Here we are concatenating the numbers 1, 2 and 4. Or more precisely, we are concatenating three one-element vectors consisting of those numbers. This is because any object is considered a one-element vector.

Thus we can also do, for instance,

```
> q <- c(x,x,8)
```

which would set **q** to (1,2,4,1,2,4,8).

Since "seeing is believing," go ahead and confirm that the data is really in **x**; to print the vector to the screen, simply type its name. If you type any variable name, or more generally an expression, while in interactive mode, R will print out the value of that variable or expression. (Python programmers will find this feature familiar.) For example,

```
> x
[1] 1 2 4
```

Yep, sure enough, **x** consists of the numbers 1, 2 and 4.

The "[1]" here means in this row of output, the first item is item 1 of that output. If there were say, two rows of output with six items per row, the second row would be labeled [7]. Our output in this case consists of only one row, but this notation helps users read voluminous output consisting of many rows.

Again, in interactive mode, one can always print an object in R by simply typing its name, so let's print out the third element of **x**:

```
> x[3]
[1] 4
```

We might as well find the mean and standard deviation:

```
> mean(x)
[1] 2.333333
> sd(x)
[1] 1.527525
```

If we had wanted to save the mean in a variable instead of just printing it to the screen, we could do, say,

```
> y <- mean(x)
```

Again, since you are learning, let's confirm that **y** really does contain the mean of **x**:

```
> y
[1] 2.333333
```

As noted earlier, we use # to write comments.

```
> y  # print out y
[1] 2.333333
```

These of course are especially useful when writing programs, but they are useful for interactive use too, since R does record your commands (see Section 28). The comments then help you remember what you were doing when you later read that record.

As the last example in this quick introduction to R, let's work with one of R's internal datasets, which it uses for demos. You can get a list of these datasets by typing

```
> ?datasets
```

One of the datasets is **Nile**, containing data on the flow of the Nile River. Let's again find the mean and standard deviation,

```
> mean(Nile)
[1] 919.35
> sd(Nile)
[1] 169.2275
```

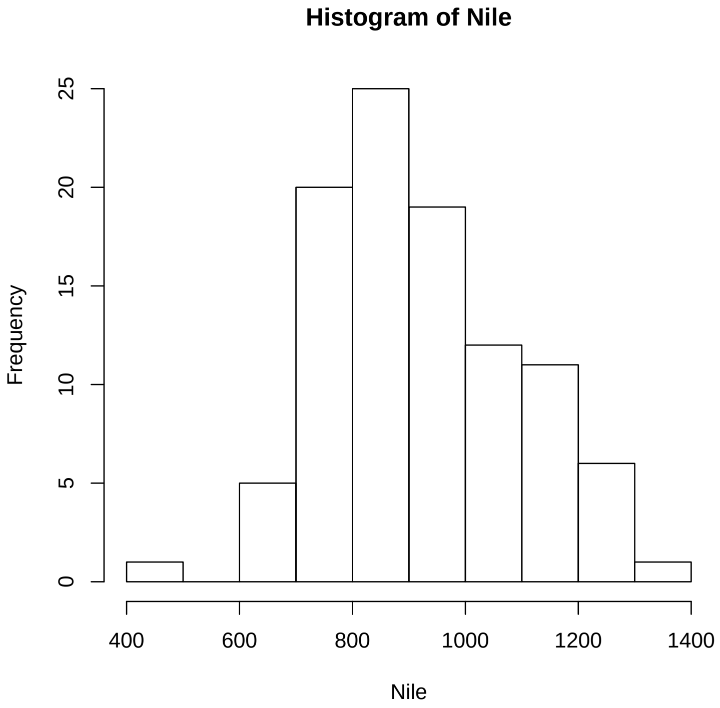

Figure 1: Nile data

and also plot a histogram of the data:

```
> hist(Nile)
```

A window pops up with the histogram in it, as seen in Figure 1. This one is bare-bones simple, but R has all kinds of bells and whistles you can use optionally. For instance, you can change the number of bins by specifying the breaks variable; hist(z,breaks=12) would draw a histogram of the data z with 12 bins. You can make nicer labels, etc. When you become more familiar with R, you'll be able to construct complex color graphics of striking beauty.

Well, that's the end of this first 5-minute introduction. We leave by calling the quit function (or optionally by hitting ctrl-d in Linux):

```
> q()
Save workspace image? [y/n/c]: n
```

That last question asked whether we want to save our variables, etc., so that we can resume work later on. If we answer y, then the next time we run R, all those objects will automatically be loaded. This is a very important feature, especially when working with large or numerous datasets; see more in Section 28.

5 Functions: a Short Programming Example

In the following example, we define a function **oddcount()** while in R's interactive mode, and then call the function on a couple of test cases. The function is supposed to count the number of odd numbers in its argument vector.

```
# comment:  counts the number of odd integers in x
> oddcount <- function(x)  {
+ k <- 0
+ for (n in x)  {
+    if (n %% 2 == 1) k <- k+1
+ }
+ return(k)
+ }
> oddcount(c(1,3,5))
[1] 3
> oddcount(c(1,2,3,7,9))
[1] 4
```

Here is what happened: We first told R that we would define a function **oddcount()** of one argument x. The left brace demarcates the start of the body of the function. We wrote one R statement per line. Since we were still in the body of the function, R reminded us of that by using + as its prompt[2] instead of the usual $>$. After we finally entered a right brace to end the function body, R resumed the $>$ prompt.

6 Scalars, Vectors, Arrays and Matrices

Remember, objects are actually considered one-element vectors. So, there is really no such thing as a scalar.

Vector elements must all have the same *mode*, which can be **integer**, **numeric** (floating-point number), **character** (string), **logical** (boolean), **complex**, **object**, etc.

Vectors indices begin at 1. Note that vectors are stored like arrays in C, i.e. contiguously, and thus one cannot insert or delete elements, *a la* Python. If you wish to do this, use a list instead.

A variable might not have a value, a situation designated as **NA**. This is like **None** in Python and **undefined** in Perl, though its origin is different. In statistical datasets, one often encounters missing data, i.e. observations for which the values are missing. In many of R's statistical functions, we can instruct the function to skip over any missing values.

Arrays and matrices are actually vectors too, as you'll see; they merely have extra attributes, e.g. in the matrix case the numbers of rows and columns. Keep in mind that since arrays and matrices are vectors, that means that everything we say about vectors applies to them too.

One can obtain the length of a vector by using the function of the same name, e.g.

```
> x <- c(1,2,4)
> length(x)
[1] 3
```

[2] Actually, this is a line continuation character.

6.1 “Declarations”

You must warn R ahead of time that you intend a variable to be one of the vector/array types. For instance, say we wish **y** to be a two-component vector with values 5 and 12. If you try

```
> y[1] <- 5
> y[2] <- 12
```

the first command (and the second) will be rejected, but

```
> y <- vector(length=2)
> y[1] <- 5
> y[2] <- 12
```

works, as does

```
> y <- c(5,12)
```

The latter is OK because the right-hand side is a vector type, so we are binding **y** to an already-existent vector.

6.2 Generating Useful Vectors with “:”, seq() and rep()

Note the : operator:

```
> 5:8
[1] 5 6 7 8
> 5:1
[1] 5 4 3 2 1
```

Beware of the operator precedence:

```
> i <- 2
> 1:i-1
[1] 0 1
> 1:(i-1)
[1] 1
```

The **seq()** (”sequence”) generates an arithmetic sequence, e.g.:

```
> seq(5,8)
[1] 5 6 7 8
> seq(12,30,3)
[1] 12 15 18 21 24 27 30
> seq(1.1,2,length=10)
 [1] 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Though it may seem innocuous, the **seq()** function provides foundation for many R operations. See examples in Sections 14 and 23.14.

The **rep()** (”repeat”) function allows us to conveniently put the same constant into long vectors. The call form is **rep(z,k)**, which creates a vector of **k*length(z)** elements, each equal to **z**. For example:

```
> x <- rep(8,4)
> x
[1] 8 8 8 8

> rep(1:3,2)
[1] 1 2 3 1 2 3
```

6.3 Vector Arithmetic and Logical Operations

You can add vectors, e.g.

```
> x <- c(1,2,4)
> x + c(5,0,-1)
[1] 6 2 3
```

You may surprised at what happens when we multiply them:

```
> x * c(5,0,-1)
[1]  5  0 -4
```

As you can see, the multiplication was elementwise. This is due to the functional programming nature of R.

The **any()** and **all()** functions are handy:

```
> x <- 1:10
> if (any(x > 8)) print("yes")
[1] "yes"
> if (any(x > 88)) print("yes")
> if (all(x > 88)) print("yes")
> if (all(x > 0)) print("yes")
[1] "yes"
```

6.4 Recycling

When applying an operation to two vectors which requires them to be the same length, the shorter one will be *recycled*, i.e. repeated, until it is long enough to match the longer one, e.g.

```
> c(1,2,4) + c(6,0,9,20,22)
[1]  7  2 13 21 24
Warning message:
longer object length
     is not a multiple of shorter object length in: c(1, 2, 4) + c(6,
     0, 9, 20, 22)
```

Here's a more subtle example:

```
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> x+c(1,2)
     [,1] [,2]
[1,]    2    6
[2,]    4    6
[3,]    4    8
```

What happened here is that **x**, as a 3x2 matrix, is also a six-element vector, which in R is stored column-by-column. We added a two-element vector to it, so our addend had to be repeated twice to make six elements. So, we were adding c(1,2,1,2,1,2) to **x**.

6.5 Vector Indexing

You can also do *indexing* of arrays, picking out elements with specific indices, e.g.

```
> y <- c(1.2,3.9,0.4,0.12)
> y[c(1,3)]
[1] 1.2 0.4
> y[2:3]
[1] 3.9 0.4
```

Note carefully that duplicates are definitely allowed, e.g.

```
> x <- c(4,2,17,5)
> y <- x[c(1,1,3)]
> y
[1]  4  4 17
```

Negative subscripts mean that we want to exclude the given elements in our output:

```
> z <- c(5,12,13)
> z[-1]  # exclude element 1
[1] 12 13
> z[-1:-2]
[1] 13
```

In such contexts, it is often useful to use the **length()** function:

```
> z <- c(5,12,13)
> z[1:length(z)-1]
[1]  5 12
```

Note that this is more general than using **z[1:2]**. In a program with general-length vectors, we could use this pattern to exclude the last element of a vector.

Here is a more involved example of this principle. Suppose we have a sequence of numbers for which we want to find successive differences, i.e. the difference between each number and its predecessor. Here's how we could do it:

```
> x <- c(12,15,8,11,24)
> y <- x[-1] - x[-length(x)]
> y
[1]  3 -7  3 13
```

Here we want to find the numbers 15-12 = 3, 8-15 = -7, etc. The expression **x[-1]** gave us the vector (15,8,11,24) and **x[-length(x)]** gave us (12,15,8,11). Subtracting these two vectors then gave us the differences we wanted.

Make careful note of the above example. This is the “R way of doing things.” By taking advantage of R’s vector operations, we came up with a solution which avoids loops. This is clean, compact and likely much faster when our vectors are long. We often use R’s functional programming features to these ends as well.

6.6 Vector Element Names

The elements of a vector can optionally be given names. For instance:

```
> x <- c(1,2,4)
> names(x)
NULL
> names(x) <- c("a","b","ab")
> names(x)
[1] "a"  "b"  "ab"
> x
 a  b ab
 1  2  4
```

We can remove the names from a vector by assigning NULL:

```
> names(x) <- NULL
> x
[1] 1 2 4
```

We can even reference elements of the vector by name, e.g.

```
> x <- c(1,2,4)
> names(x) <- c("a","b","ab")
> x["b"]
b
2
```

6.7 Matrices

A matrix is a vector with two additional attributes, the number of rows and number of columns.

6.7.1 General Operations

Multidimensional vectors in R are called *arrays*. A two-dimensional array is also called a *matrix*, and is eligible for the usual matrix mathematical operations.

Matrix row and column subscripts begin with 1, so for instance the upper-left corner of the matrix **a** is denoted **a[1,1]**. The internal linear storage of a matrix is in *column-major order*, meaning that first all of column 1 is stored, then all of column 2, etc.

One of the ways to create a matrix is via the **matrix()** function, e.g.

```
> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)
> y
```

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

Here we concatenated what we intended as the first column, the numbers 1 and 2, with what we intended as the second column, 3 and 4. That was our data in linear form, and then we specified the number of rows and columns. The fact that R uses column-major order then determined where these four numbers were put.

Though internal storage of a matrix is in column-major order, we can use the **byrow** argument in **matrix()** to TRUE in order to specify that the data we are using to fill a matrix be interpreted as being in row-major order. For example:

```
> m <- matrix(c(1,2,3,4,5,6),nrow=3)
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m <- matrix(c(1,2,3,4,5,6),nrow=2,byrow=T)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

('T' is an abbreviation for "TRUE".)

Since we specified the matrix entries in the above example, we would not have needed to specify **ncol**; just **nrow** would be enough. For instance:

```
> y <- matrix(c(1,2,3,4),nrow=2)
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

Note that when we then printed out **y**, R showed us its notation for rows and columns. For instance, [,2] means column 2, as can be seen in this check:

```
> y[,2]
[1] 3 4
```

Another way we could have built **y** would have been to specify elements individually:

```
> y <- matrix(nrow=2,ncol=2)
> y[1,1] = 1
> y[2,1] = 2
> y[1,2] = 3
> y[2,2] = 4
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

We can perform various operations on matrices, e.g. matrix multiplication, matrix scalar multiplication and matrix addition:

```
> y %*% y  # ordinary matrix multiplication
  [,1] [,2]
[1,] 7   15
[2,]10   22
> 3*y
  [,1] [,2]
[1,] 3    9
[2,] 6   12
> y+y
  [,1] [,2]
[1,] 2    6
[2,] 4    8
```

For linear algebra operations on matrices, see Section 6.12.

Again, keep in mind—and when possible, exploit—the notion of recycling (Section 6.4. For instance:

```
> x <- 1:2
> y <- c(1,3,4,10)
> x*y
[1]  1  6  4 20
```

Since **x** was shorter than **y**, it was recycled to the four-element vector **c(1,2,1,2)**, then multiplied elementwise with **y**.

6.7.2 Matrix Row and Column Names

For example:

```
> z <- matrix(c(1,2,3,4),nrow=2)
> z
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> colnames(z)  <- c("a","b")
> z
     a b
[1,] 1 3
[2,] 2 4
> colnames(z)
[1] "a" "b"
```

The function **rownames()** works similarly.

6.7.3 Matrix Indexing

The same operations we discussed in Section 6.5 apply to matrices. For instance:

```
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> z[,c(2,3)]
```

```
  [,1] [,2]
[1,] 1    1
[2,] 1    0
[3,] 0    1
[4,] 0    0
```

Here’s another example:

```
> y <- matrix(c(11,21,31,12,22,32),nrow=3,ncol=2)
> y
  [,1] [,2]
[1,]11   12
[2,]21   22
[3,]31   32
> y[2:3,]
  [,1] [,2]
[1,]21   22
[2,]31   32
> y[2:3,2]
[1] 22 32
```

You can copy a smaller matrix to a slice of a larger one:

```
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> y[2:3,] <- matrix(c(1,1,8,12),nrow=2)
> y
     [,1] [,2]
[1,]    1    4
[2,]    1    8
[3,]    1   12
```

```
> x <- matrix(nrow=3,ncol=3)
> x[2:3,2:3] <- cbind(4:5,2:3)
> x
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA    4    2
[3,]   NA    5    3
```

6.8 Sensing the Number of Rows and Columns of a Matrix

The numbers of rows and columns in a matrix can be obtained through the **nrow()** and **ncol()** functions, e.g.

```
> x
  [,1] [,2]
[1,] 1    4
[2,] 2    5
[3,] 3    6
> nrow(x)
[1] 3
```

These functions are useful when you are writing a general-purpose library function whose argument is a matrix. By being able to sense the number of rows and columns in your code, you alleviate the caller of the burden of supplying that information as two additional arguments.

6.9 Treating a Vector As a One-Column Matrix

In doing matrix work, you may be disconcerted to find that if you have code that deals with matrices of various sizes, a matrix degenerates to a vector if it has only one column. You can fix that by then applying **as.matrix()**. For example:

```
> u
[1] 1 2 3
> v <- as.matrix(u)
> attributes(u)
NULL
> attributes(v)
$dim
[1] 3 1
```

So, **as.matrix()** converted the 3-element vector **u** to a 3x1 matrix **v**.

6.10 Adding/Deleting Elements of Vectors and Matrices

Technically, vectors and matrices are of fixed length and dimensions. However, they can be reassigned, etc. Consider:

```
> x <- c(12,5,13,16,8)
> x <- c(x,20)  # append 20
> x
[1] 12  5 13 16  8 20
> x <- c(x[1:3],20,x[4:6])  # insert 20
> x
[1] 12  5 13 20 16  8 20  # delete elements 2 through 4
> x <- x[-2:-4]
> x
[1] 12 16  8 20
```

The **rbind()** and **cbind()** functions enable one to add rows or columns to a matrix.

For example:

```
> one
[1] 1 1 1 1
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> cbind(one,z)
[1,]1 1 1 1
[2,]1 2 1 0
[3,]1 3 0 1
[4,]1 4 0 0
```

You can also use these functions as a quick way to create small matrices:

```
> q <- cbind(c(1,2),c(3,4))
> q
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

We can delete rows or columns in the same manner as shown for vectors above, e.g.:

```
> m <- matrix(1:6,nrow=3)
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m <- m[c(1,3),]
> m
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

6.11 Matrix Row and Column Mean Functions

The function **mean()** applies only to vectors, not matrices. If one does call this function with a matrix argument, the mean of all of its elements is computed, not multiple means row-by-row or column-by-column, since a matrix is a vector.

The functions **rowMeans()** and **colMeans()** return vectors containing the means of the rows and columns. There are also corresponding functions **rowSums()** and **colSums()**.

6.12 Linear Algebra Operations on Vectors and Matrices

Multiplying a vector by a scalar works directly, as seen earlier. For example,

```
> y
[1]  1  3  4 10
> 2*y
[1]  2  6  8 20
```

If you wish to compute the inner product ("dot product") of two vectors, use **crossprod()**. Note that the name is a misnomer, as the function does not compute vector cross product.

