NOTES ON R FOR STOCHASTIC SIMULATION AND ELEMENTARY STATISTICAL INFERENCE*

Daniel Goodman
Bozeman, MT 59717

October 27, 2011

Contents

* Developed for graduate course sequence

Copyright © 2011 Daniel Goodman

1 OBJECTIVE OF THIS MANUAL

R is many things to many people. It can be used as an interactive calculator notepad. It is widely used as a platform to access packages for carrying out statistical analysis. The intention of this manual is a bit different. The intention is to show how **R** can be used as a programming language for purposes of creating user-designed simulations—possibly quite complicated ones—out of a small number of elementary building blocks.

2 MECHANICS OF AN R SESSION

In Windows **R**, invoking **R** will bring up the **R** main window, with a command window called the "R console." This command window shows a prompt ">". Typing in any valid **R** command after that prompt, followed by hitting the "Enter" key, will cause that command to be implemented. This is the way to run **R** "interactively." It is not the way we generally will use **R** to run "programs" of our own creation.

2.1 Making and Using Scripts

We will use a file, called a "Script," which we write with an editor, as a place to store a sequence of commands. This sequence can be quite long and complicated, so we would not want to re-create it anew each time we want to run it. Furthermore, we would like the convenience of incrementally modifying, improving, or correcting it—in part, possibly, by trial and error—or simply storing it for future use. A valid Script file can be "run" at will with a simple command.

Script files are reached from the "File" button on the top left corner of the **R** main window. The pull down menu options which are of interest from the perspective of making and running Scripts are:

New Script opens a blank editing window to create a new Script.

Open Script brings up a window to select an existing Script; opening the selected file brings it into the editing window where it may be modified.

While a Script file is "Open," and the edit window is active (with the cursor active in it), the **Edit** button on the main **R** window pulls down a menu that allows

Run all to run the entire script in the command window, echoing each line of command in the command window as it is executed, and with output (results that the Script has intructions to display to screen) writing to the command window; this mimics what would have happened if the lines in the Script file had all been typed into the command window, in order.

Run line or selection to run just the lines that have been high-lighted with the mouse in the edit window; this may be useful for testing and de-bugging, but interpretation of the results can be tricky, because lines in isolation may not accomplish the same thing that they did in context of the entire Script, and

also because some lines when run in isolation may dip into the "workspace" to get contents from a previous run if the selected lines are not sufficient to define the contents of a variable (in other words this could get the calculations in a garbled sequence).

Save saves the changes made on a Script file while it was open in the editing window.

Save as saves the changes made on a Script file while it was open in the editing window, giving it any name and path specified from the Save window; this is the way to name and store a newly created **R** Script; it should be given extension ".R"

Source R code... brings up a window to select an existing Script; "opening" the selected file from this window will cause it to "run" as if the entire sequence of commands had been entered in the **R** command window (except that the lines of command do not echo to the command window as they echo, and the text echoes to the screen from a line in the script that names the variable do not take place; other kinds of commands to write to the screen or to a file will work as expected in running a Script, as will commands to read from a file); in order to run, the Script file must have extension ".R" and of course the commands in it must be valid. The calculations stored in "workspace" by running a Script from the "Source" option may be viewed subsequently by issuing commands in the command window to display them. (Running a Script from the "Source" option does not open it to editing; this keeps a fully finished Script out of harms way.)

2.2 The Workspace Option for Saving and Examining Results

Whenever a session "runs" anything (such as one or more Scripts, or commands entered from the **R** command window) it stores the concluding status of every named quantity in the "workspace." On conclusion of the session, this memory dump can be saved to a file by using the "File" button pull down menu item **Save workspace image**, which leads to a window where you can name the file and specify the directory where it will be written. The name of the file is up to you; the file extension should be ".RData"

Then, in a future session, you can call up these results with the "File" button pull down menu item **Load workspace image**, which leads to a window where you can name (select) the file you want, and the "Open" button on that window reads the file contents into active memory. Unlike the Script file, which can be viewed or edited with any text editor or work processor, the workspace image file has a special formatting and can be read, or revised, conveniently only by **R**.

