An Introduction to R

Phil Spector
Statistical Computing Facility
University of California, Berkeley

September 20, 2002

1 Background

The R language is a project designed to create a free, open source language which can be used as a replacement for the Splus language, originally developed as the S language at AT&T Bell Labs, and currently marketed by Insightful Corporation of Seattle, Washington. While R is not 100% compatible with Splus (it is often described as a language "which bears a passing resemblance to S"), many Splus programs will run under R with no alterations. Accordingly, much of the existing documentation for Splus can still be useful under R, and many authors of code for either language are careful to make sure that their code will be suitable for both languages.

2 Strengths and Weaknesses

2.1 Strengths

- free and open source, supported by a strong user community
- highly extensible and flexible
- implementation of modern statistical methods
- moderately flexible graphics with intelligent defaults

2.2 Weaknesses

- slow or impossible with large data sets
- non-standard programming paradigms

3 Basics

R is a highly functional language; virtually everything in R is done through functions. Arguments to functions can be named; these names should correspond to the names given in the help file or the function's definition. You can abbreviate the names of arguments if there are no other named arguments to the function which begin with the abbreviation you've used. If you don't provide a name for the arguments to functions, R will assume a one-to-one correspondence between the arguments in the function's definition and the arguments which you passed to the function.

To store the output of a function into an object, use the special assignment operator `<-`, not an equal sign. For example, to save the value of the mean of a vector `x` in a scalar called `mx`, use

```
mx <- mean(x)
```

This statement is read as "mx gets mean of x". At the risk of a loss of readability of your programs, the `<-` operator can be abbreviated as an underscore (_). This also means that the underscore symbol can not be used as part of a variable name.

If you forget to save the output of a function this way, the answer will be temporarily available in an object with the name `.Last.value`.

Typing the name of any R object, including a function, will display a representation of that object. You can explicitly display an object with the functions `print` or `cat`. R provides online help through the `help` function, or by simply preceding a function's name with a question mark (`?`). If you are running R in an environment that includes a web browser, the statement

```
help.start()
```

will provide a web-based interface to the help system. In addition to plain text and HTML versions of the documentation, R also provides hardcopy documentation through LaTeX help files, available by using the `offline=T` argument to the `help` function. If you'd rather learn about a command by studying some examples, the `example` function will run one or more examples using the function passed as its argument. You should quickly get in the habit of consulting the help or example files while you use R, because many of the functions have a number of useful optional arguments which might not be apparent at first glance.

R supports a number of different data structures, accomodating virtually any type of data. At the simplest level, R supports vectors and matrices. These, however are just one- and two-dimensional examples of the more general concept of an array; R supports arrays of virtually unlimited dimensions. In addition, R implements the more traditional "observations and variables" format of rectangular data sets through objects known as data frames, where character and numeric variables can be freely mixed. Finally, R has a very general data structure known as a list, which can hold virtually any structure of data imaginable.

Besides numbers and character strings, the symbols `T` and `F` (or `TRUE` and `FALSE`) are reserved in R to represent the logical values true and false, respectively.

The objects you create during an R session are temporarily stored in a memory; each time you end an R session, you must decide whether you want to save the objects you've been using. (When you run R non-interactively, you must specify either the `--save` or `--no-save` option at invocation time.) If you choose to save the objects you've created, they are stored in a file called `.RData` in the current working directory. (Note that since this file's name begins with a period, you need to provide the UNIX `ls` command the `-a` option in order to display the file's name in a directory listing.) When you start up an R session, the program always looks in the current working directory to see if an `.RData` file exists (unless the `--no-restore` option is specified), and, if you decide to save your working environment at the end of an R session, an `.RData` file will be created. Thus, to keep data from different projects in separate places, it suffices to simply set a different working directory (with the UNIX `cd` command) for each of your projects.

In addition to the `.RData` file in your working directory, R searches for data and functions in a number of system directories. You can see the names of these directories by using the `search` function. Note that just typing the name `search` will display a print representation of the `search` function; to invoke the function with no arguments you must type

```
search()
```

You can see the contents of any directories in your search path through the `objects` function. With no arguments, it will list all the objects in your working directory. The optional argument `pos` allows you to specify other directories in your search path by providing the index of the directory as provided by the `search` command. The optional argument `pattern` will restrict the listing of objects to those whose names contain the string or regular expression specified by the `pattern` argument. Quoted strings in R can be surrounded by either single (`'`) or double (`"`) quotes. Thus, the command

```
objects(pat="dat")
```

will list those objects in your working directory whose names contain the string `dat`. (Notice that, when using named arguments to a function, you can abbreviate the name of the argument, provided that only one named argument agrees with the abbreviation.)