For matrix multiplication in the mathematical sense, the operator to use is %*%, not *. Note also that a vector is considered a one-row matrix, not a one-column matrix, and thus is suitable as the left factor in a matrix product, but not directly usable as the right factor.

The function **solve()** will solve systems of linear equations, and even find matrix inverses. For example:

```
> a <- matrix(c(1,1,-1,1),nrow=2,ncol=2)
> b <- c(2,4)
> solve(a,b)
[1] 3 1
> solve(a)
  [,1] [,2]
[1,]  0.5  0.5
[2,] -0.5  0.5
```

Use **t()** for matrix transpose, **qr()** for QR decomposition, **chol()** for Cholesky, and **det()** for determinants.

Use **eigen()** to compute eigenvalues and eigenvectors, though if the matrix in question is a covariance matrix, the R function **prcomp()** may be preferable. The function **diag()** extracts the diagonal of a square matrix, useful for obtaining variances from a covariance matrix.

7 Lists

R's **list** structure is similar to a C **struct**. It plays an important role in R, with data frames, object oriented programming and so on, as we will see later.

7.1 Creation

As an example, consider an employee database. Suppose for each employee we store name, salary and a boolean indicating union membership. We could initialize our database to be empty if we wish:

```
j <- list()
```

Or we could create a list and enter our first employee, Joe, this way:

```
j <- list(name="Joe", salary=55000, union=T)
```

We could print **j** out:

```
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE
```

Actually, the element names, e.g. "salary," are optional. One could alternatively do this:

```
> jalt <- list("Joe", 55000, T)
> jalt
[[1]]
[1] "Joe"

[[2]]
[1] 55000

[[3]]
[1] TRUE
```

Here we refer to **jalt**'s elements as 1, 2 and 3 (which we can also do for **j** above).

7.2 List Tags and Values, and the unlist() Function

If the elements in a list do have names, e.g. with **name**, **salary** and **union** for **j** above, these names are called **tags**. The value associated with a tag is indeed called its **value**.

You can obtain the tags via **names()**:

```
> names(j)
[1] "name"   "salary" "union"
```

To obtain the values, use **unlist()**:

```
> ulj <- unlist(j)
> ulj
   name  salary   union
  "Joe" "55000"  "TRUE"
> class(ulj)
[1] "character"
```

The return value of **unlist()** is a vector, in this case a vector of mode character, i.e. a vector of character strings.

7.3 Issues of Mode Precedence

Let's look at this a bit more closely:

```
> x
$abc
[1] 2

$de
[1] 5
```

Here the list **x** has two elements, with **x$abc = 2** and **x$de = 5**. Just for practice, let's call **names()**:

```
> names(x)
[1] "abc" "de"
```

Now let's try **unlist()**:

```
> ulx <- unlist(x)
> ulx
abc  de
  2   5
> class(ulx)
[1] "numeric"
```

So again **unlist()** returned a vector, but R noticed that all the values were numeric, so it gave **ulx** that mode. By contrast, with **ulj** above, though one of the values was numeric, R was forced to take the "least common denominator," and make the vector of mode character.

This sounds like some kind of precedence structure, and it is. As R's help for **unlist()** states,

> Where possible the list elements are coerced to a common mode during the unlisting, and so the result often ends up as a character vector. Vectors will be coerced to the highest type of the components in the hierarchy NULL < raw < logical < integer < real < complex < character < list < expression: pairlists are treated as lists.

But there is something else to deal with here. Though **ulx** is a vector and not a list, R did give each of the elements a name. We can remove them by settings their names to NULL, seen in Section 6.6:

```
> names(ulx) <- NULL
> ulx
[1] 2 5
```

7.4 Accessing List Elements

The $ symbol is used to designate named elements of a list, but also [[ ]] works for referencing a single element and [ ] works for a group of them:

```
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

> j[[1]]
[1] "Joe"

> j[2:3]
$salary
[1] 55000

$union
[1] TRUE
```

Note that [[ ]] returns a value, while [ ] returns a sublist.

7.5 Adding/Deleting List Elements

One can dynamically add and delete elements:

```
> z <- list(a="abc",b=12)
> z
$a
[1] "abc"

$b
[1] 12

> z$c = 1
> z
$a
[1] "abc"

$b
[1] 12

$c
[1] 1
```

```
> z$1] <- NULL  # delete element 1
> z
$b
[1] 12

$c
[1] 1

[[3]]
[1] 1 2
> if (is.null(z$d)) print("it’s not there")  # testing existence
[1] "it’s not there"
> y[[2]] <- 8  # can "skip" elements
> y
[[1]]
NULL

[[2]]
[1] 8
```

7.6 Indexing of Lists

To do indexing of a list, use [ ] instead of [[ ]]:

```
 z[2:3]
$c
[1] 1

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

Names of list elements can be abbreviated to whatever extent is possible without causing ambiguity, e.g.

```
> j$sal
[1] 55000
```

One common use is to package return values for functions that return more than one piece of information. Say for instance the function **f()** returns a matrix **m** and a vector **v**. Then one could write

```
return(list(mat=m, vec=v))
```

at the end of the function, and then have the caller access these items like this:

```
l <- f()
m <- l$mat
v <- l$vec
```

This is typical form for functions in the R library.

7.7 Size of a List

You can obtain the number of elements in a list via **length()**:

```
> length(j)
[1] 3
```

7.8 Recursive Lists

Lists can be recursive, i.e. you can have lists within lists. For instance:

```
> b <- list(u = 5, v = 12)
> c <- list(w = 13)
> a <- list(b,c)
> a
[[1]]
[[1]]$u
[1] 5

[[1]]$v
[1] 12


[[2]]
[[2]]$w
[1] 13


> length(a)
[1] 2
```

So, **a** is now a two-element list, with each element itself being a list.

8 Data Frames

On an intuitive level, a *data frame* is like a matrix, with a rows-and-columns structure. However, it differs from a matrix in that each column may have a different mode. For instance, one column may be numbers and another column might be character strings.

On a technical level, a data frame is a list of vectors. See Section 8.2 for more on this.

8.1 A Second R Example Session

For example, consider an employee dataset. Each row of our data frame would correspond to one employee. The first column might be the employee's name, thus of **character** (i.e. string) mode, with the second column being salary, thus of **numeric** mode, and with the third column being a boolean (i.e. **logical** mode) indicating whether the employee is in a union.

As my sample data set, I have created a file named **exams**, consisting of grades for the three exams in a certain course (two midterm exams and a final exam). The first few lines in the file are

```
Exam1 Exam2 Exam3
62 70 60
74 34 64
50 35 40
...
```

Note that I have separated fields here by spaces.

As you can see, other than the first record, which contains the names of the columns (i.e. the variables), each line contains the three exam scores for one student. This is the classical “two-dimensional file” notion, i.e. each line in our file contains the data for one observation in a statistical dataset. The idea of a data frame is to encapsulate such data, along with variable names into one object.

As mentioned, I’ve specified the variable names in the first record. Our variable names didn’t have any embedded spaces in this case, but if they had, we’d need to quote any such name, e.g.

```
"Exam 1" "Exam 2" "Exam 3"
```

Suppose the second exam score for the third student had been missing. Then we would have typed

```
50 NA 40
```

in that line of the exams file. In any subsequent statistical analyses, R would do its best to cope with the missing data, in the obvious manners. (We may have to set the option **na.rm=T**.) If for instance we had wanted to find the mean score on Exam 2, R would find the mean among all students except the third.

We first read in the data from the file exams into a data frame which we’ll name **testscores**:

```
> testscores <- read.table("exams",header=TRUE)
```

The parameter **header=TRUE** tells R that we do have a header line (for the variable names), so R should not count that first line in the file as data.

In R, the components of an object are accessed via the $ operator. For example, the vector of all the Exam1 scores is **testscores$Exam1**, as we confirm here:

```
> testscores$Exam1
[1]  62  74  50  62  39  60  48  80  49  49 100  30  61 100  82  37  54 65  36
[20]  97  60  80  70  50  60  24  60  75  77  71  25  93  80  92  75 26  27  55
[39]  30  44  86  35  95  98  50  50  34 100  57  99  67  77  70  53 38
```

The [1] means that items 1-19 start here, the [20] means that items 20-38 start here, etc.

We will illustrate operations on this data in the following sections.

8.2 List Representation

You will have a better understanding of data frames if you keep in mind that technically, a data frame is implemented as a list of equal-length vectors. (Recall that R’s *list* construct was covered in Section 7.) Each column is one element of the list.

For instance, in Section 8.1, the data frame **testscores**' first column is referenced as **testscores$Exam1**, which is list notation.

Recall that elements of vectors can have names. The rows of a data frame can have names too. So, a data frame does have a bit more structure than a general list of vectors.

8.3 Matrix-Like Operations

Many matrix operations can also be used on data frames.

8.3.1 rowMeans() and colMeans()

```
> colMeans(testscores)
Exam1    Exam2    Exam3
62.14545 51.27273 50.05455
```

8.3.2 rbind() and cbind()

The **rbind()** and **cbind()** matrix functions introduced in Section 6.10 work here too.

We can also create new columns from old ones, e.g. we can add a variable which is the difference between Exams 1 and 2:

```
> testscores$Diff21 <- testscores$Exam2 - testscores$Exam1
```

8.3.3 Indexing

One can also refer to the rows and columns of a data frame using two-dimensional array notation, including indexing. For instance, in our example data frame **testscores** here:

- testscores[2,3] would refer to the third score for the second student
- testscores[2,] would refer to the set of all scores for the second student
- testscores[c(1,2,5),] would refer to the set of all scores for the first, second and fifth students
- testscores[10:13,] would refer to the set of all scores for the tenth through thirteenth students
- testscores[-2,] would refer to the set of all scores for all students except the second

8.4 Creating a New Data Frame from Scratch

We saw above how to create a data frame by reading from a data file. We can also create a data frame directly, using the function **data.frame()**.

For example,

```
> z <- data.frame(cbind(c(1,2),c(3,4)))
> z
  X1 X2
1  1  3
2  2  4
```

Note again the use of the **cbind()** function.

We can also coerce a matrix to a data frame, e.g.

```
> x <- matrix(c(1,2,3,4),nrow=2,ncol=2)
> x
  [,1] [,2]
[1,] 1    3
[2,] 2    4
> y <- data.frame(x)
> y
  X1 X2
1  1  3
2  2  4
```

As you can see, the column names will be X1, X2, ... However, you can change them, e.g.

```
> z
  X1 X2
1  1  3
2  2  4
> names(z) <- c("col 1","col 2")
> z
  col 1 col 2
1  1     3
2  2     4
```

8.5 Converting a List to a Data Frame

For printing, statistical calculations and so on, you may wish to convert a list to a data frame. Here's straightforward code to do it:

```
# converts a list lst to a data frame, which is the return value
wrtlst <- function(lst) {
   frm <- data.frame()
   rw <- 1
   for (key in names(lst)) {
      frm[rw,1] <- key
      frm[rw,2] <- lst[key]
      rw <- rw+1
   }
   return(frm)
}
```

But if our list has named tags and has only numeric values, the following code may run faster:

```
# converts a list lst that has only numeric values to a data frame,
# which is the return value of the function
lsttodf <- function(lst) {
   n <- length(lst)
   # create the data frame, using the default column names
   frm <- data.frame(V1=character(n),V2=numeric(n))
   frm[,1] <- names(lst)
   frm[,2] <- as.numeric(unlist(lst))
   return(frm)
}
```

8.6 The *Factor* Factor

If your table does have a variable, i.e. a column, in character mode, you probably should set **as.is=T** in your call to **read.table()**, so that this variable stays a vector rather than a factor. Otherwise even your numeric columns will become factors, which could cause problems as seen in our example below.

The same is true for creating a data frame via a call to **data.frame()**. Consider:

```
> d <- data.frame(cbind(c(0,5,12,13),c("xyz","ab","yabc",NA)))
> d
  X1   X2
1  0  xyz
2  5   ab
3 12 yabc
4 13 <NA>
> d[1,1] <- 3
Warning message:
In `[<-.factor`(`*tmp*`, iseq, value = 3) :
  invalid factor level, NAs generated
> d
    X1   X2
1 <NA>  xyz
2    5   ab
3   12 yabc
4   13 <NA>
```

Why didn't **d[1,1]** change? Well, since the second column was a character vector, **data.frame()** treated it as a factor, and thus "demoted" the first column to factor status too. Then when we tried to assign the value 3 to **d[1,1]**, R told us that 3 was not one of the official values (**levels** for this factor.

This can be avoided by setting **stringsAsFactors** to false:

```
> d <- data.frame(cbind(c(0,5,12,13),c("xyz","ab","yabc",NA)),stringsAsFactors=F)
> d
  X1   X2
1  0  xyz
2  5   ab
3 12 yabc
4 13 <NA>
> d[1,1] <- 3
> d
  X1   X2
1  3  xyz
2  5   ab
3 12 yabc
4 13 <NA>
```

See Section 15.3.

8.7 Extended Example: Data Preparation

In a study of engineers and programmers sponsored by U.S. employers for permanent residency, I considered the question, “How many of these workers are ‘the best and the brightest,’ i.e. people of extraordinary ability?”[3]

The government data is limited. The (admittedly imperfect) way to determine whether a worker is of extraordinary ability was to look at the ratio of actual salary to the government prevailing wage for that job and location. If that ratio is substantially higher than 1.0 (by law it cannot be less than 1.0), one can reasonably assume that this worker has a high level of talent.

I used R to prepare and analyze the data, and will present excerpts of my preparation code here. First, I read in the data file:

```
all2006 <- read.csv("2006.csv",header=T,as.is=T)
```

The function **read.csv()** is essentially identical to **read.table()**, except that the input data are in the Comma Separated Value format exported by spreadsheets, which is the way the dataset was prepared by the Department of Labor (DOL). We now have a data frame, **all2006**, consisting of all the data for the year 2006.

Some wages in the dataset were reported in hourly, rather than yearly, terms. This indicates possible part-time job status, which I wanted to exclude. A look at the documentation for the dataset showed that column 17 was the pay period, i.e. yearly, hourly, etc., while columns 15 and 19 contained the actual salary and prevailing wage. So, I did some filtering:

```
all2006 <- all2006[all2006[,17]=="Year",]
all2006 <- all2006[all2006[,15]> 20000,]
all2006 <- all2006[all2006[,19]> 200,]
```

I also needed to create a new column for the ratio between actual wage and prevailing wage:

```
all2006 <- cbind(all2006,all2006[,15]/all2006[,19])
```

This new variable is in column 25. Since I knew I would be calculating the median in this column for many subsets of the data, I defined a function to do this work:

```
medrat <- function(dataframe) {
    return(median(dataframe[,25],na.rm=T))
}
```

Note the need to exclude NA values, which are common in government datasets.

In addition, I wanted to analyze the talent patterns at particular companies, using the company name stored in column 7:

```
makecorp <- function(corpname) {
    t <- all2006[all2006[,7]==corpname,]
```

[3] N. Matloff, *New Insights from the Dept. of Labor PERM Labor Certification Database*, Sloan Foundation West Coast Program on Science and Engineering Workers conference, January 18, 2008.

```
    return(t)
}
goog2006 <- makecorp("GOOGLE INC.")
medrat(goog2006)
ms2006 <- makecorp("MICROSOFT CORPORATION")
medrat(ms2006)
...
```

I also wanted to analyze by occupation. DOL has a code number for each job title, stored in column 14, which I used to create the corresponding subsets of the original dataset:

```
makeocc <- function(df,lowerbd,upperbd) {
    return(df[df[,14]>=lowerbd & df[,14]<=upperbd,])
}
# for 2007 data
prg2007 <- makeocc(all2007,"15-1021.00","15-1052.00") # programmers
se2007 <- makeocc(all2007,"15-1030.00","15-1039.00") # s.w. engineers
engr2007 <- makeocc(all2007,"17-2000.00","17-2999.00") # other engineers
...
```

One more example: I wanted to analyze by nationality, which is in column 24. Note my use of looping over a vector of mode character:

```
mainnatnames <- c("CHINA", "INDIA", "CANADA", "GERMANY", "UNITED KINGDOM")
natwithindf <- function(natlist,df) {
   for (nat in natlist) {
      tmpdf <- df[df[,24]==nat,]
      mr <- medrat(tmpdf)
      cat(nat,": ",mr,"\n")
   }
}
natwithindf(mainnatnames,prg2007)
natwithindf(mainnatnames,se2007)
...
```

9 Factors and Tables

Consider the data frame, say in a file **ct.dat**,

```
"VoteX" "VoteLastTime"
"Yes" "Yes"
"Yes" "No"
"No" "No"
"Not Sure" "Yes"
"No" "No"
```

where in the usual statistical fashion each row represents one subject under study. In this case, say we have asked five people (a) ”Do you plan to vote for Candidate X?” and (b) ”Did you vote in the last election?” (The first line in the file is a header.)

Let’s read in the file:

```
> ct <- read.table("ct.dat",header=T)
> ct
```

```
     VoteX VoteLastTime
1      Yes          Yes
2      Yes           No
3       No           No
4 Not Sure          Yes
5       No           No
```

We can use the **table()** function to convert this data to contingency table format, i.e. a display of the counts of the various combinations of the two variables:

```
> cttab <- table(ct)
> cttab
          VoteLastTime
VoteX      No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1
```

The 2 in the upper-left corner of the table shows that we had, for example, two people who said No to (a) and No to (b). The 1 in the middle-right indicates that one person answered Not Sure to (a) and Yes to (b).

We can in turn change this to a data frame—not the original one, but a data-frame version of the contingency table:

```
> ctdf <- as.data.frame(cttab)
> ctdf
     VoteX VoteLastTime Freq
1       No           No    2
2 Not Sure           No    0
3      Yes           No    1
4       No          Yes    0
5 Not Sure          Yes    1
6      Yes          Yes    1
```

This is useful, for instance, in the log-linear model (Section 36.2.2).

Note that in our original data frame, the two columns are called *factors* in R. A factor is basically a vector of mode **character**, intended to represent values of a categorical variable, such as the **ct$VoteX** variable above. The **factor** class includes a component **levels**, which in the case of **ct$VoteX** are Yes, No and Not Sure.

Of course, all of the above would still work if our original data frame **ct** we had three factors, or more.

We get counts on a single factor in isolation as well, e.g.

```
> y <- factor(c("a","b","a","a","b"))
> z <- table(y)
> z
y
a b
3 2
> as.vector(z)
[1] 3 2
```

Note the use here of **as.vector()** to extract only the counts.

Among other things, this gives us an easy way to determine what proportion of items satisfy a certain condition. For example:

```
> x <- c (5,12,13,3,4,5)
> table(x == 5)

FALSE  TRUE 
    4     2 
```

So our answer is 2/6.

The function **table()** is often used with **cut()**. To explain what the latter does, first consider the call

```
y <- cut(x,b,labels=F)
```

where **x** is a vector of observations, and **b** defines bins, which are the semi-open intervals **(b[1],b[2]], (b[2],b[3]],...**. Then **y[j]** will be the index **i** such that **x[j]** falls into bin **i**.

For instance,

```
> cut(1:8,c(0,4,7,8),labels=F)
[1] 1 1 1 1 2 2 2 3
```

The function **cut()** has many, many other options but in our context here, the point is that we can pipe the output of **cut()** into **talbe()**, thus getting counts of the numbers of observations in each bin.

```
bincounts <- function(x,b) {
   y <- cut(x,b)
   return(as.vector(table(y)))
}
```

10 Missing Values

Data sets in the real world tend to have problems. Observed values can be incorrect, and are often missing altogether. R designates the latter case by NA.

One can test for this condition by using **is.na()**. When applied to a vector, e.g.

```
> z <- c(5,NA,12)
> is.na(z)
[1] FALSE  TRUE FALSE
```

this function can be used to identify all the missing values, which then can be avoided in one's analysis. Thus for instance the **mean()** function has such an option:

```
> mean(z)
[1] NA
> mean(z,na.rm=T)
[1] 8.5
```

11 Functional Programming Features

Functional programming has several benefits:

- Clearer, more compact code.
- Potentially much faster execution speed.
- Less debugging (since you write less code).
- Easier transition to parallel programming.

11.1 Elementwise Operations on Vectors

Suppose we have a function **f()** that we wish to apply to all elements of a vector **x**. In many cases, we can accomplish this by simply calling **f()** on **x** itself.

11.1.1 Vectorized Functions

As we saw in Section 6.3, many operations are **vectorized**, such as + and $>$:

```
> u <- c(5,2,8)
> v <- c(1,3,9)
> u+v
[1]  6  5 17
> u > v
[1]  TRUE FALSE FALSE
```

The key point is that if an R function uses vectorized operations,[4] it too is vectorized, i.e. it can be applied to vectors in an elementwise fashion. For instance:

```
> w <- function(x) return(x+1)
> w(u)
[1] 6 3 9
```

Here **w()** uses +, which is vectorized, so **w()** is vectorized as well.

This applies to many of R's built-in functions. For instance, let's apply the function for rounding to the nearest integer to an example vector **y**:

```
> y <- c(1.2,3.9,0.4)
> z <- round(y)
> z
[1] 1 4 0
```

The point is that the **round()** function was applied individually to each element in the vector **y**. In fact, in

[4] If not, use **as.matrix()** to convert your vector to a matrix, and use **apply()**. An example is given in Section 11.4.3.

```
> round(1.2)
[1] 1
```

the operation still works, because the number 1.2 is actually considered to be a vector that happens to consist of a single element 1.2.

Here we used the built-in function **round()**, but you can do the same thing with functions that you write yourself.

Note that the functions can also have extra arguments, e.g.

```
> f <- function(elt,s) return(elt+s)
> y <- c(1,2,4)
> f(y,1)
[1] 2 3 5
```

As seen above, even operators such as + are really functions. For example, the reason why elementwise addition of 4 works here,

```
> y <- c(12,5,13)
> y+4
[1] 16  9 17
```

is that the + is actually considered a function! Look at it here:

```
> '+'(y,4)
[1] 16  9 17
```

11.1.2 The Case of Vector-Valued Functions

The above operations work with vector-valued functions too:

```
z12 <- function(z) return(c(z,zˆ2))
x <- 1:8
> z12(x)
 [1]  1  2  3  4  5  6  7  8  1  4  9 16 25 36 49 64
> matrix(z12(x),ncol=2)
     [,1] [,2]
[1,]    1    1
[2,]    2    4
[3,]    3    9
[4,]    4   16
[5,]    5   25
[6,]    6   36
[7,]    7   49
[8,]    8   64
```

11.2 Filtering

Another idea borrowed from functional programming is filtering, which is one of the most common operations in R.

11.2.1 On Vectors

For example:

```
> z <- c(5,2,-3,8)
> w <- z[z*z > 8]
> w
[1] 5  -3  8
```

Here is what happened above: We asked R to find the indices of all the elements of **z** whose squares were greater than 8, then use those indices in an indexing operation on **z**, then finally assign the result to **w**.

Look at it done piece-by-piece:

```
> z <- c(5,2,-3,8)
> z
[1]  5  2 -3  8
> z*z > 8
[1]  TRUE FALSE  TRUE  TRUE
```

Evaluation of the expression **z*z** $>$ **8** gave us a vector of booleans! Let's go further:

```
> z[c(TRUE,FALSE,TRUE,TRUE)]
[1]  5 -3  8
```

This example will place things into even sharper focus:

```
> z <- c(5,2,-3,8)
> j <- z*z > 8
> j
[1]  TRUE FALSE  TRUE  TRUE
> y <- c(1,2,30,5)
> y[j]
[1]  1 30  5
```

We may just want to find the positions within **z** at which the condition occurs. We can do this using **which()**:

```
> which(z*z > 8)
[1] 1 3 4
```

Here's an extension of an example in Section 6.5:

```
# x is an array of numbers, mostly in nondecreasing order, but with some
# violations of that order nviol() returns the number of indices i for
# which x[i+1] < x[i]

nviol <- function(x) {
   diff  <- x[-1]-x[1:(length(x)-1)]
   return(length(which(diff < 0)))
}
```

I noted in Section 2 that using the **nrow()** function in conjunction with filtering provides a way to obtain a count of records satisfying various conditions. If you just want the count and don't want to create a new table, you should use this approach.

You can also use this to selectively change elements of a vector, e.g.

```
> x <- c(1,3,8,2)
> x[x > 3] <- 0
> x
[1] 1 3 0 2
```

11.2.2 On Matrices and Data Frames

Filtering can be done with matrices and data frames too. Note that one must be careful with the syntax. For instance:

```
> x
     x
[1,] 1 2
[2,] 2 3
[3,] 3 4
> x[x[,2] >= 3,]
     x
[1,] 2 3
[2,] 3 4
```

Again, let's dissect this:

```
> j <- x[,2] >= 3
> j
[1] FALSE  TRUE  TRUE
> x[j,]
     x
[1,] 2 3
[2,] 3 4
```

Here is another example:

```
> m <- matrix(c(1,2,3,4,5,6),nrow=3)
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m[m[,1] > 1,]
     [,1] [,2]
[1,]    2    5
[2,]    3    6
> m[m[,1] > 1 & m[,2] > 5,]
[1] 3 6
```

11.3 Combining Elementwise Operations and Filtering, with the ifelse() Function

The form is

```
ifelse(b,u,v)
```

where b is a boolean vector, and u and v are vectors.

The return value is a vector, element i of which is **u[i]** if **b[i]** is true, or **v[i]** if **b[i]** is false. This is pretty abstract, so let's go right to an example:

```
> x <- 1:10
> y <- ifelse(x %% 2 == 0,5,12)
> y
 [1] 12  5 12  5 12  5 12  5 12  5
```

Here we wish to produce a vector in which there is a 5 wherever **x** is even, with a 12 wherever **x** is odd. So, the first argument is c(F,T,F,T,F). The second argument, 1, is treated as c(1,1,1,1,1) by recycling, and similarly for the third argument.

Here is another example, in which we have explicit vectors.

```
> x <- c(5,2,9,12)
> ifelse(x > 6,2*x,3*x)
[1] 15  6 18 24
```

The advantage of **ifelse()** over the standard if-then-else is that it is vectorized. Thus it's potentially much faster.

Due to the vector nature of the arguments, one can nest **ifelse()** operations. In the following example, involving an abalone data set, gender is coded as 'M', 'F' or 'I', the last meaning infant. We wish to recode those characters as 1, 2 or 3:

```
> g <- c("M","F","F","I","M")
> ifelse(g == "M",1,ifelse(g == "F",2,3))
[1] 1 2 2 3 1
```

The inner call to **ifelse()**, which of course is evaluated first, produces a vector of 2s and 3s, with the 2s corresponding to female cases, and 3s being for males and infants. The outer call results in 1s for the males, in which cases the 3s are ignored.

Remember, the vectors involved could be columns in matrices, and this is a very common scenario. Say our abalone data is stored in the matrix **ab**, with gender in the first column. Then if we wish to recode as above, we could do it this way:

```
> ab[,1] <- ifelse(ab[,1] == "M",1,ifelse(ab[,1] == "F",2,3))
```

11.4 Applying the Same Function to All Elements of a Matrix, Data Frame and Even More

This is not just for compactness of code, but for speed. If speed is an issue, such as when working with large data sets or long-running simulations, one must avoid explicit loops as much as possible, because R can do them a lot faster than you can.

To this end, there is the **apply()** function and its variants.

11.4.1 Applying the Same Function to All Rows or Columns of a Matrix

The arguments of **apply()** are the matrix/data frame to be applied to, the dimension—1 if the function applies to rows, 2 for columns—and the function to be applied.

For example, here we apply the built-in R function **mean()** to each column of a matrix **z**.

```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> apply(z,2,mean)
[1] 2 5
```

Here is an example of working on rows, using our own function:

```
> f <- function(x) x/c(2,8)
> y <- apply(z,1,f)
> y
  [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

You might be surprised that the size of the result here is 2 x 3 rather than 3 x 2. If the function to be applied returns a vector of k components, the result of **apply()** will have k rows. You can use the matrix transpose function **t()** to change it.

As you can see, the function to be applied needs at least one argument, which will play the role of one row or column in the array. In some cases, you will need additional arguments, which you can place following the function name in your call to **apply()**.

For instance, suppose we have a matrix of 1s and 0s, and want to create a vector as follows: For each row of the matrix, the corresponding element of the vector will be either 1 or 0, depending on whether the majority of the first **c** elements in that row are 1 or 0. Here **c** will be a parameter which we may wish to vary. We could do this:

```
> copymaj <- function(rw,c) {
+    maj <- sum(rw[1:c]) / c
+    return(ifelse(maj > 0.5,1,0))
+ }
> x <- matrix(c(1,1,1,0, 0,1,0,1, 1,1,0,1, 1,1,1,1, 0,0,1,0),nrow=4)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    1    1    0
[2,]    1    1    1    1    0
[3,]    1    0    0    1    1
[4,]    0    1    1    1    0
> apply(x,1,copymaj,3)
[1] 1 1 0 1
> apply(x,1,copymaj,2)
[1] 0 1 0 0
```

Here the values 3 and 2 form the actual arguments for the formal argument **c** in **copymaj()**.

So, the general form of apply is

```
apply(m,dimcode,f,fargs}
```

where **m** is the matrix, **dimcode** is 1 or 2, according to whether we will operate on rows or columns, **f** is the function to be applied, and **fargs** is an optional list of arguments to be supplied to **f**.

Note carefully that in writing **f()** *itself, its first argument must be a vector that will be supplied by the caller as a row or column of* **m**.

As R moves closer and closer to parallel processing, functions like **apply()** will become more and more important. For example, the **clusterApply()** function in the snow package gives R some parallel processing capability, by distributing the submatrix data to various network nodes, with each one basically running **apply()** on its submatrix, and then collect the results. See Section 32.2.

11.4.2 Applying the Same Function to All Elements of a List

The analog of **apply()** for lists is **lapply()**. It applies the given function to all elements of the specified list. For example:

```
> lapply(list(1:3,25:27),median)
[[1]]
[1] 2

[[2]]
[1] 26
```

In this example the list was created only as a temporary measure, so we should convert back to numeric:

```
> as.numeric(lapply(list(1:3,25:27),median))
[1]  2 26
```

11.4.3 Applying the Same Function to a Vector

If your function is vectorized, then you can simply call it on the vector, as in Section 11.1.

Otherwise, you can still avoid writing a loop, e.g. writing

```
lv <- length(v)
outvec <- vector(length=lv)
for (i in 1:lv) {
   outvec[i] <- f(v[i])
}
```

as follows:

```
outvec <- apply(as.matrix(v),1,f)
```

The call to **as.matrix()** will return a matrix whose sole column is **v**.

This may not save you much time if you are running R on just one machine, but if you are using, for instance, the snow package (see Section 32.2), with **parApply()** instead of **apply()**, it could be well worth doing.

11.5 Functions Are First-Class Objects

Functions can be used as arguments, assigned, etc. For instance,

```
> f1 <- function(a,b) return(a+b)
> f2 <- function(a,b) return(a-b)
> f <- f1
> f(3,2)
[1] 5
> f <- f2
> f(3,2)
[1] 1
> g <- function(h,a,b) h(a,b)
> g(f1,3,2)
[1] 5
> g(f2,3,2)
[1] 1
```

Since you can view any object when in R's interactive mode by typing the name of the object, and since functions are objects, you can view the code for a function (either one you wrote, or one in R) in this manner. For example,

```
> f1
function(a,b) return(a+b)
```

This is handy if you're using a function that you've written but have forgotten what its arguments are, for instance. It's also useful if you are not quite sure what an R library function does; by looking at the code you may understand it better.

Also, a nice feature is that you can edit functions from within R. In the case above, I could change **f1()** by typing

```
> f1 <- edit(f1)
```

This would open the default editor on the code for **f1**, which I could then edit and save back to **f1**.

The editor invokved will depend on R's internal options variable **editor**. In Unix-class systems, R will set this from your EDITOR or VISUAL environment variable, or you can set it yourself, e.g.

```
> options(editor="/usr/bin/vim")
```

See the online documentation if you have any problems.

Note that in this example I am saving the revision back to the same function. Note too that when I do so, I am making an assignment, and thus the R interpreter will compile the code; if I have any errors, the assignment will not be done. I can recover by the command

```
> x <- edit()
```

Warning: Apparently comments are not preserved.

12 R Programming Structures

R is a full programming language, similar to scripting languages such as Perl and Python. One can define functions, use constructs such as loops and conditionals, etc.

12.1 Use of Braces for Block Definition

The body of a **for**, **if** or similar statement does not need braces if it consists of a single statement.

12.2 Loops

12.3 Basic Structure

In our function **oddcount()** in Section 5, the line

```
+ for (n in x)  {
```

will be instantly recognized by Python programmers. It of course means that there will be one iteration of the loop for each component of the vector **x**, with **n** taking on the values of those components. In other words, in the first iteration, **n = x[1]**, in the second iteration **n = x[2]**, etc.

Looping with **while** and **repeat** are also available, complete with **break**, e.g.

```
> i <- 1
> while(1) {
+    i <- i+4
+    if (i > 10) break
+ }
> i
[1] 13
```

(Of course, **break** can be used with **for** too.)

12.3.1 Looping Over Nonvector Sets

The **for** construct works on any vector, regardless of mode. One can loop over a vector of file names, for instance. Say we have files **x** and **y** with contents

```
1
2
3
4
5
6
```

and

```
5
12
13
```

Then this loop prints each of them:

```
> for (fn in c("x","y")) print(scan(fn))
Read 6 items
[1] 1 2 3 4 5 6
Read 3 items
[1]  5 12 13
```

R does not directly support iteration over nonvector sets, but there are indirect yet easy ways to accomplish it. One way would be to use **lapply()**, as shown in Section 11.4.2. Another would be to use **get()**, e.g.:

```
> u
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    4
> v
     [,1] [,2]
[1,]    8   15
[2,]   12   10
[3,]   20    2
> for (m in c("u","v")) {
+    z <- get(m)
+    print(lm(z[,2] ~ z[,1]))
+ }

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept)       z[, 1]
    -0.6667       1.5000

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept)       z[, 1]
     23.286       -1.071
```

The reader is welcome to make his/her own refinements here.

12.4 Return Values

By the way, you often don't need the **return()** call. The last value computed will be returned by default. In the **oddcount()** example in Section 5, instead of writing

```
return(k)
```

we could simply write

```
k
```

This is true for nonscalars too (recall that a scalar is really a one-element vector anyway), e.g.:

```
> r <- function(x,y) {
+    c(x+y,x-y)
+ }
> r(3,2)
[1] 5 1
```

12.5 If-Else

The syntax for if-else is like this:

```
> if (r == 4) {
+    x <- 1
+    y <- 2
+ } else {
+    x <- 3
+    y <- 4
+ }
```

Note that the braces are necessary, even for single-statement bodies for the if and else, and the newlines are important too. For instance, the left brace before the **else** is what the parser uses to tell that this is an **if-else** rather than just an **if**; this would be easy for the parser to handle in batch mode but not in interactive mode. However, if you place the **else** on the same line as the **if**, this problem will not occur.

See also the **ifelse()** function discussed in Section 11.

12.6 Local and Global Variables

Say a variable **z** appearing within a function has the same name as a global. Then it will be treated as local, except that its initial value will be that of the global. Subsequent assignment to it within the function will ordinarily (see exception in Section 12.8) not change the value of the global. For example:

```
> u
[1] 1
> v
[1] 2
> f
function(x) {
   y <- u
   y <- y + 3
   u <- x
return(x+y+u+v)
}
> f(5)
[1] 16
> u
[1] 1
> v
[1] 2
> y
Error: object "y" not found
```

12.7 Function Arguments Don't Change

Yet another influence of the functional programming philosophy is that functions do not change their arguments, i.e. there are no *side effects* (unless the result is re-assigned to the argument). Consider, for instance, this:

```
> x <- c(4,1,3)
> y <- sort(x)
> y
```

```
[1] 1 3 4
> x
[1] 4 1 3
```

The point is that **x** didn't change.

Again, if you want the value of an argument to change, the best way to accomplish this is to reassign the return value of the function to the argument.