To use R objects which are stored in an `.RData` file other than the one in the current directory, pass a character string containing the full path name of the file to the `attach` command. In addition, a wide variety of functions are organized into sections known as libraries. To see which libraries are available on a given installation of R, type the command `library()`; once you've found a library of interest, you can make the functions and documentation in the library available with a command of the form `library(libraryname)`. The command `library(help=libraryname)` will list the functions which are available, and the `help` command can then be used in the usual way to get more information about individual functions in the library.

You can enter UNIX commands in an interactive or batch R program by passing a character string containing the command to the the `system` function.

You can interactively debug R functions by passing the unquoted name of the function you wish to debug to the `debug` function, and then running the program of interest. See the help file for the `debug` function for details on how to use the debugger.

4 Specific Tasks

4.1 Entering and Exiting the Program

To start an interactive session with R, type

```
R
```

at the UNIX prompt. On most implementations or R, you can redisplay and edit previous commands using arrow keys, or `emacs`-style keystrokes. Inside of R, the interactive prompt is a single greater-than sign (>).

In addition, if you are familiar with the `emacs` editor, there is a special mode for running R which provides command recall, variable name completion, simplified access to help files and many other features. If it is installed, you can access it through typing `M-x R`, followed by the Return key.

R obeys standard UNIX redirection, so you can execute source files in the usual way, with a command like this one at the UNIX prompt:

```
R < infile >& outfile
```

The R commands to be executed would be in the file `infile`, and output would be sent to `outfile`. The `BATCH` command of R packages this capability in a slightly different fashion. Typing

```
R BATCH infile outfile
```

at the UNIX prompt would have a similar effect as the previous command.

To execute R statements from a file from within an interactive session, you can use the `source` command. Type

```
source("infile")
```

at the R prompt to execute the commands in the file `infile`.

To exit an interactive R session, type `q()`. Note that just typing the `q` without the parentheses will simply display a text representation of the `q` function.

4.2 Reading Data

The `c` function (mnemonic for combine) can be used to read small amounts of data directly into an R object. For example, you could create a vector called `x` containing five numbers by using a statement like the following

```
x <- c(12, 19, 22, 15, 12)
```

To read white-space-separated data into a vector, use the function `scan`. The `sep` argument can be used for separators other than the default of white space. For example, to read the data in the file `datafile` into a vector called x, use

```
x <- scan("datafile")
```

With no filename argument, `scan` reads your input from standard input; terminate the data entry with a blank line. To read character data into a vector, use the `what` argument as follows:

```
chardata <- scan("charfile", what="")
```

Often the data you are reading from a file or entering at the keyboard is a matrix. The function `matrix` can be used to reshape the elements of a vector into a matrix. Since the output of one R function is suitable as input to another R function, the calls to `scan` and `matrix` can be combined. Matrices in R are internally stored by columns, so if your data is arranged by rows (as is usually the case), you must set the `byrows` argument to the matrix function to `T`. Suppose that the file `matfile` contained a 10×5 matrix, stored by rows. The following statement would read the matrix into an R object called `mat`

```
mat <- matrix(scan("matfile"),ncol=5,byrow=T)
```

In addition to the `ncol` argument, there is also an `nrow` argument which could have be used (with a value of 5 in the previous example). As shown in the example, if one or the other of these two arguments is missing, R will figure out the other based on the number of input items it encounters. You can also provide descriptive labels for rows and columns using the `dimnames` function.

If your data has a mix of numeric and character variables, you will probably want to store it in a data frame. To read data from a file directly into a data frame, use the function `read.table`. To use `read.table`, all the variables for a given observation must be on the same line in the file to be read. If the optional argument `headers=T` is given, then the first line of the file is interpreted as a set of names to be used for the variables in the file, otherwise default names of `V1, V2,`, etc. will be used. The function `data.frame` can also be used to create data frames directly from other R objects such as matrices or lists.

It should be mentioned that R does not contain a wide range of functions to handle input data. If your input data is not suitable for `scan` or `read.table`, you may need to consider preprocessing the data with a program like `perl` or `sas` before reading it into R.