12.8 Writing to Globals Using the Superassignment Operator

If you do want to write to global variables (or more precisely, to variables one level higher than the current scope), you can use the **superassignment** operator, $>> -$. For example,

```
> two <- function(u) {
+    u <<- 2*u
+    y <<- 2*y
+    z <- 2*z
+ }
> x <- 1
> y <- 2
> z <- 3
> two(x)
> x
[1] 1
> y
[1] 4
> z
[1] 3
```

12.9 Arithmetic and Boolean Operators and Values

```
x + y            addition
x - y            subtraction
x * y            multiplication
x / y            division
x ^ y            exponentiation
x %% y           modular arithmetic
x %/% y          integer division
x == y           test for equality
x <= y           test for less-than-or-equal
x >= y           test for greater-than-or-equal
x && y            boolean and for scalars
x || y            boolean or for scalars
x & y            boolean and for vectors (vector x,y,result)
x | y            boolean or for vectors (vector x,y,result)
!x               boolean negation
```

The boolean values are TRUE and FALSE. They can be abbreviated to T and F, but must be capitalized. These values change to 1 and 0 in arithmetic expressions, e.g.

```
> 1 < 2
[1] TRUE
> (1 < 2) * (3 < 4)
[1] 1
> (1 < 2) * (3 < 4) * (5 < 1)
[1] 0
> (1 < 2) == TRUE
[1] TRUE
> (1 < 2) == 1
[1] TRUE
```

There are set operations, e.g.

```
> x <- c(1,2,5)
> y <- c(5,1,8,9)
> union(x,y)
[1] 1 2 5 8 9
> intersect(x,y)
[1] 1 5
> setdiff(x,y)
[1] 2
> setdiff(y,x)
[1] 8 9
> setequal(x,y)
[1] FALSE
> setequal(x,c(1,2,5))
[1] TRUE
> 2 %in% x  # note that plain "in" doesn't work
[1] TRUE
> 2 %in% y
[1] FALSE
```

You can invent your own operators! Just write a function whose name begins and ends with %. Here is an operator for the symmetric difference between two sets (i.e. all the elements in exactly one of the two operand sets):

```
> "%sdf%" <- function(a,b) {
+    sdfxy <- setdiff(x,y)
+    sdfyx <- setdiff(y,x)
+    return(union(sdfxy,sdfyx))
+ }
> x %sdf% y
[1] 2 8 9
```

12.10 Named Arguments

In Section 8.1, we read in a dataset from a file **exams**:

```
> testscores <- read.table("exams",header=TRUE)
```

The parameter **header=TRUE** told R that we did have a header line, so R should not count that first line in the file as data.

This is an example of the use of *named arguments*. The function **read.table()** has a number of arguments, some of which are optional, which means that we must specify which arguments we are using, by using their names, e.g. **header=TRUE** above. (Again, Python programmers will find this familiar.) The ones you don't specify all have default values.

13 Writing Fast R Code

A central theme in R programming is avoidance of explicit loops. instead relying on R's rich functionality to do the work for you. Not only does this save you programming and debugging time, it produces faster code, since R's functions have been written for efficiency. **This can be of the utmost importance in applications with large data sets or large amounts of computation.**

For instance, let's rewrite our function **oddcount()** in Section 5:

```
> oddcount <- function(x) return(length(which(x%%2==1)))
```

Let's test it:

```
> oddcount(c(1,5,6,2,19))
[1] 3
```

There is no explicit loop in this version of our **oddcount()**. We used R's vector filtering to avoid a loop, and even though R internally will loop through the array, it will do so much faster than we would with an explicit loop in our R code. In essence, the looping is done in C in native machine code, rather than interpretively in R code.

Of course, R's functional programming features, described in Section 11, provide many ways to help us avoid explicit loops.

For more detailed information on improving R performance, see `http://www.statistik.uni-dortmund.de/useR-2008/tutorials/useR2008introhighperfR.pdf`.

14 Simulation Programming in R

14.1 The Basics

Here is an example, finding P(Z < 1) for a N(0,1) random variable Z:

```
> count <- 0
> for (i in seq(1,100000))
+    if (rnorm(1) < 1.0) count <- count + 1
> count/100000
[1] 0.832
> count/100000.0
[1] 0.832
```

14.2 Achieving Better Speed

But as noted in Section 13, you should try to use R's built-in features for greater speed. The above code would be better written

```
> x <- rnorm(100000)
> length(x[x < 1.0])/100000.0
```

We achieve an increase in speed at the expense of using more memory, by keeping our random numbers in an array instead of generating and discarding them one at a time. Suppose for example we wish to simulate sampling from an adult human population in which height is normally distributed with mean 69 and standard deviation 2.5 for men, with corresponding values 64 and 2 for women. We'll create a matrix for the data, with column 1 showing gender (1 for male, 0 for female) and column 2 showing height. The straightforward way to do this would be something like

```
sim1 <- function(n)  {
   xm <- matrix(nrow=n,ncol=2)
   for (i in 1:n)  {
      d <- rnorm(1)
      if (runif(1) < 0.5) {
         xm[i,1] <- 1
         xm[i,2] <- 2.5*d + 69
      } else {
         xm[i,1] <- 0
         xm[i,2] <- 2*d + 64
      }
   }
   return(xm)
}
```

We could avoid a loop this way:

```
sim2 <- function(n)  {
   d <- matrix(nrow=n,ncol=2)
   d[,1] <- runif(n)
   d[,2] <- rnorm(n)
   smpl <- function(rw) {  # rw = one row of d
      if (rw[1] < 0.5) {
         y <- 1
         x <- 2.5*rw[2] + 69
      } else {
         y <- 0
         x <- 2*rw[2] + 64
      }
      return(c(y,x))
   }
   z <- apply(d,1,smpl)
   return(t(z))
}
```

Here is a quick illustration of the fact that we do gain in performance:

```
> system.time(sim1(1000))
[1] 0.028 0.000 0.027 0.000 0.000
> system.time(sim2(1000))
[1] 0.016 0.000 0.018 0.000 0.000
```

Here is a slightly more complicated example, using a classical problem from elementary probability courses. Urn 1 contains 10 blue marbles and eight blue ones. In Urn 2 the mixture is six blue and six yellow. We draw a marble at random from Urn 1 and transfer it to Urn 2, and then draw a marble at random from Urn 2. What is the probability that that second marble is blue? This quantity is easy to find analytically, but we'll use simulation. Here is the straightforward way:

```
sim3 <- function(nreps)  {
nb1 = 10  # 10 blue marbles in Urn 1
n1 <- 18  # number of marbles in Urn 1 at 1st pick
n2 <- 13  # number of marbles in Urn 2 at 2nd pick
count <- 0
for (i in 1:nreps)  {
   nb2 = 6  # 6 blue marbles orig. in Urn 2
   # pick from Urn 1 and put in Urn 2
   if (runif(1) < nb1/n1) nb2 <- nb2 + 1
   # pick from Urn 2
   if (runif(1) < nb2/n2) count <- count + 1
}
return(count/nreps)  # est. P(pick blue from Urn 2)
}
```

But here is how we can do it without loops:

```
sim4 <- function(nreps)  {
nb1 = 10  # 10 blue marbles in Urn 1
nb2 = 6  # 6 blue marbles orig. in Urn 2
n1 <- 18  # number of marbles in Urn 1 at 1st pick
n2 <- 13  # number of marbles in Urn 2 at 2nd pick
u <- matrix(c(runif(2*nreps)),nrow=nreps,ncol=2)
simfun <- function(rw,nb1,n1,nb2,ny2,n2) {
   if (rw[1] < nb1/n1) nb2 <- nb2 + 1
   if (rw[2] < nb2/n2) b <- 1 else b <- 0
   return(b)
}
z <- apply(u,1,simfun,nb1,n1,nb2,ny2,n2)
return(mean(z))  # est. P(pick blue from Urn 2)
}
```

Here we have set up a matrix **u** with two columns of U(0,1) random variates. The first column is used for our simulation of drawing from Urn 1, and the second for the drawing from Urn 2. Our function **simfun()** works on one repetition of the experiment. We have set up the call to **apply()** to go through all of the **nreps** repetitions.

Actually, on my machine, this second approach was actually slower. So, one must not assume that using **apply()** will necessarily speed things up. Note, though, that in a parallel version of R (Section 32, we'd likely get quite a speedup.

14.3 Built-In Random Variate Generators

R has functions to generate variates from a number of different distributions. For example, **rbinom()** generates binomial or Bernoulli random variates.[5] If we wanted to, say, find the probability of getting at least 4 heads out of 5 tosses of a coin, we could do this:

```
> x <- rbinom(100000,5,0.5)
> length(x[x >= 4])/100000
[1] 0.18791
```

There are also **rnorm()** for the normal distribution, **rexp()** for the exponential, **runif()** for the uniform, **rgamma()** for the gamma, **rpois()** for the Poisson and so on.

[5] A sequence of independent 0-1 valued random variables with the same probability of 1 for each is called **Bernoulli**.

14.3.1 Obtaining the Same Random Stream in Repeated Runs

By default, R will generate a different random number stream from run to run of a program. If you want the same stream each time, call **set.seed()**, e.g.

```
> set.seed(8888)  # or your favorite number as an argument
```

15 Input/Output

15.1 Reading from the Keyboard

You can use **scan()**:

```
> z <- scan()
1: 12 5
3: 2
4:
Read 3 items
> z
[1] 12  5  2
```

Use **readline()** to input a line from the keyboard as a string:

```
> w <- readline()
abc de f
> w
[1] "abc de f"
```

15.2 Printing to the Screen

In interactive mode, one can print the value of a variable or expression by simply typing the variable name or expression. In batch mode, one can use the **print()** function, e.g.

```
print(x)
```

The argument may be an object.

It's a little better to use **cat()** instead of **print()**, as the latter can print only one expression and its output is numbered, which may be a nuisance to us. E.g.

```
> print("abc")
[1] "abc"
> cat("abc\n")
abc
```

The arguments to **cat()** will be printed out with intervening spaces, for instance

```
> x <- 12
> cat(x,"abc","de\n")
12 abc de
```

If you don't want the spaces, use separate calls to **cat()**:

```
> z <- function(a,b) {
+    cat(a)
+    cat(b,"\n")
+ }
> z("abc","de")
abcde
```

15.3 Reading a Matrix or Data Frame From a File

The function **read.table()** was discussed in Section 8.1. Here is a bit more on it.

- The default value of **header** is FALSE, so if we don't have a header, we need not say so.
- By default, character strings are treated as R **factors** (Section 9). To turn this "feature" off, include the argument **as.is=T** in your call to **read.table()**.
- If you have a spreadsheet export file, i.e. of type **.csv** in which the fields are separated by commas instead of spaces, use **read.csv()** instead of **read.table()**. There is also **read.xls** to read core spreadsheet files.
- Note that if you read in a matrix via **read.table()**, the resulting object will be a data frame, even if all the entries are numeric. You may need it as a matrix, in which case do a followup call to **as.matrix()**.

There appears to be no good way of reading in a matrix from a file. One can use **read.table()** and then convert. A simpler way is to use **scan()** to read in the matrix row by row, making sure to use the **byrow** option in the function **matrix()**. For instance, say the matrix **x** is

```
1 0 1
1 1 1
1 1 0
1 1 0
0 0 1
```

We can read it into a matrix this way:

```
> x <- matrix(scan("x"),nrow=5,byrow=T)
```

15.4 Reading a File One Line at a Time

You can use **readLines()** for this. We need to create a *connection* first, by calling **file()** .

For example, suppose we have a file **z**, with contents

```
1 3
1 4
2 6
```

Then we can do this:

```
> c <- file("z","r")
> readLines(c,n=1)
[1] "1 3"
> readLines(c,n=1)
[1] "1 4"
> readLines(c,n=1)
[1] "2 6"
```

If **readLines()** encounters the end of the file, it returns a null string.

15.5 Writing to a File

15.5.1 Writing a Table to a File

The function **write.table()** works very much like **read.table()**, in this case writing a data frame instead of reading one.

In the case of writing a matrix, to a file, just state that you want no row or column names, e.g.

```
> write.table(xc,"xcnew",row.names=F,col.names=F)
```

15.5.2 Writing to a Text File Using cat()

(The point of the word *text* in the title of this section is that, for instance, the number 12 will be written as the ASCII characters '1' and '2', as with **printf**() with %d format in C—as opposed to the bits 000...00001100.)

The function **cat()** can be used to write to a file, one part at a time. For example:

```
> cat("abc\n",file="u")
> cat("de\n",file="u",append=T)
```

The file is saved after each operation, so at this point the file on disk really does look like

```
abc
de
```

One can write multiple fields. For instance

```
> cat(file="v",1,2,"xyz\n")
```

would produce a file **v** consisting of a single line,

```
1 2 xyz
```

15.5.3 Writing a List to a File

You have various options here. One would be to convert the list to a data frame, as in Section 8.5, and then call **write.table()**.

15.5.4 Writing to a File One Line at a Time

Use **writeLines**(). See Section 15.4.

15.6 Directories, Access Permissions, Etc.

R has a variety of functions for dealing with directories, file access permissions and the like.

Here is a short example. Say our current working directory contains files **x** and **y**, as well as a subdirectory **z**. Suppose the contents of **x** is

```
12
5
13
```

and **y** contains

```
3
4
5
```

The following code sums up all the numbers in the non-directory files here:

```
tot <- 0
ndatafiles <- 0
for (d in dir()) {
   if (!file.info(d)$isdir) {
      ndatafiles <- ndatafiles + 1
      x <- scan(d,quiet=T)
      tot <- tot + sum(x)
   }
}
cat("there were",ndatafiles,"nondirectory files, with a sum of",tot,"\n")
```

Type

```
> ?files
```

to get more information on permissions etc.

The functions **getwd()** and **setwd()** can be used to determine or change the current working directory.

The ".." notation for parent directory in Linux systems does work on them.

15.7 Accessing Files on Remote Machines Via URLs

Functions such as **read.table()**, **scan()** and so on accept file names as arguments. For example, I placed a file **z** on my Web page, with the contents

```
1 2
3 4
```

and accessed it from within R running on my home machine:

```
> z <- read.table("http://heather.cs.ucdavis.edu/˜matloff/z")
> z
  V1 V2
  1  1  2
  2  3  4
```

You can also read the file one line at a time, as in Section 15.4.

16 Object Oriented Programming

16.1 Managing Your Objects

16.1.1 Listing Your Objects with the ls() Function

The **ls()** command will list all of your current objects.

A useful named argument is **pattern**, which enables wild cards. For example:

```
> ls()
 [1] "acc"       "acc05"     "binomci"   "cmeans"    "divorg"    "dv"
 [7] "fit"       "g"         "genxc"     "genxnt"    "j"         "lo"
[13] "out1"      "out1.100"  "out1.25"   "out1.50"   "out1.75"   "out2"
[19] "out2.100"  "out2.25"   "out2.50"   "out2.75"   "par.set"   "prpdf"
[25] "ratbootci" "simonn"    "vecprod"   "x"         "zout"
"zout.100"
[31] "zout.125"  "zout3"     "zout5"     "zout.50"   "zout.75"
> ls(pattern="ut")
 [1] "out1"     "out1.100" "out1.25"  "out1.50"  "out1.75"  "out2"
 [7] "out2.100" "out2.25"  "out2.50"  "out2.75"  "zout"     "zout.100"
[13] "zout.125" "zout3"    "zout5"    "zout.50"  "zout.75"
```

16.1.2 Removing Specified Objects with the rm() Function

To remove objects you no longer need, use **rm()**. For instance,

```
> rm(a,b,x,y,z,uuu)
```

would remove the objects **a**, **b** and so on.

One of the named arguments of **rm()** is **list**, which makes it easier to remove multiple objects. For example,

```
> rm(list = ls())
```

would assign all of your objects to list, thus removing everything. If you make use of **ls()**'s **pattern** argument this tool becomes even more powerful.

16.1.3 Saving a Collection of Objects with the save() Function

Calling **save()** on a collection of objects will write them to disk for later retrieval by **load()**.

16.1.4 Listing the Characteristics of an Object with the names(), attributes() and class() Functions

An object consists of a gathering of various kinds of information, with each kind being called an *attribute*. The **names()** function will tell us the names of the attributes of the given object. For a data frame, for example, these will be the names of the columns. For a regression object, these will be **coefficients**, **residuals** and so on. Calling the **attributes()** function will give you all this, plus the class of the object itself. To just get the class, call **class()**.

16.1.5 The exists() Function

The function **exists()** returns TRUE or FALSE, depending on whether the argument exists. Be sure to quote the argument, e.g.

```
> exists("acc")
[1] TRUE
```

shows that the object **acc** exists.

16.1.6 Accessing an Object Via Strings

The call **get("u")** will return the object **u**. An example appears on page 48.

16.2 Generic Functions

As mentioned in my introduction, R is rather polymorphic, in the sense that the same function can have different operation for different classes. One can apply **plot()**, for example, to many types of objects, getting an appropriate plot for each. The same is true for **print()** and **summary**().

In this manner, we get a uniform interface to different classes. So, when someone develops a new R class for others to use, we can try to apply, say, **summary()** and reasonably expect it to work. This of course means that the person who wrote the class, knowing the R idiom, would have had the foresight of writing such a function in the class, knowing that people would expect one.

The functions above are known as *generic functions*. The actual function executed will be determined by the class of the object on which you are calling the function.

For example, let's look at a simple regression analysis (details in Section 36.1):

```
> x <- c(1,2,3)
> y <- c(1,3,8)
> lmout <- lm(y ~ x)
> lmout

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
       -3.0          3.5
```

Note that we printed out the object **lmout**. (Remember, by simply typing the name of an object in interactive mode, the object is printed.) What happened then was that the R interpreter saw that **lmout** was an object of class **lm** (the quotation marks are part of the class name), and thus instead of calling **print()**, it called **print.lm()**, a special print method in the **lm** class.

In fact, we can take a look at that method:

```
> print.lm
function (x, digits = max(3, getOption("digits") - 3), ...)
{
    cat("\nCall:\n", deparse(x$call), "\n\n", sep = "")
    if (length(coef(x))) {
        cat("Coefficients:\n")
        print.default(format(coef(x), digits = digits), print.gap = 2,
            quote = FALSE)
    }
    else cat("No coefficients\n")
    cat("\n")
    invisible(x)
}
<environment: namespace:stats>
```

Don't worry about the details here; our main point is that the printing was dependent on context, with a different print function being called for each different class.

You can see all the implementations of a given generic method by calling **methods()**, e.g.

```
> methods(print)
  [1] print.acf*                        print.anova
  [3] print.aov*                        print.aovlist*
  [5] print.ar*                         print.Arima*
  [7] print.arima0*                     print.AsIs
  [9] print.Bibtex*                     print.by
...
```

You can see all the generic methods this way:

```
> methods(class="default")
...
```

16.3 Writing Classes

A class is named via a quoted string:

```
> class(3)
[1] "numeric"
> class(list(3,TRUE))
[1] "list"
> lmout <- lm(y ~ x)
> class(lmout)
[1] "lm"
```

If a class is derived from a parent class, the named of the derived class will be a vector consisting of two strings, first one for the derived class and then one for the parent.

Methods are implemented as generic functions. The name of a method is formed by concatening the function name with a period and the class name, e.g. **print.lm()**.

The class of an object is stored in its **"class"** attribute (the quotation marks are necessary).

16.3.1 Old-Style Classes

Older versions of R used a cobbled-together structure for classes, referred to as S3. Under this approach, a class instance is created by forming a list, with the elements of the list being the member variables of the class. (Readers who know Perl may recognize this *ad hoc* nature in Perl's own OOP system.) The **"class"** attribute is set by hand by using the **attr()** or **class()** function, and then various generic functions are defined.

For instance, continuing our employee example from Section 7, we could write

```
> j <- list(name="Joe", salary=55000, union=T)
> class(j) <- "employee"
> attributes(j)  # let's check
$names
[1] "name"   "salary" "union"

$class
[1] "employee"
```

Now write a generic function:

```
print.employee <- function(wrkr) {
   cat(wrkr$name,"\n")
   cat("salary",wrkr$salary,"\n")
   cat("union member",wrkr$union,"\n")
}
```

Now test it:

```
> j
Joe
salary 55000
union member TRUE
```

Compare this to the call to the default **print()** back in Section 7:

```
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

> j[[1]]
[1] "Joe"
```

Here is a more involved example. Here we will write an R class **"ut"** for upper-triangular matrices. Recall that this means that these are square matrices whose elements below the diagonal are 0s. For example:

$$\begin{pmatrix} 1 & 5 & 12 \\ 0 & 6 & 9 \\ 0 & 0 & 2 \end{pmatrix}$$

The component **mat** of this class will store the matrix. There is no point in storing the 0s, so only the diagonal and above-diagonal elements will be stored, in column-major order. We could initialize storage for the above matrix, for instance, via the call c(1,5,6,12,9,2). The component **ix** of this class shows where in **mat** the various columns begin. For the above case, **ix** would be c(1,2,4), meaning that column 1 begins at **mat[1]**, column 2 begins at **mat[2]** and column 3 begins at **mat[4]**

The function below creates an instance of this class. Its argument **inmat** is in full matrix format, i.e. including the 0s.

```
ut <- function(inmat) {
   nr <- nrow(inmat)
   rtrn <- list()
   class(rtrn) <- "ut"
   rtrn$mat <- vector(length=sum1toi(nr))
   rtrn$ix <- sum1toi(0:(nr-1)) + 1
   for (i in 1:nr) {
      ixi <- rtrn$ix[i]
      # copy the i-th column of inmat to mat
      rtrn$mat[ixi:(ixi+i-1)] <- inmat[1:i,i]
   }
   return(rtrn)
}

# returns 1+...+i
sum1toi <- function(i) return(i*(i+1)/2)
```

Much of the R library still uses the S3 approach, but in the next section we will describe the newer system, S4.

16.3.2 New-Style Classes

Here one creates the class by calling **setClass()**. Continuing our employee example, we could write

```
> setClass("employee",
+    representation(
+       name="character",
+       salary="numeric",
+       union="logical")
+ )
[1] "employee"
```

Now, let's create an instance of this class, for Joe, using **new()**:

```
> joe <- new("employee",name="Joe",salary=55000,union=T)
> joe
An object of class employee
```

```
Slot "name":
[1] "Joe"

Slot "salary":
[1] 55000

Slot "union":
[1] TRUE
```

Note that the member variables are called *slots*. We reference them via the @ symbol, e.g.

```
> joe@salary
[1] 55000
```

The **slot()** function can also be used.

To define a generic function on a class, use **setMethod()**. Let's do that for our class **"employee"** here. We'll implement the **show()** function. To see what this function does, consider our command above,

```
> joe
```

As we know, in R, when we type the name of a variable while in interactive mode, the value of the variable is printed out:

```
> joe
An object of class employee
Slot "name":
[1] "Joe"

Slot "salary":
[1] 55000

Slot "union":
[1] TRUE
```

The action here is that **show()** is called.[6] In fact, we would get the same output here by typing

```
> show(joe)
```

Let's override that, with the following code (which is in a separate file, and brought in using **source**()):

```
setMethod("show", "employee",
   function(object) {
      inorout <- ifelse(object@union,"is","is not")
      cat(object@name,"has a salary of",object@salary,
         "and",inorout, "in the union", "\n")
   }
)
```

The first argument gives the name of the generic function which we will override, with the second argument giving the class name. We then define the new function.

Let's try it out:

```
> joe
Joe has a salary of 55000 and is in the union
```

[6] The function **show()** has precedence over **print**().

17 Type Conversions

The **str()** function converts an object to string form, e.g.

```
> x <- c(1,2,4)
> class(x)
[1] "numeric"
> str(x)
 num [1:3] 1 2 4
```

There is a generic function **as()** which does conversions, e.g.

```
> x <- c(1,2,4)
> y <- as.character(x)
> y
[1] "1" "2" "4"
> as.numeric(y)
[1] 1 2 4
> q <- as.list(x)
> q
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 4

> r <- as.numeric(q)
> r
[1] 1 2 4
```

You can see all of this family by typing

```
> methods(as)
```

The **unclass()** function converts a class object to an ordinary list.

18 Stopping Execution

You can write your code to check for various anomalies specific to your application, and if one is found, call **stop()** to stop execution.

19 Functions for Statistical Distributions

R has functions available for various aspects of most of the famous statistical distributions. Prefix the name by d for the density, p for the cdf, q for quantiles and r for simulation. The suffix of the name indicates the distribution, such as *norm*, *unif*, *chisq*, *binom*, *exp*, etc.

For example for the chi-square distribution:

```
> mean(rchisq(1000,different=2))  find mean of 1000 chi-square(2) variates
[1] 1.938179
> qchisq(0.95,1)  find 95th percentile of chi-square(2)
[1] 3.841459
```

An example of the use of **rnorm()**, to generate random normally-distributed variates, is in Section 14, as well as one for **rbinon()** for binomial/Bernoulli random variates. The function **dnorm()** gives the normal density, **pnorm()** gives the normal CDF, and **qnorm()** gives the normal quantiles.

The d-series, for density, gives the probability mass function in the case of discrete distributions. The first argument is a vector indicating at which points we wish to find the values of the pmf. For instance, here is how we would find the probabilities of 0, 1 or 2 heads in 3 tosses of a coin:

```
> dbinom(0:2,3,0.5)
[1] 0.125 0.375 0.375
```

See the online help pages for details, e.g. by typing

```
> help(pnorm)
```

20 Math Functions

The usual **exp()**, **log()**, **log10()**, **sqrt()**, **abs()** etc. are available, as well as **min()**, **which.min()** (returns the index for the smallest element), **max()**, **which.max()**, **pmin()**, **pmax()**, **sum()**, **prod()** (for products of multiple factors), **round()**, **floor()**, **ceiling()**, **sort()** etc. The function **factorial()** computes its namesake, so that for instance **factorial(3)** is 6.

Note that the function **min()** returns a scalar even when applied to a vector. By contrast, if **pmin()** is applied to vectors, it returns a vector of the elementwise minima. For example:

```
> z
     [,1] [,2]
[1,]    1    2
[2,]    5    3
[3,]    6    2
> min(z[,1],z[,2])
[1] 1
> pmin(z[,1],z[,2])
[1] 1 3 2
```

Also, some special math functions, described when you invoke **help()** with the argument Arithmetic.

The function **combn** generates combinations:

```
> c32 <- combn(1:,3,2)
> c32
     [,1] [,2] [,3]
[1,]    1    1    2
[2,]    2    3    3
> class(c32)
[1] "matrix"
```

The function also allows the user to specify a function to be called by **combn()** on each combination.

Function minimization/maximization can be done via **nlm()** and **optim()**.

R also has some calculus capabilities, e.g.

```
> D(expression(exp(x^2)),"x")  # derivative
exp(x^2) * (2 * x)
> integrate(function(x) x^2,0,1)
0.3333333 with absolute error < 3.7e-15
```

There are R packages such as **odesolve** for differential equations, **ryacas** to interface R with the Yacas symbolic math system, and so on.

21 String Manipulation

R has a number of string manipulation utilities, such as

- **grep()**: Searches for a substring, like the Linux command of the same name.
- **nchar()**: Finds the length of a string.
- **paste()**: Assembles a string from parts.
- **sprintf()**: Assembles a string from parts.
- **substr()**: Extracts a substring.
- **strsplit()**: Splits a string into substrings.

21.1 Example of nchar(), substr(): Testing for a Specified Suffix in a File Name

```
# tests whether the file name fn has the suffix suff,
# e.g. "abc" in "x.abc"
testsuffix <- function(fn,suff) {
   ncf <- nchar(fn)  # nchar() gives the string length
   dotpos <- ncf - nchar(suff) + 1  # dot would start here if there is one
   # now check that suff is at the end of the name
   return(substr(fn,dotpos,ncf)==suff)
}
```

21.1.1 Example of paste(), sprintf(): Forming File Names

Suppose I wish to create five files, **q1.pdf** through **q5.pdf** consisting of histograms of 100 random N(0,i^2) variates. I could execute the code[7]

[7] The main point here is the string manipulation, creating the file names **fname**. Don't worry about the graphics operations, or check Section 23.15 for details.

```
for (i in 1:5)  {
   fname <- paste("q",i,".pdf")
   pdf(fname)
   hist(rnorm(100,sd=i))
   dev.off()
}
```

The **paste()** function concatenates the string ”q” with the string form of **i** For example, when i = 2, the variable **fname** will be “q 2”.

But that wouldn’t quite work, as it would give me filenames like ”q 2.pdf”. On Linux systems, filenames with embedded spaces create headaches.

One solution would be to use the **sep** argument:

```
for (i in 1:5)  {
   fname <- paste("q",i,".pdf",sep="")
   pdf(fname)
   hist(rnorm(100,sd=i))
   dev.off()
}
```

Here we used an empty string for the separator.

Or, we could use a function borrowed from C:

```
for (i in 1:5)  {
   fname <- sprintf("q%d.pdf",i)
   pdf(fname)
   hist(rnorm(100,sd=i))
   dev.off()
}
```

Since even many C programmers are unaware of the **sprintf()** function, some explanation is needed. This function works just like **printf()**, except that it “prints” to a string, not to the screen. Here we are “printing” to the string **fname**. What are we printing? The function says to first print “q”, then print the character version of **i**, then print “.pdf”. When **i** = 2, for instance, we print ”z2.pdf” to **fname**.

For floating-point quantities, note also the difference between %f and %g formats:

```
> sprintf("abc%fdef",1.5)
[1] "abc1.500000def"
> sprintf("abc%gdef",1.5)
[1] "abc1.5def"
```

21.1.2 Example of grep()

Note that **grep()** searches for a given string in a vector of strings, returning the indices of the strings in which the pattern is found. That makes it perfect for row selection, e.g.

```
> d <- data.frame(cbind(c(0,5,12,13),x))
> d
  V1    x
1  0  xyz
```

```
2  5 yabc
3 12  abc
4 13 <NA>
> dabc <- d[grep("ab",d[,2]),]
> dabc
  V1    x
2  5 yabc
3 12  abc
```

21.1.3 Example of strsplit()

```
> s <- strsplit("a b"," ")
> s
[[1]]
[1] "a" "b"

> s[1]
[[1]]
[1] "a" "b"

> s[[1]]
[1] "a" "b"
> s[[1]][1]
[1] "a"
```

22 Sorting

Ordinary numerical sorting of a vector can be done via **sort()**.

```
> x <- c(13,5,12,5)
> sort(x)
[1]  5  5 12 13
```

If one wants the inverse, use **order()**. For example:

```
> order(x)
[1] 2 4 3 1
```

Here is what **order()**'s output means: The 2 means that **x[2]** is the smallest in **x**; the 4 means that **x[4]** is the second-smallest, etc.

You can use **order()**, together with indexing, to sort data frames. For instance:

```
> y <- read.table("y")
> y
    V1 V2
1  def  2
2   ab  5
3 zzzz  1
> r <- order(y$V2)
> r
[1] 3 1 2
> z <- y[r,]
> z
    V1 V2
3 zzzz  1
1  def  2
2   ab  5
```

What happened here? We called **order()** on the second column of **y**, yielding a vector telling us which numbers from that column should go before which if we were to sort them. The 3 in this vector tells us that **x[3,2]** is the smallest number; the 1 tells us that **x[1,2]** is the second-smallest; and the 2 tells us that **x[2,2]** is the third-smallest.

We then used indexing (Section 8.3.3) to produce the frame sorted by column 2, storing it in **z**.

23 Graphics

R has a very rich set of graphics facilities. The top-level R home page, `http://www.r-project.org/`, has some colorful examples, and there is a very nice display of examples in the R Graph Gallery, `http://addictedtor.free.fr/graphiques`.

I cannot cover even a small part of that material here, but will give you enough foundation to work the basics and learn more.

23.1 The Workhorse of R Graphics, the plot() Function

This is the workhorse function for graphing, serving as the vehicle for producing many different kinds of graphs.

As mentioned earlier, it senses from the type of the object sent to it what type of graph to make, i.e. **plot()**, is a *generic function* (Section 16.2). It is really a placeholder for a family of functions. The function that actually gets called will depend on the class of the object on which it is called.

Let's see what happens when we call **plot()** with an X vector and a Y vector, which are interpreted as a set of pairs in the (X,Y) plane. For example,

```
> plot(c(1,2,3), c(1,2,4))
```

will cause a window to pop up, plotting the points (1,1), (2,2) and (3,4). (Here and below, we will not show the actual plots.)

The points in the graph will be symbolized by empty circles. If you want a different character type, specify a value for the named argument **pch** ("point character"). You can change the size of the character via the named argument **cex**; see Section 23.10.

As noted in Section 23.2, one typically builds a graph, by adding more and more to it in a succession of several commands. So, as a base, we might first draw an empty graph, with only axes. For instance,

```
> plot(c(-3,3), c(-1,5), type = "n", xlab="x", ylab="y")
```

draws axes labeled "x" and "y", the horizontal one ranging from x = -3 to x = 3, and the vertical one ranging from y = -1 to y = 5. The argument **type="n"** means that there is nothing in the graph itself.

23.2 Plotting Multiple Curves on the Same Graph

The **plot()** function works in stages, i.e. you can build up a graph in stages by issuing more and more commands, each of which adds to the graph. For instance, consider the following:

```
> x <- c(1,2,3)
> y <- c(1,3,8)
> plot(x,y)
> lmout <- lm(y ~ x)
> abline(lmout)
```

The call to **plot()** will graph the three points as in our example above. At this point the graph will simply show the three points, along with the X and Y axes with hash marks.

The call to **abline()** then adds a line to the current graph. Now, which line is this? As we know from Section 36.1, the result of the call to the linear regression function **lm()** is a class instance containing the slope and intercept of the fitted line, as well as various other quantities that won't concern us here. We've assigned that class instance to **lmout**. The slope and intercept will now be in **lmout$coefficients**.

Now, what happens when we call **abline()**? This is simply a function that draws a straight line, with the function's arguments being treated as the intercept and slope of the line. For instance, the call **abline(c(2,1))** would draw the line

$$y = 1 \cdot x + 2$$

on whatever graph we've built up so far.

But actually, even **abline()** is a generic function, and since we are invoking it on the output of **lm()**, this version of the function knows that the slope and intercept it needs will be in **lmout$coefficients**, and it plots that line. Note again that it superimposes this line onto the current graph—the one which currently graphs the three points. In other words, the new graph will show both the points and the line.

23.3 Starting a New Graph While Keeping the Old Ones

Each time you call **plot()** (directly or indirectly), the current graph window will be replaced by the new one. If you don't want that to happen, you can on Linux systems call **X11()**. There are similar calls for other platforms.

23.4 The lines() Function

Though there are many options, the two basic arguments to **lines()** are a vector of X values and a vector of Y values. These are interpreted as (X,Y) pairs representing points to be added to the current graph, with lines connecting the points.

For instance, if x and y are the vectors (1.5,2.5) and (3,), then the call

```
> lines(c(1.5,2.5),c(3,3))
```

would add a line from (1.5,3) to (2.5,3) to the present graph.

If you want the lines “connecting the dots” but don’t want the dots themselves, include **type=”l”** in your call to **lines()**, or to **plot()**:

```
> plot(x,y,type="l")
```

23.5 Another Example

Let’s plot nonparametric density estimates (these are basically smoothed histograms) for Exams 1 and 2 from our **exams** file in Section 8.1 in the same graph. We use the function **density()** to generate the estimates. Here are the commands we issue:

```
> d1 = density(testscores$Exam1,from=0,to=100)
> d2 = density(testscores$Exam2,from=0,to=100)
> plot(d2,main="",xlab="")
> lines(d1)
```

Here’s what we did: First, we computed nonparametric density estimates from the two variables, saving them in objects **d1** and **d2** for later use. We then called **plot()** to draw the curve for Exam2. The internal structure of **d2** contains vectors of X and Y coordinates needed by **plot()** to draw the figure. We then called **lines()** to add Exam1’s curve to the graph.

Note that we asked R to have blank labels for the figure as a whole and for the X axis; otherwise, R would have gotten such labels from **d2**, which would have been specific to Exam 2.

The call to **plot()** both initiates the plot and draws the first curve. (Without specifying type=”l”, only the points would have been plotted.) The call to **lines()** then adds the second curve.

You can use the **lty** parameter in **plot()** to specify the type of line, e.g solid, dashed, etc. Type

```
> help(par)
```

to see the various types and their codes.

23.6 Adding Points: the points() Function

The **points()** function adds a set of (x,y)-points, with labels for each, to the currently displayed graph. For instance, in our first example, Section 4, the command

```
points(testscores$Exam1,testscores$Exam3,pch="+")
```

would superimpose onto the current graph the points of the exam scores from that example, using ”+” signs to mark them.

As with most of the other graphics functions, there are lots of options, e.g. point color, background color, etc.

23.7 The legend() Function

A nice function is **legend()**, which is used to add a legend to a multicurve graph. For instance,

```
> legend(2000,31162,legend="CS",lty=1)
```

would place a legend at the point (2000,31162) in the graph, with a little line of type 1 and label of ”CS”. Try it!