4.3 Storing Data Sets

The objects which you create during your R session are stored in memory. Before committing to saving all the objects at the end of your session, can remove unwanted objects using the `rm()` function from inside of R. (See the discussion in Section **??**.)

4.4 Accessing and Creating Variables

The basic tools for accessing the elements of vectors, matrices and data frames are subscripts; in R, subscripts are specified using square brackets (`[ ]`); parentheses are reserved for function calls. You can refer to an entire row or column of a matrix by omitting the subscript for the other dimension. For example, to access the third column of the matrix `x`, you could use the expression `x[,3]`. Keep in mind that if you use a single subscript, R will interpret it as the index into a vector created by stacking all the columns of the matrix together.

If you've assigned row or column names to a matrix with the `dimnames` function, you can also use character strings to access parts of a matrix. (Data frames automatically have row and column names assigned when they are created.) If you used statements like the following to create a matrix:

```
mat <- matrix(c(5,4,2,3,7,8,9,1,6),nrow=3,byrow=T)
dimnames(mat) <- list(NULL,c("X","Y","Z"))
```

then you could refer to the second column of the matrix as either `mat[,2]` or `mat[,"Y"]`. The `dimnames` function accepts a list of length 2 containing the row dimnames and the column dimnames; since we only wished to set dimnames for the columns, the special value `NULL` was used for the row dimnames.

While the above techniques will work for data frames as well as matrices, there is a simpler way to refer to variables by name in a data frame, namely separating the data frame's name from the name of the variable with a dollar sign (`$`). For example, if a data frame called `soil` contained variables called `Ca`, `K` and `pH`, you could access the variable `pH` as `soil$pH`. Note that, like other identifiers in R, variable names in a data frame are case sensitive. Alternatively, you can use the `attach` command to make a data frame part of your search path, and refer to variable names directly. The dollar sign notation can also be used to extract named elements out of an R list.

You can create new objects with the assignment operator (`<-`) mentioned in Section **??**. For example, to create a variable `z` which would be the ratio of two variables `x` and `y`, you could use the statement

```
z <- x / y
```

Virtually all operators and functions in R will operate on entire vectors and matrices in a single call. In the above example, if `x` and `y` were each vectors of length `n`, then `z` would also be a vector of length `n`, containing the ratios of the corresponding elements of `x` and `y`.

4.5 Subsetting Data

A variety of subscripting expressions can be used to extract parts of R matrices and data frames. As mentioned previously, you can specify a subscript for either rows or columns to extract entire rows and columns of a matrix. You can also provide a vector of row or column numbers (or names, if `dimnames` were assigned to the matrix), to extract subsets of a matrix or data frame. You can also provide a vector of row or column numbers to extract subsets of a matrix or data frame. The colon operator (`:`), which generates sequences, is often useful in this regard; it generates a vector of integers, separated by 1, from its first argument to its second argument. (The `seq` function provides additional capabilities for generating sequences.) For example, to extract the first, third and fourth variables for the first 10 observations of a data frame or matrix called `data`, you could use the following expression

```
data[1:10,c(1,3,4)]
```

If `dimnames` were assigned to the matrix, a vector of names can be substituted for the vector of numbers in the example just given. A vector of names can be composed using the `c` function, surrounding the names with double or single quotes, and separating them with commas.

A further useful feature of numeric subscripts in R is that negative subscripts represent all values except those specified in the subscripts. So in the previous example, if we wanted all the columns of `data`, except for the first, third and fourth, we could use the expression

```
data[,-c(1,3,4)]
```

Subscripts with a value of 0 are simply ignored.

Logical subscripts provide a powerful tool for subsetting data in R. When using logical subscripts, they must be the same length as the object which is being subscripted; those elements in the subscripted object corresponding to values of `TRUE` will be extracted. Thus, to select all the rows of the matrix `data` for which the third column is less than 10, the following expression could be used

```
data[data[,3] < 10,]
```

Notice the comma after the logical expression, to indicate that we wish to extract those rows for which the third column of `data` is less than 10.

5 Missing Values

The value NA is used to represent missing values for input in R. An NA can be assigned to a variable directly to create a missing value, but to test for missing values, the function is.na must be used.