23.8 Adding Text: the text() and mtext() Functions

Use the **text()** function to place some text anywhere in the current graph. For example,

```
text(2.5,4,"abc")
```

would write the text ”abc” at the point (2.5,4) in the graph. The center of the string, in this case ”b”, would go at that point.

In order to get a certain string placed exactly where you want it, you may need to engage in some trial and error. R has no ”undo” command (though the ESS interface to R described below does). For that reason, you might want to put all the commands you're using to build up a graph in a file, and then use **source()** to execute them.

But you may find the **locator()** function to be a much quicker way to go. See Section 23.9.

To add text in the margins, use **mtext()**.

23.9 Pinpointing Locations: the locator() Function

Typing

```
locator(1)
```

will tell R that you will click in 1 place in the graph. Once you do so, R will tell you the exact coordinates of the point you clicked on. Call locator(2) to get the locations of 2 places, etc. (Warning: Make sure to include the argument.)

You can combine this, for example, with **text()**, e.g.

```
> text(locator(1),"nv=75")
```

23.10 Changing Character Sizes: the cex() Function

The **cex()** (”character expand”) function allows you to expand or shrink characters within a graph, very useful. You can use it as a named parameter in various graphing functions.

```
text(2.5,4,"abc",cex = 1.5)
```

would print the same text as in our earlier example, but with characters 1.5 times normal size.

23.11 Operations on Axes

You may wish to have the ranges on the X- and Y-axes of your plot to be broader or narrower than the default. You can do this by specifying the **xlim** and/or **ylim** parameters in your call to **plot()** or **points()**. For example, **ylim=c(0,90000)** would specify a range on the Y-axis of 0 to 90000.

This is especially useful if you will be displaying several curves in the same graph. Note that if you do not specify xlim and/or ylim, then draw the largest curve first, so there is room for all of them.

23.12 The polygon() Function

You can use **polygon()** to draw arbitrary polygonal objects, with shading etc. For example, the following code draws the graph of the function $f(x) = 1 - e^{-x}$, then adds a rectangle that approximates the area under the curve from x = 1.2 to x = 1.4:

```
> f <- function(x) return(1-exp(-x))
> curve(f,0,2)
> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),col="gray")
```

In the call to **polygon()** here, the first argument is the set of X coordinates for the rectangle, while the second argument specifies the Y coordinates. The third argument specifies that the rectangle should be shaded in gray; instead we could have, for instance, used the **density** argument for striping.

23.13 Smoothing Points: the lowess() Function

Just plotting a cloud of points, whether connected or not, may turn out to be just an uninformative mess. In many cases, it is better to smooth out the data by fitting a nonparametric regression estimator—nonparametric meaning that it is not necessarily in the form of a straigh line—such as **lowess()**:

```
plot(lowess(x,y))
```

The call **lowess(x,y)** returns the pairs of points on the regression curve, and then **plot()** plots them. Of course, we could get both the cloud and the smoothed curve:

```
plot(x,y)
lines(lowess(x,y))
```

23.14 Graphing Explicit Functions

Say you wanted to plot the function $g(t) = (t^2 + 1)^{0.5}$ for t between 0 and 5. You could use the following R code:

```
g <- function(t) { return (t^2+1)^0.5 }  # define g()
x <- seq(0,5,length=10000)  # x = [0.0004, 0.0008, 0.0012,..., 5]
y <- g(x)  # y = [g(0.0004), g(0.0008), g(0.0012), ..., g(5)]
plot(x,y,type="l")
```

But even better, you could use the **curve()** function:

```
> curve((x^2+1)^0.5,0,5)
```

23.15 Graphical Devices and Saving Graphs to Files

R has the notion of a graphics device. The default device is the screen. If we want to have a graph saved to a file, we must set up another device. For example, if we wish to save as a PDF file, we do something like the following. (Warning: This is actually too tedious an approach, and a shortcut will be presented later on. But the reader should go through this "long way" once, to understand the principles.)

```
> pdf("d12.pdf")
```

This opens a file, which we have chosen here to call **d12.pdf**. We now have two devices open, as we can confirm:

```
> dev.list()
X11 pdf
  2   3
```

The screen is named X11 when R runs on Linux; it is device number 2 here. Our PDF file is device number 3. Our active device is now the PDF file:

```
> dev.cur()
pdf
  3
```

All graphics output will now go to this file instead of to the screen. But what if we wish to save what's already on the screen? We could re-establish the screen as the current device, then copy it to the PDF device, 3:

```
> dev.set(2)
X11
  2
> dev.copy(which=3)
pdf
  3
```

Note carefully that the PDF file is not usable until we close it, which we do as follows:

```
> dev.set(3)
pdf
  3
> dev.off()
X11
  2
```

(We could also close the device by exiting R, though it's probably better to proactively close.)

The above set of operations to print a graph can become tedious, but there is a shortcut:

```
> dev.print(device=pdf,"d12.pdf")
X11
  2
```

This opens the PDF file **d12.pdf**, copies the X11 graph to it, closes the file, and resets X11 as the active device.

23.16 3-Dimensional Plots

There are a number of functions to plot data in three dimensions, such as **persp()** and **wireframe()**, which draw surfaces, and **cloud()**, which draws three-dimensional scatter plots. There are many more.

For **wireframe()** and **cloud()**, one loads the **lattice** library. Here is an example:

```
> a <- 1:10
> b <- 1:15
> eg <- expand.grid(x=a,y=b)
> eg$z <- eg$x^2 + eg$x * eg$y
> wireframe(z ~ x+y, eg)
```

The call to **expand.grid()** creates a data frame, consisting of two columns named **x** and **y**, combining all the values of the two inputs. Here **a** and **b** had 10 and 15 values, respectively, so the resulting data frame will have 150 rows.

We then added a third column, named **z**, as a function of the first two columns. Our call to **wireframe()** then creates the graph. Note that **z**, **x** and **y** of course refer to names of columns in **eg**.

All the points would be connected as a surface (like connecting points by lines in two dimensions). With **cloud()**, though, the points would just be isolated.

For **wireframe()**, the (X,Y) pairs must form a rectangular grid, though not necessarily evenly spaced.

Note that the data frame that is input to **wireframe()** need not have been created by **expand.grid()**.

By the way, these functions have many different options. A nice one for **wireframe()**, for instance, is **shade=T**, which makes it all easier to see.

23.17 The Rest of the Story

As mentioned, we are not even scratching the surface here. See one of the references cited in Section 39.2.4 to learn more about R's excellent set of graphics facilities.

24 The Innocuous c() Function

As seen before, the concatenation function **c()** does what its name implies, e.g.

```
> x <- c(1,2,3)
> y <- c(4,5)
> c(x,y)
[1] 1 2 3 4 5
```

But when mixing modes, **c()** will apply the precedence discussed in Section 7.3. Note carefully too that by default **c()** will recurse down through data structures, resulting in a “flatten” operation.

25 The Versatile attach() Function

Abstractly described, the call **attach(x)** loads the namespace in the *database* **x**, making those objects available via those names. There are two main contexts for this:

- The database **x** could be a list. The result is that, say, **x$u** will now be referenceable as simply **$u**.

  The typical usage of this is for data frames, which you may recall are in fact lists. In the example in Section 8.1, for instance, we could type

  ```
  > attach(testscores)
  ```

  This command tells R that from now on, when we refer, for example, to **Exam3**, we mean **testscores$Exam3**:

  ```
  > mean(Exam3)
  [1] 50.05455
  ```

  If we want R to stop doing that, we use **detach()**, e.g.

  ```
  > detach()
  > mean(Exam3)
  Error in mean(Exam3) : Object "Exam3" not found
  ```

- If we had previously saved a collection of objects to disk using **save()**, we can now restore them to our R session, by using **attach()**.

26 Debugging

The R base package includes a number of debugging facilities. They are nowhere near what a good debugging tool offers, but with skillful usage they can be effective.

A much more functional debugging package is available for R, of course called **debug**. I will discuss this in Section 26.4.

26.1 The debug() Function

One of the tools R offers for debugging your R code is the built-in function **debug()**. It works in a manner similar to C debuggers such as GDB.

26.1.1 Setting Breakpoints

Say for example we suspect that our bug is in the function **f()**. We enable debugging by typing

```
> debug(f)
```

This will set a breakpoint at the beginning of **f()**.

To turn off this kind of debugging for a function **f()**, type

```
> undebug(f)
```

Note that if you simultaneously have a separate window open in which you are editing your source code, and you had executed **debug(f), then if** you reload using **source()**, the effect is that of calling **undebug(f)**.

If we wish to set breakpoints at the line level, we insert a line

```
browser()
```

before line at which we wish to break.

You can make a breakpoint set in this fashion conditional by placing it within an **if** statement, e.g.

```
if (k == 6) browser()
```

You may wish to add an argument named, say, **dbg**, to most of your functions, with **dbg = 1** meaning that you wish to debug that part of the code. The above then may look like

```
if (dbg && k == 6) browser()
```

26.1.2 Stepping through Our Code

When you execute your code and hit a breakpoint, you enter the debugger, termed the *browser* in R. The command prompt will now be something like

```
Browse[1]
```

instead of just >. Then you can invoke various debugging operations, such as:

- **n or Enter:** You can single-step through the code by hitting the Enter key. (If it is a line-level breakpoint, you must hit n the first time, then Enter after that.)
- **c:** You can skip to the end of the "current context" (a loop or a function) by typing c.
- **where:** You can get a stack report by typing `where`.
- **Q:** You can return to the > prompt, i.e. exit the debugger, by typing Q.
- All normal R operations and functions are still available to you. So for instance to query the value of a variable, just type its name, as you would in ordinary interactive usage of R. If the variable's name is one of the **debug()** commands, though, say c, you'll need to do something like **print(c)** to print it out.

26.2 Automating Actions with the trace() Function

The **trace()** function is quite flexible and powerful, though it takes some initial effort to learn. I will discuss some of the simpler usage forms here.

The call

```
> trace(f,t)
```

would instruct R to call the function **t()** every time we enter the function **r()**. For instance, say we wish to set a breakpoint at the beginning of the function **gy()**. We could do this by the command

```
> trace(gy,browser)
```

This would have the same effect as placing the command **browser()** in our source code for **gy()**, but would be quicker and more convenient than inserting such a line, saving the file and rerunning **source()** to load in the new version of the file.

It would also be quicker and more convenient to undo, by simply running

```
> untrace(gy)
```

You can turn tracing on or off globally by calling **tracingState()**, with the argument TRUE to turn it on, FALSE to turn it off. Recall too that these boolean constants in R can be abbreviated T and F.

26.3 Performing Checks After a Crash with the traceback() and debugger() Functions

Say your R code crashes when you are not running the debugger. There is still a debugging tool available to you after the fact: You can do a “post mortem” by simply calling **traceback()**.

You can get a lot more if you set R up to dump frames on a crash:

```
> options(error=dump.frames)
```

If you’ve done this, then after a crash run

```
> debugger()
```

You will then be presented with a choice of levels of function calls to look at. For each one that you choose, you can take a look at the values of the variables there. After browsing through one level, you can return to the **debugger()** main menu by hitting n.

26.4 The debug Package

The **debug** package provides a more usable debugging interface than R's built-in facilities do. It features a pop-up window in which you can watch your progress as you step through your source code, gives you the ability to easily set breakpoints, etc.

It requires another package, **mvbutils**, and the Tcl/Tk scripting and graphics system. The latter is commonly included in Linux distributions, and is freely downloadable for all the major platforms. It suffers from a less-than-perfect display, but is definitely worthwhile, much better than R's built-in debugging tools.

26.4.1 Installation

Choose an installation directory, say **/MyR**. Then install **mvbutils** and **debug**:

```
> install.packages("mvbutils","/MyR")
> install.packages("debug","/MyR")
```

For R version 2.5.0, I found that a bug in R caused the **debug** package to fail. I then installed the patched version of 2.5.0, and **debug** worked fine. On one machine, I encountered a Tcl/Tk problem when I tried to load **debug**. I fixed that (I was on a Linux system) by setting the environment variable, in my case by typing

```
% setenv TCL_LIBRARY /usr/share/tcl8.4
```

26.4.2 Path Issues

Each time you wish to use **debug**, load it by executing

```
> .libPaths("/MyR")
> library(debug)
```

Or, place these in an R startup file, say **.Rprofile** in the directory in which you want these commands to run automatically.

Or, create a file **.Renviron** in your home directory, consisting of the line

```
R_LIBS=~/MyR
```

26.4.3 Usage

Now you are ready to debug. Here are the main points:

- Breakpoints are first set at the function level. Say you have a function **f()** at which you wish to break. Then type

  ```
  > mtrace(f)
  ```

Do this for each function at which you want a breakpoint.

- Then go ahead and start your program. (I'm assuming that your program itself consists of a function.) Execution will pause at **f()**, and a window will pop up, showing the source code for that function. The current line will be highlighted in green. Back in the R interactive window, you'll see a prompt D(1)>.
- At this point, you can single-step through your code by repeatedly hitting the Enter key. You can print the values of variables as you usually do in R's interactive mode.
- You can set finer breakpoints, at the line level, using **bp()**. Once you are in **f()**, for instance, to set a breakpoint at line 12 in that function type

```
D(1)> bp(12)
```

- To set a conditional breakpoint, say at line 12 with the condition **k == 5**, issue **bp(12,k==5)**.
- To avoid single-stepping, issue **go()**, which will execute continuously until the next breakpoint.
- To set a temporary breakpoint at line n, issue **go(n)**.
- To restart execution of the function, issue **skip(1)**.
- If there is an execution error, the offending line will be highlighted.
- To cancel all **mtrace()** breaks, issue **mtrace.off()**. To cancel one for a particular function **f()**, issue **mtrace(f,tracing=F)**.
- To cancel a breakpoint, say at line 12, issue **bp(12,F)**.
- To quit, issue **qqq()**.
- For more details, see the extensive online help, e.g. by typing

```
D(1)> ?bp
```

26.5 Ensuring Consistency with the set.seed() Function

If you're doing anything with random numbers, you'll need to be able to reproduce the same stream of numbers each time you run your program during the debugging session. To do this, type

```
> set.seed(8888)  # or your favorite number as an argument
```

26.6 Syntax and Runtime Errors

The most common syntax errors will be lack of matching parentheses, brackets or braces. When you encounter a syntax error, this is the first thing you should check and double-check. I highly recommend that you use a text editor, say Vim, that does parenthesis matching and syntax coloring for R.

Beware that often when you get a message saying there is a syntax error on a certain line, the error may well be elsewhere. This can occur with any language, but R seems especially prone to it.

If it just isn't obvious to you where your syntax error is, I recommend selectively commenting-out some of your code, thus enabling you to better pinpoint the location of the syntax problem.

If during a run you get a message

```
could not find function "evaluator"
```

and a particular function call is cited, it means that the interpreter cannot find that function. You may have forgotten to load a library or source a code file.

You may sometimes get messages like,

```
There were 50 or more warnings (use warnings() to see the first 50)
```

These should be heeded; run **warnings()**, as suggested. The problem could range from nonconvergence of an algorithm to misspecification of a matrix argument to a function. In many cases, the program output may be invalid, though it may well be fine too, say with a message "fitted probabilities numerically 0 or 1 occurred in: glm..."

27 Startup Files

If there are R commands you would like to have executed at the beginning of every R session, you can place them in a file **.Rprofile** either in your home directory or in the directory from which you are running R. The latter directory is searched for such a file first, which allows you to customize for a particular project.

Other information on startup files is available by typing

```
> help(.Rprofile)
```

28 Session Data

As you proceed through an interactive R session, R will record the commands you submit. And as you long as you answer yes to the question "Save workspace image?" put to you when you quit the session, R will save all the objects you created in that session, and restore them in your next session. You thus do not have to recreate the objects again from scratch if you wish to continue work from before.

The saved workspace file is named **.Rdata**, and is located either in the directory from which you invoked this R session (Linux) or in the R installation directory (Windows). Note that that means that in Windows, if you use R from various different directories, each save operation will overwrite the last. That makes Linux more convenient, but note that the file can be quite voluminous, so be sure to delete it if you are no longer working on that particular project.

You can also save the image yourself, to whatever file you wish, by calling **save.image()**. You can restore the workspace from that file later on by calling **load()**.

29 Packages (Libraries)

29.1 Basic Notions

R uses packages to store groups of related pieces of software.[8] The libraries are visible as subdirectories of your library directory in your R installation tree, e.g. **/usr/lib/R/library**. The ones automatically loaded when you start R include the **base** subdirectory, but in order to save memory and time, R does not automatically load all the packages. You can check which packages are currently loaded by typing

```
> .path.package()
```

29.2 Loading a Package from Your Hard Drive

If you need a package which is in your R installation but not loaded into memory yet, you must request it. For instance, suppose you wish to generate multivariate normal random vectors. The function **mvrnorm()** in the package MASS does this. So, load the library:

```
> library(MASS)
```

Then **mvrnorm()** will now be ready to use. (As will be its documentation. Before you loaded MASS, “help(mvrnorm)” would have given an error message).

29.3 Downloading a Package from the Web

However, the package you want may not be in your R installation. One of the big advantages of open-source software is that people love to share. Thus people all over the world have written their own special-purpose R packages, placing them in the CRAN repository and elsewhere.

29.3.1 Using install.package()

One way to install a package is, not surprisingly, to use the **install.packages**() function.

As an example, suppose you wish to use the **mvtnorm** package, which computes multivariate normal cdf’s and other quantities. Choose a directory in which you wish to install the package (and maybe others in the future), say **/a/b/c**. Then at the R prompt, type

```
> install.packages("mvtnorm","/a/b/c/")
```

This will cause R to automatically go to CRAN, download the package, compile it, and load it into a new directory **/a/b/c/mvtnorm**.

You do have to tell R where to find that package, though, which you can do via the **.libPaths**() function:

[8] This is one of the very few differences between R and S. In S, packages are called *libraries*, and many of the functions which deal with them are different from those in R.

```
> .libPaths("/a/b/c/")
```

This will add that new directory to the ones R was already using. If you use that directory often enough, you may wish to add that call to **.libPaths()** in your **.Rprofile** startup file in your home directory.

A call to **.libPaths()**, without an argument, will show you a list of all the places R will currently look at for loading a package when requested.

29.3.2 Using "R CMD INSTALL"

Sometimes one needs to install "by hand," to do modifications needed to make a particular R package work on your system. The following example will show how I did so in one particular instance, and will serve as a case study on how to "scramble" if ordinary methods don't work.

I wanted to install a package **Rmpi** on our department's instructional machines, in the directory **/home/matloff/R**. I tried using **install.packages()** first, but found that the automated process could not find the MPI library on our machines. The problem was that R was looking for those files in **/usr/local/lam**, whereas I knew they were in **/usr/local/LAM**.

So, I downloaded the **Rmpi** files, in the packed form **Rmpi_0.5-3.tar.gz**. I unpacked that file in my directory **/tmp**, producing a directory **/tmp/Rmpi**.

Now if there had been no problem, I could have just typed

```
% R CMD INSTALL -l /home/matloff/R  Rmpi
```

from within the **/tmp** directory. That command would install the package contained in **/tmp/Rmpi**, placing it in **/home/matloff/R**. This would have been an alternative to calling **install.packages()**.

But as noted, I had to deal with a problem. Within the **/tmp/Rmpi** directory there was a **configure** file, so I ran

```
% configure --help
```

on my Linux command line. It told me that I could specify the location of my MPI files to **configure** as follows:

```
% configure --with-mpi=/usr/local/LAM
```

This is if one runs **configure** directly, but I ran it via R:

```
% R CMD INSTALL -l /home/matloff/R  Rmpi --configure-args=--with-mpi=/usr/local/LAM
```

Well, that seemed to work, in the sense that R did install the package, but it also noted that it had a problem with the threads library on our machines. Sure enough, when I tried to load **Rmpi**, I got a runtime error, saying that a certain threads function wasn't there.

I knew that our threads library was fine, so I went into **configure** file and commented-out two lines:

```
# if test $ac_cv_lib_pthread_main = yes; then
     MPI_LIBS="$MPI_LIBS -lpthread"
# fi
```

In other words, I forced it to use what I knew (or was fairly sure) would work. I then reran "R CMD INSTALL," and the package then loaded with no problem.

29.4 Documentation

You can get a list of functions in a package by calling **library()** with the **help** argument, e.g.

```
> library(help=mvrnorm)
```

for help on the **mvrnorm** package.

29.5 Built-in Data Sets

R includes a few real data sets, for use in teaching, research or in testing software. Type the following:

```
> library(utils)
> help(data)
```

Here **data** is contained within the **utils** package. We load that package, and use **help()** to see what's in it, in this case various data sets. We can load any of them but typing its name, e.g.

```
> LakeHuron
```

30 Handy Miscellaneous Features

30.1 Scrolling through the Command History

During a session, you can scroll back and forth in your command history by typing ctrl-p and ctrl-n. You can also use the **history()** command to list your more recent commands (which will be run through the pager). Set the named argument **max.show=Inf** if you want to see all of them.

30.2 The Pager

This displays material one screen at a time. It is automatically invoked by some R commends, such as **help()**, and you can invoke it yourself on lengthy output. For instance, if you have a long vector **x** and wish to display it one screen at a time, then instead of typing

```
> x
```

type

```
> page(x)
```

Type “/abc” to search for the string ”abc” in the pager output, hit q to quit the pager, etc.

30.3 Calculating Run Time

If you are not sure which of several approaches to use to get the fastest R code, you can time them with the function **system.time()**. An example is shown in Section 14.

31 Writing C/C++ Functions to be Called from R

You may wish to write your own C/C++ functions to be called from R, as they may run much faster than if you wrote them in R. (Recall, though, that you may get a lot of speed out of R if you avoid using loops. See Section 13.) The SWIG package can be used for this; see `http://www.swig.org`.

32 Parallel R

Since many R users have very large computational needs, various tools for some kind of parallel operation of R have been devised. These involve parallel invocations of R, communicating through a network.

32.1 Rmpi

The Rmpi provides an interface for R to MPI, the popular message-passing system.

32.1.1 Installation

Say you want to install in the directory **/a/b/c/**. The easiest way to do so is

```
> install.packages("Rmpi","/a/b/c/")
```

This will install Rmpi in the directory **/a/b/c/Rmpi**.

I found that in order to get it to work, I needed to edit the shell script **Rslaves.sh**, replacing

```
#       $R_HOME/bin/R --no-init-file --slave --no-save < $1 > $hn.$2.$$.log 2>&1# else
#       $R_HOME/bin/R --no-init-file --slave --no-save < $1 > /dev/null 2>&1
```

by

```
        $R_HOME/bin/R --slave --no-save < $1 > $hn.$2.$$.log 2>&1
else
        $R_HOME/bin/R --slave --no-save < $1 > /dev/null 2>&1
```

You'll need to arrange for the directory **/a/b/c** (not **/a/b/c/Rmpi**) to be added to your R library search path. I recommend placing a line

```
.libPaths("/a/b/c/")
```

in a file **.Rprofile** in your home directory.

32.1.2 Usage

Fire up MPI, and then in R load in Rmpi, by typing

```
> library(Rmpi)
```

Then start Rmpi:

```
> mpi.spawn.Rslaves()
```

This will start R on all machines in the group you started MPI on. Optionally, you can specify fewer machines via the named argument **nslaves**.

The first time you do this, try this test:

```
mpi.remote.exec(paste("I am",mpi.comm.rank(),"of",mpi.comm.size()))
```

The available functions are similar to (and call) those of, such as

- **mpi.comm.size():**

  Returns the number of MPI processes, including the master that spawned the other processes. The master will be rank 0.

- **mpi.comm.rank():**

  Returns the rank of the process that executes it.

- **mpi.send(), mpi.recv()**:

  The usual send/receive operations.

- **mpi.bcast(), mpi.scatter(), mpi.gather():**

  The usual broadcast, scatter and gather operations.

- Etc.

Details are available at:

- `http://cran.r-project.org/web/packages/Rmpi/index.html`

  Site for download of package and manual.

- `http://ace.acadiau.ca/math/ACMMaC/Rmpi/sample.html`

  Nice tutorial.

But we forego details here, as snow provides a nicer programmer interface, to be described next.

32.2 snow

snow runs on top of Rmpi or directly via sockets, allowing the programmer to more conveniently express the parallel disposition of work.

For instance, just as the ordinary R function **apply()** applies the same function to all rows of a matrix, the snow function **parApply()** does that in parallel, across multiple machines; different machines will work on different rows.

32.2.1 Installation

Follow the same pattern as described above for Rmpi. If you plan to have snow run on top of Rmpi, you'll of course need the latter too.

Again, I needed to change a shell script, **RSOCKnode.sh**, replacing

```
${RPROG:-R} --vanilla <<EOF > ${OUT:-/dev/null} 2>&1 &
```

by

```
${RPROG:-R} --no-save --no-restore <<EOF > ${OUT:-/dev/null} 2>&1 &
```

32.2.2 Starting snow

Make sure snow is in your library path (see material on Rmpi above). Then load snow:

```
> library(snow)
```

One then sets up a cluster, by calling the snow function **makeCluster()**. The named argument **type** of that function indicates the networking platform, e.g. "MPI," "PVM" or "SOCK." The last indicates that you wish snow to run on TCP/IP sockets that it creates itself, rather than going through MPI. You may prefer to use MPI for this, as it provides more flexibility, since one's code could include calls to both snow functions and MPI (i.e. Rmpi) functions. However, you may not want to bother with MPI if snow itself is enough. In the examples here, I used "SOCK," on machines named **pc48** and **pc49**, setting up the cluster this way:

```
> cls <- makeCluster(type="SOCK",c("pc48","pc49"))
```

For MPI or PVM, one specifies the number of nodes to create, rather than specifying the nodes themselves.

Note that the above R code sets up worker nodes at the machines named **pc48** and **pc49**; these are in addition to the master node, which is the machine on which that R code is executed

There are various other optional arguments. One you may find useful is **outfile**, which records the result of the call in the file **outfile**. This can be helpful if the call fails.

32.2.3 Overview of Available Functions

Let's look at a simple example of multiplication of a vector by a matrix. We set up a test matrix:

```
> a <- matrix(c(1,2,3,4,5,6,7,8,9,10,11,12),nrow=6)
> a
     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9
[4,]    4   10
[5,]    5   11
[6,]    6   12
```

We will multiply the vector $(1,1)^T$ (T meaning transpose) by our matrix **a**, by defining a dot product function:

```
> dot <- function(x,y) {return(x%*%y)}
```

Let's test it using the ordinary **apply()**:

```
> apply(a,1,dot,c(1,1))
[1]  8 10 12 14 16 18
```

To review your R, note that this applies the function **dot()** to each row (indicated by the 1, with 2 meaning column) of **a** playing the role of the first argument to **dot()**, and with c(1,1) playing the role of the second argument.

Now let's do this in parallel, across our two machines in our cluster **cls**:

```
> parApply(cls,a,1,dot,c(1,1))
[1]  8 10 12 14 16 18
```

The function **clusterCall(cls,f,args)** applies the given function **f()** at each worker node in the cluster **cls**, using the arguments provided in **args**.

The function **clusterExport(cls,varlist)** copies the variables in the list **varlist** to each worker in the cluster **cls**. You can use this to avoid constant shipping of large data sets from the master to the workers; you just do so once, using **clusterExport()** on the corresponding variables, and then access those variables as global. For instance:

```
> z <- function() return(x)
> x <- 5
> y <- 12
> clusterExport(cls,list("x","y"))
> clusterCall(cls,z)
[[1]]
[1] 5

[[2]]
[1] 5
```

The function **clusterEvalQ(cls,expression)** runs **expression** at each worker node in **cls**. Continuing the above example, we have

```
> clusterEvalQ(cls,x <- x+1)
[[1]]
[1] 6

[[2]]
[1] 6

> clusterCall(cls,z)
[[1]]
[1] 6

[[2]]
[1] 6

> x
[1] 5
```

Note that **x** still has its original version back at the master.

The function **clusterApply(cls,individualargs,f,commonargsgohere)** runs **f()** at each worker node in **cls**, with arguments as follows. The first argument to **f()** for worker i is the ith element of the list **individualargs**, i.e. **individualargs[[i]]**, and optionally one can give additional arguments for **f()** following **f()** in the argument list for **clusterApply()**.

Here for instance is how we can assign an ID to each worker node, like MPI **rank**:[9]

```
> myid <- 0
> clusterExport(cls,"myid")
> setid <- function(i) {myid <<- i}  # note superassignment operator
> clusterApply(cls,1:2,setid)
[[1]]
[1] 1

[[2]]
[1] 2

> clusterCall(cls,function() {return(myid)})
[[1]]
[1] 1

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

Recall that the way snow works is to have a master node, the one from which you invoke functions like **parApply()**, and then a cluster of worker nodes. Suppose the function you specify as an argument to **parApply()** is **f()**, and that **f()** calls **g()**. Then **f()** itself (its code) is passed to the cluster nodes, but **g()** is not. Therefore you must first pass **g()** to the cluster nodes via a call to **clusterExport()**.

Don't forget to stop your clusters before exiting R, by calling **stopCluster(clustername)**.

There are various other useful snow functions. See the user's manual for details.

32.2.4 More Snow Examples

In the first example, we do a kind of one-level Quicksort, breaking the array into piles, Quicksort-style, then having each cluster node work on its pile, then consolidate. We assume a two-node cluster.

[9] I don't see a provision in snow itself that does this.

```
# sorts the array x on the cluster cls
qs <- function(cls,x) {
   pvt <- x[1]  # pivot
   chunks <- list()
   chunks[[1]] <- x[x <= pvt]  # low pile
   chunks[[2]] <- x[x > pvt]  # high pile
   # now parcel out the piles to the clusters, which sort the piles
   rcvd <- clusterApply(cls,chunks,sort)
   lx <- length(x)
   lc1 <- length(rcvd[[1]])
   lc2 <- length(rcvd[[2]])
   y <- vector(length=lx)
   if (lc1 > 0) y[1:lc1] <- rcvd[[1]]
   if (lc2 > 0) y[(lc1+1):lx] <- rcvd[[2]]
   return(y)
}
```

The second example implements the Shearsort sorting algorithm. Here one imagines the nodes laid out as an n x n matrix (they may really have such a configuration). Here is the pseudocode:

```
for i = 1 to log2(n^2) + 1
    if i is odd
       sort each even row in descending order
       sort each odd row in ascending order
    else
       sort each column is ascending order
```

And here is the Snow code:

```
is <- function(cls,dm) {
   n <- nrow(dm)
   numsteps <- ceiling(log2(n*n)) + 1
   for (step in 1:numsteps) {
      if (step %% 2 == 1) {
         # attach a row ID to each row, so will know odd/even
         augdm <- cbind(1:n,dm)
         # parcel out to the cluster members for sorting
         dm <- parApply(cls,augdm,1,augsort)
         dm <- t(dm)  # recall need to transpose after apply()
      } else dm <- parApply(cls,dm,2,sort)
   }
   return(dm)
}

augsort <- function(augdmrow) {
   nelt <- length(augdmrow)
   if (augdmrow[1] %% 2 == 0) {
      return(sort(augdmrow[2:nelt],decreasing=T))
   } else return(sort(augdmrow[2:nelt]))
}
```

32.2.5 Parallel Simulation, Including the Bootstrap

If you wish to use snow on simulation code, including bootstrapping, you need to make sure that the random number streams at each cluster node are independent. Indeed, just think of what would happen if you just take the default random number seed—you'll get identical results at all the nodes, which certainly would defeat the purpose of parallel operation!

The careful way to do this is to install the R package **rlecuyer**. Then, before running a simulation, call **clusterSetupRNG()**, which in its simplest form consists simply of

```
clusterSetupRNG(cl)
```

for the cluster **cl**.

32.2.6 Example

Here we estimate the prediction error rate using “leaving one out” cross-validation in logistic regression.

```
# working on the cluster cls, find prediction error rate for the data
# matrix dm, with the response variable having index resp and the
# predictors having indices prdids
prerr <- function(cls,dm,resp,prdids) {
   # set up an artificial matrix for parApply()
   nr <- nrow(dm)
   loopmat <- matrix(1:nr,nrow=nr)
   # parcel out the rows of loopmat to the various members of the
   # cluster cls; for each row, they will apply delone() with arguments
   # being that row, dm, resp and prdids; the return vector consists
   # of 1s and 0s, 1 meaning that our prediction was wrong
   errs <- parApply(cls,loopmat,1,delone,dm,resp,prdids)
   return(sum(errs)/nr)
}

# temporarily delete row delrow from the data matrix dm, with the
# response variable having index resp and the predictors having indices
# prdids
delone <- function(delrow,dm,resp,prdids) {
   # get all indices except delrow
   therest <- ab1(1,delrow,nrow(dm))
   # fit the model
   lmout <- glm(dm[therest,resp] ~ dm[therest,prdids], family=binomial)
   # predict the deleted row
   cf <- as.numeric(lmout$coef)
   predvalue <- logit(dm[delrow,prdids],cf)
   if (predvalue > 0.5) {predvalue <- 1}
   else {predvalue <- 0)
   return(abs(dm[delrow,resp]-predvalue))
}

# "allbutone": returns c(a,a+1,...,b-1,b+1,...c)
ab1 <- function(a,b,c) {
   if (a == b) return((b+1):c)
   if (b == c) return(a:(b-1))
   return(c(a:(b-1),(b+1):c))
}

# finds the value of the logistic function at a given x for a given set
# of coefficients b
logit <- function(x,b) {
   lin <- c(1,x) %*% b
   return(1/(1+exp(-lin)))
}
```

32.2.7 To Learn More about snow

I recommend the following Web pages:

- `http://cran.cnr.berkeley.edu/web/packages/snow/index.html`
  CRAN page for snow; the package and the manual are here.

- http://www.bepress.com/cgi/viewcontent.cgi?article=1016&context=uwbiostat
  A research paper.
- http://www.cs.uiowa.edu/~luke/R/cluster/cluster.html
  Brief intro by the author.
- http://www.sfu.ca/~sblay/R/snow.html#clusterCall
  Examples, short but useful.

33 Using R from Python

Python is an elegant and powerful language, but lacks built-in facilities for statistical and data manipulation, two areas in which R excels. Thus an interface between the two languages would be highly useful; RPy is probably the most popular of these. RPy is a Python module that allows access to R from Python. For extra efficiency, it can be used in conjunction with NumPy.

You can build the module from the source, available from http://rpy.sourceforge.net, or download a prebuilt version. If you are running Ubuntu, simply type

```
sudo apt-get install python-rpy
```

To load RPy from Python (whether in Python interactive mode or from code), execute

```
from rpy import *
```

This will load a variable **r**, which is a Python class instance.

Running R from Python is in principle quite simple. For instance, running

```
>>> r.hist(r.rnorm(100))
```

from the Python prompt will call the R function **rnorm()** to produce 100 standard normal variates, and then input those values into R's histogram function, **hist()**. As you can see, R names are prefixed by **r.**, reflecting the fact that Python wrappers for R functions are members of the class instance **r**.[10]

By the way, note that the above code will, if not refined, produce ugly output, with your (possibly voluminous!) data appearing as the graph title and the X-axis label. You can avoid this by writing, for example,

```
>>> r.hist(r.rnorm(100),main='',xlab='')
```

RPy syntax is sometimes less simple than the above examples would lead us to believe. The problem is that there may be a clash of R and Python syntax. Consider for instance a call to the R linear model function **lm()**. In our example, we will predict **b** from **a**:

[10] They are loaded dynamically, as you use them.

```
>>> a = [5,12,13]
>>> b = [10,28,30]
>>> lmout = r.lm(’v2 ~ v1’,data=r.data_frame(v1=a,v2=b))
```

This is somewhat more complex than it would have been if done directly in R. What are the issues here?

First, since Python syntax does not include the tilde character, we needed to specify the model formula via a string. Since this is done in R anyway, this is not a major departure.

Second, we needed a data frame to contain our data. We created one using R’s **data.frame()** function, but note that again due to syntax clash issues, RPy converts periods in function names to underscores, so we need to call **r.data_frame()**. Note that in this call, we named the columns of our data frame **v1** and **v2**, and then used these in our model formula.

The output object is a Python dictionary, as can be seen:

```
>>> lmout
{’qr’: {’pivot’: [1, 2], ’qr’: array([[ -1.73205081, -17.32050808],
       [  0.57735027,  -6.164414  ],
       [  0.57735027,   0.78355007]]), ’qraux’: [1.5773502691896257, 1.6213286481208891], ’rank’: 2, ’tol’: 9.999
```

You should recognize the various attributes of **lm()** objects there. For example, the coefficients of the fitted regression line, which would be contained in **lmout$coefficients** if this were done in R, are here in Python as **lmout[’coefficients’**. So, we can access those coefficients accordingly, e.g.