R propogates missing values throughout computations, so often computations performed on data containing missing values will result in more missing values. Some of the basic statistical functions (like mean, min, max, etc.) have an argument called na.rm, which, if set to TRUE, will remove the NAs from your data before calculations are performed. In addition, the statistical modeling functions (like aov, glm, loess, etc.) provide an argument called na.action. This argument can be set to a function which will be called to operate on the data before it is processed. One very useful choice for the na.action argument is na.omit. This function (which can be called idependently of the statistical modelling functions) will remove all the rows of a data frame or matrix which contain any missing values.

6 Graphics

R provides two types for functions for graphics: high-level functions, which produce an entire plot with a single call, and low-level functions, which are used to add additional information to existing graphs. Among the available high-level routines are barplot, boxplot, contour, coplot (conditioning plots), hist, pairs (scatterplot matrix), persp (3-dimensional perspective), and plot. Low level routines, which provide finer control over the details of existing plots include abline (drawing regression lines), axis (for custom axes), legend, lines, points, polygon, symbols, and text. If you use the example function with any of the graphical functions, you can have the system pause after each graph by issuing the command

```
par(ask=T)
```

Many aspects of the appearance of graphics can be controlled through graphical parameters, set either by a call to the par function, or by passing the parameters directly to high- or low-level functions.

The help file for the par function provides a complete list of graphics parameters.

When you issue a high-level plotting command, or make a call to the par function, R will automatically open a suitable device driver in order to display your plot if one is not already open. In addition, you can produce output in other formats by explicitly calling an appropriate driver. See the help file Devices for a list of the available drivers on your system.

One graphics parameter which is often useful is mfrow, which determines the arrangement of multiple figures on a page. For example, the R statement

```
par(mfrow=c(3,2))
```

would result in six plots being placed on the page, with three rows of two columns each. The plots would be placed by rows; a similar function, mfcol defines the arrangement, put plots the multiple figures by columns.

To display special symbols (like greek letters or subscripts), the expression function can be used. Run

```
example(plotmath)
```

for a demonstration of this capability.

7 Programming

Most functions and operators in R will operate on entire vectors, and the most efficient programming techniques in R are ones which utilize this approach. R does provide loops for more traditional programming, but they tend to be inefficient, especially for large problems. The for loop is a basic tool which can be used

for repetitive processing. It takes the form

```
for(name in values) expression
```

where *name* is a variable which will be set equal to each element of *values* inside of *expression*. If *expression* contains more than one statement, the statements must be surrounded by curly braces ({ }). The for loop works for lists as well as vectors.

Logical subscripts, introduced in Section **??**, can be used on the left hand side of an assignment statement to avoid the use of loops in many cases. A simple, but useful, example is replacing occurences of a particular value with missing values (NA). For example, a double loop could be used to replace all occurences of 9 in a matrix with NA:

```
for(i in 1:nrow(x))for(j in 1:ncol(x))if(x[i,j] == 9)x[i,j] <- NA
```

However, using logical subscripts, the following single statement is much simpler and far more efficient:

```
x[x == 9] <- NA
```

Many functions accept vectors as arguments, so loops can often be avoided by using a vector argument. For example, suppose we wish to form a vector with each of the numbers from 1 to 5 repeated 6 times. One approach would be to use a for loop:

```
result <- NULL
for(i in 1:5)result <- c(result,rep(i,6))
```

However, the same result can be acheived by using vector arguments:

```
result <- rep(1:5,rep(6,5))
```

It is worth noting that this gives a very different value from rep(1:5,6):

```
> rep(1:5,6)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

When processing each row or column of a matrix or data frame, the function apply can be used to eliminate the need for loops. For example, to calculate the sum of each row of a matrix called mat, the following call to apply could be used

```
rowsums <- apply(mat,1,sum)
```

The 1 in the apply call refers to processing by rows; a 2 results in processing by columns. The third argument to apply, in this case, sum, is a function which will be applied to each row or column of the matrix in turn. A similar function, tapply, will apply a function to different subsets of a vector based on the value of a second vector. For example, suppose the vector score represents tests scores in an experiment, and the vector group indicates which group each of the observations came from. To calculate the means for each group, tapply could be called as follows:

```
group.means <- tapply(score,group,mean)
```

If the function being passed to apply or tapply requires more than one argument, the additional arguments can be included in the argument list after the name of the function being passed.

For functions like apply and tapply, it is often useful to construct a function to be applied to each row or column of a matrix, if the function you need is not otherwise available. Often a simple function is all that is needed, and you can pass the function definition to apply or tapply. For example, suppose we have a 2 column matrix called meanvar whose rows contain a mean and corresponding variance, and we wish to generate a vector of random numbers from normal distributions with these means and variances. We could use apply by writing a function which calls the R function rnorm with appropriate arguments as follows:

```
rvars <- apply(meanvar,1,function(x)rnorm(1,x[1],x[2]))
```

The variable `x` in the function definition is a dummy variable which will take as its value each row of the matrix in turn. The returned value, `rvars`, will be a vector with as many elements as `meanvar`, and the `i`*th* element of `rvars` will be a random number from a normal distribution with mean equal to `meanvars[i,1]` and variance equal to `meanvars[i,2]`.

8 Writing Functions

Writing functions is easy in R, because any statement which you use interactively can be used in a function. Furthermore, you can access any variable which is available in the calling environment within a function, but when you change the value of a variable, R automatically creates a local copy, preserving the value of the variable in the calling environment.

Simple functions can be defined interactively:

```
sqr <- function(x) x * x
```

Functions are defined using the `function` statement. Note that, like all other objects in R, the result of defining a function must be assigned to a value, in this case the variable `sqr`. Having defined this function, you can now use the function `sqr` anywhere that any other program could be used. If the body of your function is longer than one line, it must be surrounded by curly braces ({ }).

Although R has an explicit `return` statement, it is not necessary in many cases, because, in the absence of a `return` statement, R functions return the last expression which was evaluated in the function body. Notice that you can not change the value of arguments passed to a function; once you try to do so, R will create a local copy of the argument which will be deleted when control returns to the calling program. If you need to pass back more than one object from a function, you should package the objects you wish to return in a vector, matrix, list or data frame, and return that object.

For functions longer than one or two lines, it is probably best to use an editor to create and modify your functions. R provides a number of functions which will open an editor, allow you to create or modify a function, and, upon closing the editor, return the value of the function. When you use these functions, it's important to store the returned value in a variable for later use. Among the editing functions available are `vi`, `emacs`, `pico`, and `xedit`. The R function `fix` allows you to edit a function, and automatically stores it when you exit the editor. One final option is the use the ESS package in emacs (see Section **??**) to manage your R session and to edit your functions.

One convenient feature of functions in R is that they allow you to specify default values for any of the arguments to the function. Suppose you wish to create a function which will generate a random sample from some distribution, and return the mean of that sample. While it would be desirable to vary the number of observations generated, and the function used to generate them, it would also be convenient if, when these values are not specified, the function automatically defaults to some predetermined values, say a sample size of 25, and observations from a normal distribution. By including an equal sign and a default value in the argument list of a function definition, you can set defaults which can easily be overriden. A suitable function definition would be:

```
ranmean <- function(n=25,ranfun=rnorm)mean(ranfun(n))
```

Now a call to `ranmean()` will use the default values, while a call to `ranmean(100)` would generate the mean from 100 observations, and `ranmean(ranfun=rexp)` would use a sample of size 25 from the exponential distribution.

9 Dynamic Loading of C Programs

9.1 Data Representation

R allows you to call external programs written either in C or in FORTRAN. In either case, the program should not be written as a main program, but as a subroutine (in FORTRAN) or a function (in C) which R will be able to dynamically load into its already executing image. The value returned by a function is **not** available to R; all information between R and the external program must be passed through the arguments of the subroutine or function. To illustrate how to make external programs usable inside of R, a simple example will be used. Before writing the function, it is important to understand the representations of data within R and the correspondence to the data types in C and FORTRAN. These relationships are summarized in the following table:

R	C	FORTRAN
`"single"`	`float*`	`real`
`"double"`	`double*`	`double precision`
`"integer"`	`long*`	`integer`
`"character"`	`char**`	`character`
`"complex"`	`struct { double re,im;}*`	`double complex`
`"list"`	`char**`	-

Note that in each case, the **address** of the data object will be passed to and returned from the function, never the actual value itself. While this is the natural approach in FORTRAN (which uses call by address in its parameter lists), C programmers must take extra caution when writing functions to be used in R. This technique is necessary to allow objects altered in the external function to remain changed when they "return" to the R environment.

The first step in writing the function is to write the interface within R. To illustrate, suppose we want to write a function which will call an external program to calculate the cumulative sum of a vector of numbers. (The R function `cumsum()` already takes care of this, but we will use this simple task as an example.) We need only to pass the vector of values to the R function, but we must also pass the length of the vector to our external program. The interface for a C program could be written as follows:

```
function(x)
{
    dyn.load("./cs.so")
    n <- length(as.vector(x))
    .C("csum",
            as.integer(n),
            result = as.double(x))$result
}
```

The following points should be noted:

- The function `dyn.load` dynamically loads the named shared object into the already existing version of R. However, once a module is dynamically loaded, it remains available through the remainder of the R session. Thus, once the function is debugged and working, the `dyn.load` call should be preceded by a check to see if the module was already loaded as follows:

  ```
  if(!is.loaded(symbol.C("csum")))dyn.load("./cs.so")
  ```

- Since the object file stays loaded throughout the R session, it may be necessary to restart R if you need to reload a new version of your program into R.

- Only shared objects are suitable for dynamic loading into R. The details of producing a shared object are presented below.
- The name of the C source file, the resulting object file, and the R function name need not be the same. What is important is that the `dyn.load` function be passed the name of the shared object file, and the `.C()` function be passed the function name which is defined in the C program.
- The data object "x" is coerced to a vector before its length is taken. This allows the function to handle matrices, although it will simply calculate the cumulative sum for all the elements of the matrix, taken in column-by-column order.
- The arguments "n" and "x" are coerced to an integer and a double precision vector, respectively, before being passed to the C routine. This will insure that the R interface converts the data to the appropriate type (if necessary) before calling your C routine.
- The `.C()` function actually returns a list, with each of the list elements containing one of the arguments. Elements in the list can be named (like `result` in the example) and referred to using the usual "`$`" notation. In this example, the result uses the same storage as the input vector x. The R interface insures that data stored in the calling frame will not be damaged using this technique, and it is encouraged.
- Since an R function returns the result of the last statement in the function, the technique outlined above causes the function to return the appropriate value. It should be emphasized that the original (calling frame) vector "`x`" is *not* overwritten; it has the same value after returning from the function as it did before the function was called.

The C function for this example is as follows:

```
void csum(long *n,double *x)

{  long i;

   for(i=1;i<*n;i++)
      x[i] += x[i - 1];

}
```

Note that both the vector x and the length n were passed to the C program as addresses. As mentioned previously, this is a requirement of the R interface.

To compile the function, assuming it is stored in the file `filename.c`, the following UNIX commands would be used:

```
cc -KPIC -c filename.c
cc -G -o filename.so filename.o
```

If you're using the GNU C compiler, `gcc`, these are the appropriate commands:

```
gcc -fPIC -c filename.c
gcc -shared -o filename.so filename.o
```

The `-KPIC` or `fPIC` option is necessary to create an object file that is appropriate for making a shared object suitable for dynamic loading into R; the `-G` or `-shared` option in the second command produces the actual shared object.

9.2 Using External Subroutine Libraries

As a more complex example of a function, suppose we wish to write a random number generator for the Weibull distribution, using the NAG subroutine library routine `g05dpf`. We will model our function after existing random number generators, and name it `nagrweibull`, since there is an existing `rweibull` function in R. The arguments to the function will be the number of random variables generated, and optionally, shape and scale parameters, which will both default to 1.0 if not specified.

For more complex functions, the interface plays a more important role. It is here where defaults are set and parameters should be checked; if illegal or out-of-range data is sent to an external program, the results will be unpredictable, to say the least. Thus there is a great reward for paying attention to these details in the interface, before a problem arises in the external function.

If your object file requires external libraries, you must include them in the final compile which produces the shared object, so that R can find the necessary libraries when the program is run. Thus, if your program needed the NAG libraries, for example, you would use UNIX commands like the following to create an appropriate shared object:

```
cc -KPIC -c prog.c
cc --f77 -G -o prog.so prog.o -lnag
```

By default, the C library, accessed by `-lc`, is always included, so you don't need to specify it explicitly.) If additional object files were needed to create the shared object for dynamic loading, they would simply be compiled using the `-KPIC` option and included on the command line of the final compile command.

The interface for the `nagrweibull` function could be written as follows:

```
function(n, shape = 1, scale = 1)
{
   dyn.load("./rw.so")
   if(shape <= 0)
      stop("Shape parameter must be > 0.")
   if(scale <= 0)
      stop("Scale parameter must be > 0.")

   result <- double(n)