```
>>> lmout[’coefficients’]
{’v1’: 2.5263157894736841, ’(Intercept)’: -2.5964912280701729}
>>> lmout[’coefficients’][’v1’]
2.5263157894736841
```

One can also submit R commands to work on variables in R’s namespace, using the function **r()**. This is convenient if there are many syntax clashes. Here is how we could run the **wireframe()** example in Section 23.16 in RPy:

```
>>> r.library(’lattice’)
>>> r.assign(’a’,a)
>>> r.assign(’b’,b)
>>> r(’g <- expand.grid(a,b)’)
>>> r(’g$Var3 <- g$Var1^2 + g$Var1 * g$Var2’)
>>> r(’wireframe(Var3 ~ Var1+Var2,g)’)
>>> r(’plot(wireframe(Var3 ~ Var1+Var2,g))’)
```

We first used **r.assign()** to copy a variable from Python’s namespace to R’s. We then ran **expand.grid()** (with a period in the name instead of an underscore, since we are running in R’s namespace), assigning the result to **g**. Again, the latter is in R’s namespace.

Note that the call to **wireframe()** did not automatically display the plot, so we needed to call **plot()**.

The official documentation for RPY is at `http://rpy.sourceforge.net/rpy/doc/rpy.pdf`, with a useful presentation available at `http://www.daimi.au.dk/~besen/TBiB2007/lecture-notes/rpy.html`.

34 Tools

34.1 Using R from emacs

There is a very popular package which allows one to run R (and some other statistical packages) from within emacs, ESS. I personally do not use it, but it clearly has some powerful features for those who wish to put in a bit of time to learn the package. As described in the R FAQ, ESS offers R users:

> R support contains code for editing R source code (syntactic indentation and highlighting of source code, partial evaluations of code, loading and error-checking of code, and source code revision maintenance) and documentation (syntactic indentation and highlighting of source code, sending examples to running ESS process, and previewing), interacting with an inferior R process from within Emacs (command-line editing, searchable command history, command-line completion of R object and file names, quick access to object and search lists, transcript recording, and an interface to the help system), and transcript manipulation (recording and saving transcript files, manipulating and editing saved transcripts, and re-evaluating commands from transcript files).

34.2 GUIs for R

As seen above, one submits commands to R via text, rather than mouse clicks in a Graphical User Interface (GUI). If you can't live without GUIs, you should consider using one of the free GUIs that have been developed for R, e.g. R Commander (`http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/`), JGR (`http://stats.math.uni-augsburg.de/JGR/`), or the Eclipse plug-in StatEt (`http://jgr.markushelbig.org/JGR.html`).

35 Inference on Simple Means and Proportions

The function **t.test** provides Student-t confidence intervals and hypothesis tests in one- and two-mean situations. It returns an object of class **htest**, which includes various components; type

```
> ?t.test
```

for details.

There is a similar function **prop.test** for proportions.

36 Linear and Generalized Linear Models

R has a very rich set of facilities for linear models. At least two entire books have been written on this one aspect of R! But we'll give some of the basics here.

36.1 Linear Regression Analysis

In statistical *regression analysis*, we try to predict a *response variable* Y by one or more *predictor variables*, $X^{(1)}, X^{(2)}, ..., X^{(k)}$. The regression function is defined to be

$$m(t) = E(Y|X^{(1)} = t_1, ..., X^{(k)} = t_k) \tag{1}$$

where t is the vector $(t_1, ..., t_k)$.

Typically we model m(t) as linear in the t_i, i.e. we assume that

$$m(t_1, ..., t_k) = \beta_0 + \beta_1 t_1 + ... + \beta_k t_k \tag{2}$$

Here the β_i are population values, to be estimated via our sample data.

In our example in Section 8.1, let's try to predict the third exam score from the first. (Our goal here may not be prediction *per se*, but rather to see whether the two exams are highly correlated.)

In R, we can use **lm()** ("linear model") function to fit a linear model here, meaning that we will find the values of c and d for which c + d Exam1 best predicts Exam3:

```
> fit1 <- lm(Exam3 ~ Exam1,data=testscores)
> fit1$coefficients
(Intercept)        Exam1
  3.7008841    0.7458898
```

We find that c = 3.7 and d = 0.75.

The notation

```
Exam3 ~ Exam1
```

means that we wish to predict **Exam3** from **Exam1**.

```
Exam3 ~ Exam1 + Exam2
```

would mean to predict **Exam3** from **Exam2** and **Exam2**.

```
Exam3 ~ Exam1 * Exam2
```

means to predict **Exam3** from **Exam2** and **Exam2**, with an interaction term, i.e.

```
E(Exam3 | Exam1, Exam2) = a + b Exam1 + c Exam2 + d Exam1*Exam2
```

The interaction term would be referred to as `Exam1:Exam2` by R.

Note that **coefficients** is a member variable within the **lm** class, hence the $ sign as we saw in Section 7.

Another member variable of that class is **formula**, a string in which we specify the regression model, in this case predicting Exam3 from Exam1. Note by the way that since the model is a string, we can program many models in a loop, using R's string manipulation facilities (Section 21).

Actually, the **fit1** object contains a lot more information, such as **fit1$residuals**, a vector consisting of the prediction errors, i.e. how far off our predicted values are from the real ones.

If we simply call **summary()**, as we did above, it prints out, but the return value of that function is also an object, of class **summary.lm**.

If we make sure to have our data in matrix form rather than as a data frame (applying **as.matrix()** if necessary), then we can conveniently specify multiple predictors by making use of R's indexing operations. For instance, if we have a data matrix **z**,

```
> lm(z[,1] ~ z[,2:3])
```

would be easier to type than

```
> lm(z[,1] ~ z[,2]+z[,3])
```

Computation of the estimated coefficients requires a matrix inversion (or equivalent operation), and this will fail if one predictor is an exact linear combination of the others (or is so within roundoff error). If so, R will remove one of the predictors, and you will see NA listed as the corresponding estimated coefficient.

36.2 Generalized Linear Models

Here we assume that some function of the regression function m(t), rather than m(t) itself, has a linear form.

In R, the function that handles such models is **glm()**. The named argument **family** specifies which function of m(t) has a linear form.

36.2.1 Logistic Regression

Probably the most famous generalized linear model is the *logistic model*, which is applied when Y is a boolean quantity, coded by the values 1 and 0. Here the quantity $\ln[m(t)/(1-m(t))$ is assumed to be equal to $\beta_0 + \beta_1 t_1 + ... + \beta_k t_k$ for some population values β_i. This is equivalent to saying

$$m(t) = P(Y = 1|X = t) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 t_1 + ... + \beta_r t_r)}} \tag{3}$$

This is an intuitively appealing model, as it produces values in (0,1), appropriate for modeling a probability function, and is a monotonic function in the linear form.

One calls **glm()** to fit a generalized linear model. In the logistic case, we specify the argument **family=binomial**, e.g.

```
glm(x[,1] ~ x[,2], family=binomial)
```

36.2.2 The Log-Linear Model

Consider the example **ct** in Section 9, and in particular the data frame of counts **ctdf**. We can perform a log-linear model analysis then we can use **glm()** to regress the counts (**ctdf$Freq** against the factors **ctdf$VoteX** and **ctdf$VoteLast**, setting **family=poisson**:

```
> g <- glm(Freq ˜ Vote.for.X * Vote.Last.Time, family=poisson, data=ctdf)
```

Here we have chosen to fit the saturated model.

36.2.3 The Return Value of glm()

The function **glm()** returns an object of type **glm**, a subclass of **lm**. The latter provides us with a number of member variables and functions, such as discussed in Sec. 36.3.

In addition, there are various **glm()**-specific variables and functions, such as:

- **iter** is the number of iterations used
- **converged** is a logical variable indicating whether convergence occurred

36.3 Some Generic Functions for All Linear Models

These will work on all the linear models:

- **vcov()** will return the estimated covariance matrix for our estimated coefficients
- **fitted()** will return the fitted values, i.e. the predicted Y values for the X's in our sample
- **coef()** returns the estimated coefficients

37 Principal Components Analysis

R includes two functions for principal components analysis, **princomp()** and **prcomp()**. The latter is preferred, as it has better numerical accuracy, though the former is included for compatibility with S.

In its simplest form, the call is

```
> prc <- prcomp(m)
```

where **m** is a matrix or data frame.

38 Sampling Subsets

38.1 The sample() Function

We can use the **sample()** function to do sampling with or without replacement from a finite set. This is handy in simulations, for example.

38.2 The boot() Function

The *bootstrap* is a resampling method for performing statistical inference in analytically intractable situations. If we have an estimator but no standard error, we can get one by resampling from our sample data many times, calculating the estimator each time, and then taking the standard deviation of all those generated values. You could just use **sample()** for this, but R has a package, **boot**, that automates the procedure for you.

To use the package, you must first load it:

```
> library(boot)
```

Inside that package is a function, **boot()**, that will do the work of bootstrapping.

Suppose for example we have a data array **y** of length 100, from which we wish to estimate a population median, using **y**, and have a standard error as well. We could do the following.

First we must define a function which defines the statistic we wish to compute, which in this case is the median. This function will be called by **boot()** (it is named **statistic()** in the list of **boot()**'s formal parameters). We could define it as follows:

```
> mdn <- function(x,i) {
+    return(median(x[i]))
+ }
```

It may seem to you that all this is doing is to call R's own **median()** function, and thus may wonder why we need to define our own new function. But it is definitely needed, with the key being our second parameter, **i**. When we call **boot()**, the latter will generate a specified number of indices (see below), sampled randomly with replacement from 1,...,100 (recall 100 is our sample size here). R will then **i** to this set of random indices when it calls **mdn()**.

For example R might generate **i** to be the vector [3,22,59,3,14,6,...] Of course, in **boot()**'s call to **mdn()**, the formal parameter **x** is our vector **y** here. So, the expression **x[i]** means **y[c(3,22,59,3,14,6,...)]**, in other words the vector **[y[3],y[22],y[59],y[3],y[14],y[6],...]**—exactly the kind of thing the bootstrap is supposed to do.

To then call **boot()**, we do something like

```
> b <- boot(y,mdn,R=200)
```

This tells R to apply the bootstrap to the data **y**, calculating the statistic **mdn()** on each of 200 resamplings of **y**.

Normally, we would assign the result of **boot()** to an object, as we did with **b** above. Among the components of that object are **b$t**, which is a matrix whose i^{th} row gives the value of the statistic as found on the i^{th} bootstrap resampling, and **b$t0**, which is the value of our statistic on the original data set.

A somewhat more sophisticated example (they can become quite complex) would be that in which our data is stored in a data frame, say a frame **d** consisting of 100 rows of two columns. We might be doing regression of the first column against the second. (Let's assume that both the predictor and response data is random, i.e. this is not a "fixed-X regression" situation.) An index **i** here of [3,22,59,3,14,6,...] could mean that our resampling would give us rows 3, 22, 59 and so on of **d**. We could set our **statistic()** function to

```
dolm <- function(fulldata,i)  {
   bootdata <- fulldata[i,]
   lmout <- lm(bootdata[,1]˜bootdata[,2])
   return(lmout$coef)
```

(I've put in some extra steps for clarity.)

Our call to **boot()** could then be, for instance,

```
> boot(d,dolm,R=500)
```

39 To Learn More

There is a plethora of resources one can drawn upon to learn more about R.

39.1 R's Internal Help Facilities

39.1.1 The help() and example() Functions

For online help, invoke **help()**. For example, to get information on the **seq()** function, type

```
> help(seq)
```

or better,

```
> ?seq
```

Each of the help entries comes with examples. One really nice feature is that the **example()** function will actually run thus examples for you. For instance:

```
?seq> example(seq)

seq> seq(0, 1, length=11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

seq> seq(rnorm(20))
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
seq> seq(1, 9, by = 2) # match
[1] 1 3 5 7 9

seq> seq(1, 9, by = pi)# stay below
[1] 1.000000 4.141593 7.283185

seq> seq(1, 6, by = 3)
[1] 1 4
...
```

Imagine how useful this is for graphics! To get a quick and very nice example, the reader is urged to run the following **RIGHT NOW**:

```
> example(persp)
```

39.1.2 If You Don't Know Quite What You're Looking for

You can use the function **help.search()** to do a "Google"-style search through R's documentation in order to determine which function will play a desired role. For instance, in Section 29 above, we needed a function to generate random variates from multivariate normal distributions. To determine what function, if any, does this, we could type

```
> help.search("multivariate normal")
```

getting a response which contains this excerpt:

```
mvrnorm(MASS)            Simulate from a Multivariate Normal
                         Distribution
```

This tells us that the function **mvrnorm()** will do the job, and it is in the package **MASS**.

You can also go to the place in your R directory tree where the base or other package is stored. For Linux, for instance, that is likely **/usr/lib/R/library/base** or some similar location. The file **CONTENTS** in that directory gives brief descriptions of each entity.

39.2 Help on the Web

39.2.1 General Introductions

- `http://cran.r-project.org/doc/manuals/R-intro.html`, is the R Project's own introduction.
- `http://personality-project.org/r/r.guide.html`, by Prof. Wm. Revelle of the Dept. of Psychology of Northwestern University; especially good for material on multivariate statistics and structural equation modeling.
- `http://www.math.csi.cuny.edu/Statistics/R/simpleR/index.html`: a rough form of John Verzani's book, *simpleR*; nice coverage of various statistical procedures.

- http://zoonek2.free.fr/UNIX/48_R/all.html: A large general reference by Vincent Zoonekynd; really excellent with as wide a statistics coverage as I've seen anywhere.

- http://wwwmaths.anu.edu.au/~johnm/r/usingR.pdf: A draft of John Maindonald's book; he also has scripts, data etc. on his full site http://wwwmaths.anu.edu.au/~johnm/r/.

- http://www2.uwindsor.ca/~hlynka/HlynkaIntroToR.pdf: A nice short first introduction by M. Hlynka of the University of Windsor.

- http://www.math.ilstu.edu/dhkim/Rstuff/Rtutor.html: A general tutorial but with lots of graphics and good examples from real data sets, very nice job, by Prof. Dong-Yun Kim of the Dept. of Math. at Illinois State University.

- http://www.ling.uni-potsdam.de/~vasishth/VasishthFS/vasishthFS.pdf: A draft of an R-simulation based textbook on statistics by Shravan Vasishth.

- http://www.medepi.net/epir/index.html: A set of tutorials by Dr. Tomas Aragon. Though they are aimed at a epidemiologist readership, there is much material here. Chapter 2, *Working with R Data Objects," is definitely readable by general audiences.

- http://cran.stat.ucla.edu/doc/contrib/Robinson-icebreaker.pdf: *icebreakeR*, a general tutorial by Prof. Andrew Robinson, excellent.

39.2.2 Especially for Reference

- http://www.mayin.org/ajayshah/KB/R/index.html: *R by Example*, a quick handy chart on how to do various tasks in R, nicely categorized.

- http://cran.r-project.org/doc/contrib/Short-refcard.pdf: R reference card, 4 pages, very handy.

- http://www.stats.uwo.ca/computing/R/mcleod/default.htm: A.I. McLeod's *R Lexicon*.

39.2.3 Especially for Programmers

- http://zoonek2.free.fr/UNIX/48_R/02.html: The programming section of Zoonekynd's tutorial; includes some material on OOP.

- http://cran.r-project.org/doc/FAQ/R-FAQ.html: R FAQ, mainly aimed at programmers.

- http://bayes.math.montana.edu/Rweb/Rnotes/R.html: Reference manual, by several prominent people in the R/S community.

- http://wiki.r-project.org/rwiki/doku.php?id=tips:tips: Tips on miscellaneous R tasks that may not have immediately obvious solutions.

39.2.4 Especially for Graphics

There are many extensive Web tutorials on this, including:

- http://www.sph.umich.edu/~nichols/biostat_bbag-march2001.pdf: A slide-show tutorial on R graphics, very nice, easy to follow, by M. Nelson of Esperion Therapeutics.
- http://zoonek2.free.fr/UNIX/48_R/02.html: The graphics section of Zoonekynd's tutorial. Lots of stuff here.
- http://www.math.ilstu.edu/dhkim/Rstuff/Rtutor.html: Again by Prof. Dong-Yun Kim of the Dept. of Math. at Illinois State University. Extensive material on use of color.
- http://wwwmaths.anu.edu.au/~johnm/r/usingR.pdf: A draft of John Maindonald's book; he also has scripts, data etc. on his full site http://wwwmaths.anu.edu.au/~johnm/r/. Graphics used heavily throughout, but see especially Chapters 3 and 4, which are devoted to this topic.
- http://www.public.iastate.edu/%7emervyn/stat579/r_class6_f05.pdf. Prof. Marasinghe's section on graphics.
- http://addictedtor.free.fr/graphiques/: The R Graphics Gallery, a collection of graphics examples with the source code.
- http://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html: Web page for the book, *R Graphics*, by Paul Murrell, Chapman and Hall, 2005; Chapters 1, 4 and 5 are available there, plus all the figures from the book and the R code which generated them.
- http://www.biostat.harvard.edu/~carey/CompMeth/StatVis/dem.pdf: By Vince Carey of the Harvard Biostatistics Dept. Lots of pictures, but not much explanation.

39.2.5 For Specific Statistical Topics

- *Practical Regression and Anova using R*, by Julian Faraway (online book!), http://www.cran.r-project.org/doc/contrib/Faraway-PRA.pdf.

39.2.6 Web Search for R Topics

Lots of resources here.

- Various R search engines are listed on the R home page; http://www.r-project.org. Click on Search.
- You can search the R site itself by invoking the function **RSiteSearch()** from within R. It will interact with you via your Web browser.
- I use RSeek, http://www.rseek.org a lot.
- I also use finzi.psych.upenn.edu.

A Installing/Updating R

A.1 Installation

There are precompiled binaries for Windows, Linux and MacOS X at the R home page, `www.r-project.org`. Just click on Download (CRAN).

Or if you have Fedora Linux, just type

```
$ yum install R
```

In Ubuntu Linux, do:

```
$ sudo apt-get install r-base
```

On Linux machines, you can compile the source yourself by using the usual

```
configure
make
make install
```

sequence. If you want to install to a nonstandard directory, say **/a/b/c**, run **configure** as

```
configure --prefix=/a/b/c
```

A.2 Updating

Use **updatepackages()**, either for specific packages, or if no argument is specified, then all R packages.