   .C("rweib",
            as.integer(n),
            as.double(shape),
            as.double(scale),
            result = result)$result
}
```

The same considerations of the previous example apply here, especially regarding the check using `is.loaded` once initial testing has shown that the C program is working properly. The function `stop()` is used to print an appropriate error message and terminate execution before too much damage is done. The C program which calls the NAG routine is as follows:

```
void rweib(long *n,double *a,double *b,double *x)

{  double g05dpf_(double*, double*,long*);
   long i,ifail;

   for(i=0;i<*n;i++)
      *(x++) = g05dpf_(a,b,&ifail);
}
```

Notice that, when calling a routine from a FORTRAN library from within a C program, an underscore must be appended to the routine name, thus the routine is referred to as `g05dpf_`. This is true whether or not the function will be called from R. As in the previous example, **all** arguments to the C function must be pointers — likewise, all arguments to the FORTRAN function must be pointers. This is why the address of ifail is passed to the routine instead of ifail itself. Since R does not have access to values returned by functions, the necessary values are loaded into a vector and then passed back through the argument list of the function. To create the necessary shared object file, the following commands could be used, assuming that the program was stored under the filename `rweib.c`:

```
cc -KPIC -c rweib.c
cc --f77 -G -o rw.so -lnag
```

To use gcc, the corresponding commands would be

```
gcc -fPIC -c rweib.c
gcc -shared -fPIC -o rweib.so rweib.c -lF77 -lM77 -lsunmath -lnag -lm
```

Three of the referenced libraries in the second command (`-lF77 -lM77 -lsunmath`) are only necessary when using `gcc` under Solaris.

9.3 Dealing with Matrices

If your C or Fortran function will deal with matrices, recall that R, like Fortran, stores it matrices by columns as a single-dimensioned array. If you don't store matrices that way in a C program, you will have to write one conversion program which takes the matrix arguments from R and converts them to the form that you expect in your C program, and another which takes your results from the C program and converts them to column-by-column format.

An additional complication when working with matrices is that, by default, the matrix character of an object is lost in the C or Fortran interface when it returns to the R environment. To make sure this doesn't happen, you can use the `storage.mode` function of R to permanently change the storage mode of the object, rather than temporarily coercing it into the appropriate type (with `as.double()` or `as.integer()` for example). For example, suppose we have a C function called `matfunc` which accepts a pointer to an $n \times p$ matrix of doubles, and two pointers to integer representing the dimensions n and p. If we call such a function with an R statement like:

```
newx <- .C("matfunc",x=as.double(x),as.integer(n),as.integer(p))$x
```

we would find that, upon returning from the `.C()` function, `newx` would be a vector of length $n * p$, and not a matrix, as it was when it was passed to the `.C()` function. The solution is as follows:

```
storage.mode(x) <- "double"
newx <- .C("matfunc",x=x,as.integer(n),as.integer(p))$x
```

Notice that when you set the storage mode in this way you should not call the `as.double` function for the matrix in question.

An alternative solution would be to reconstruct the matrix after the call to `.C()`; since the arrangement of the elements doesn't change, all that is required is a call to the `matrix()` function:

```
newx <- .C("matfunc",x=as.double(x),as.integer(n),as.integer(p))$x
newx <- matrix(newx,n,p)
```

10 Statistical Functions

10.1 Descriptive Statistics

For descriptive statistics, R has individual functions for most common statistics. Interestingly, there is no built-in standard deviation function; you need to use the square root of the variance. The `summary` function, when used on a numeric vector, will provide the minimum, the maximum, and the first, second and third quartiles. The function `stem` provides a stemleaf diagram, along with a few descriptive statistics. For categorical variables, the `table` function can provide both one-way and multi-way contingency tables.

10.2 Statisical Modeling

R provides a number of functions for statistical modeling, including `lm` (linear models), `aov` (analysis of variance) and `glm` (generalized linear models). Other modeling routines may be available through libraries, for example `coxph` (Cox proportional hazard models from library `survival5`), `tree` (Recursive Tree models for classification or regression from library `tree`) and `nls` (Non-linear regression from library `nls`). For more information, use either the "Search" or "Keywords" capabilities of `help.start()`, as well as the R Homepage to keep up to date with new libraries as they become available.

For most statistical modeling applications, there is a clear distinction between variables which enter the model as *factors* (discrete categorical variables), and *regressors* (continuous numeric variables). For example, in an analysis of variance, regressor variables are entered directly into the design matrix, while factor variables are entered as one or more columns of dummy variables.

For variables you have identified as factors, R will automatically generate appropriate dummy variables, and most of the functions which display the results of the analysis will treat these groups of dummy variables as a single effect. To let R know a variable is a factor, use the function `factor`. For example, to change a variable called `group` to a factor, use

```
group <- factor(group)
```

If the variable group were stored in a data frame called `mydata`, a similar call would be used:

```
mydata$group <- factor(mydata$group)
```

Using a data frame for statistical modeling is especially easy, because all the modeling functions accept an argument called `data`; when a data frame is given as a `data` argument, R will resolve variable names in that data frame, making your formulas more readable, and eliminating the need to repeat the data frame name over and over.

To construct a formula in R, you use the tilda (`~`), with your dependent variable on the left-hand side and your independent variables on the right-hand side. You can also construct other terms for the right hand side using the symbols in Table **??** below. Note that if you want any of the model operators in the table to behave in their usual fashion inside a formula, the term including the operator should be passed to the `I()` function.

R uses an object-oriented approach to statistical modeling. That means that each of the modeling procedures produces an object which contains an attribute known as the *class* of the object, and that certain functions will do the "right" thing when the are called with such an object as their argument.

For example, suppose we have a data frame called `corn`, containing variables `Yield`, `Block`, and `Variety`. Since R will automatically treat character variables as factors, it is only necessary to identify those numeric variables which you would like to be treated as factors; suppose `Block` is one such variable, taking on the values 1, 2, 3 or 4. The statement

```
corn$Block  <- factor(corn$Block)
```

will identify the variable `Block` as a factor when it is used in a statistical model. Thus, an anaylsis of variance performed with the following statement:

Table 1: Operators Used in Model Formulas

Operator	Usual Meaning	Meaning in Formula
`+`	addition	Add term
`-`	subtraction	Remove or exclude term
`*`	multiplication	Main effect and interactions
`/`	division	Main effect and nesting
`:`	sequence	Interaction
`^`	exponentiation	Limit depth of interactions
`%in%`	none	Nesting

```
corn.aov <- aov(Yield ~ Block*Variety, data=corn)
```

would correctly assign 3 degrees of freedom to the main effect for `Block`, instead of the single degree of freedom which would result if the variable was treated as a regressor. Functions like `print`, `summary`, `anova`, and `plot` would then all provide meaningful output when passed the `corn.aov` object.

10.3 Multivariate Techniques

Multivariate techniques are available in R through a several libraries, including `mva`, which provides cluster analysis, canonical correlation and principal components analysis, and `MASS`, which provides discriminant analysis and multidimensional scaling. As always, the `help.start()` command is a good starting place for tracking down the routines you need, and the R Homepage should be consulted to see if what you're looking for is available in a recently released library.

11 Resources

11.0.1 The R Homepage

You can learn more about R, and download source code or binaries, as well as a wide variety of user-written libraries at `http://www.R-project.org`

11.0.2 Newsgroup Archives:

R news archive: `http://www.ens.gu.edu.au/robertk/R`
Splus news archive: `http://lib.stat.cmu.edu/s-news`

11.0.3 Emacs Resources

Information about the ESS project (Emacs Speaks Statistics) can be found at `http://ess.stat.wisc.edu/ESS-5.0-REA`

11.0.4 Books:

While these books have been written for S or Splus, they provide useful information for users of R. Venables and Ripley's books, in particular, provide specific information about R.

1. *The New S Language: A Programming Environment for Data Analysis and Graphics*, by R.A. Becker, J.M. Chambers and A.R. Wilks, Wadsworth & Brooks/Cole, Pacific Grove, 1988.

2. *Statistical Models in S*, by J.M. Chambers and T.J. Hastie (eds.), Wadsworth & Brooks/Cole, Pacific Grove, 1991.

3. *An Introduction to S and S-Plus*, by P. Spector, Duxbury Press, 1994.

4. *Modern Applied Statistics with S-Plus. Third Edition*, by W.N. Venables and B.D. Ripley, Springer-Verlag, 1999.

5. *S Programming*, by W.N. Venables and B.D. Ripley, Springer-Verlag, 2000.

6. Scripts and complements for Venables and Ripley's books (with specific information about R) are available online at `http://www.stats.ox.ac.uk/pub/MASS3/Sprog`.

7. Documentation is also available at the R Homepage referenced above.