When the workspace image from a session is in active memory (whether as a result of loading it from file, or as a result of running something during the session), all the contents of these named quantities are then accessible for use by further interactive commands from the **R** command window, or by running some Script which uses some of those named quantities. When the new session begins, all the named quantities will inititally have the numeric values that were obtained from the previous run(s) in that session or from the loaded workspace image file.

Having this information in active memory creates an opportunity for examining numeric results, or graphic results at leisure, without the need to re-run the Script or to repeat the session. It also creates an opportunity to use those "outputs" from a previous session as "inputs" in a new session.

Numeric contents of variables in the workspace can be examined by simply typing the name of the quantity in a console command line, or with the *print* command which works from the console or in a Script. Graphics can be created from stored numeric values by typing the graphics command to operate on the named quantity. The resulting displays can be grabbed from the screen, or copied to the Windows clipboard, and then pasted into whatever Windows product you wish for writing your report, or diary, of the work.

The fact that workspace memory can contain numeric values in named quantities as a result of activities from earlier in a session also can be a programming liability. If you then issue commands or run a Script calling on these quantities without initializing them properly, **R** will get their old values out of workspace memory, and proceed. If this is unintended, you will almost certainly get incorrect results. One way to guard against this inadvertent use of "left over" values is to start a new session (quit and then reopen R). Another way is with an explicit clear workspace command, to obtain a clean slate before starting new work. One way to clear the workspace is from the pull down menu obtained from the "Misc" button on the **R** main window, and then selecting **Remove all objects**.

2.3 Program Testing

R recognizes an astronomical number of functions and operations, and it is extremely flexible in accepting variations in syntax and in supplying defaults when something is not specified explicitly. As a consequence, many editing typos, and outright errors in coding logic, still will run and generate "something" as output. Therefore, the burden is on the programmer to verify that a program (Script) actually accomplishes the intended task correctly.

Often, such testing is done by applying the program to a "simple" job with only one or two simulation trials, one or two data observations, and one or two variables, and possibly with special-case inputs such as 0 variability. Then you can see whether the form of the answer is as expected; and numerical results can be compared against direct calculations. Or you can run an example with a known text book solution. Other tests may be carried out by graphing, or displaying numerically, intermediate quantities. And for some statistical simulation programs, the code can be tested by using a very large sample size and/or a large number of trials to check whether the results then are as expected when the randomness cancels out.

2.3.1 Flexibility and portability of Scripts

For a very complicated Script where a large investment has been made in testing, debugging, and validation, it may pay to make that Script "self-contained" so that it is not routinely opened to further editing which might inadvertently introduce new errors. In order to provide flexibility to use that unchanging block of code for a variety of future jobs, you would leave the changing control parameters (such as identity of data file, data themselves, number of trials, etc.) unspecified in the Script. Then you can set these controls interactively from the

command window before running the Script, and when you run the Script it will obtain the needed values (provided they were given the right names) from the workspace. As a check, to make sure that the Script did get the values you intended, the Script itself should echo those values.

3 COMMANDS

The remaining sections of this document will deal with commands, and structures of several commands in sequence, that are useful in a Script for carrying out simulations that illustrate fundamental statistical tests and statistical estimates. These commands can be tried out interactively from the **R** command window. Further options for their use can be learned by typing "help(thecommandname)" in the **R** command window. But expect that the main use of the command, in this course, will be as a building block in a Script that accomplishes some larger task.

3.1 Referencing Files in a Command

Commands that reference a file will expect the file name and path to be enclosed in double quotes. The path specification follows the Unix, rather than the Windows, convention of using the forward slash rather than the backslash. Thus, for example:

```
"c:/dira/subdirb/filename.ext"
```

references a file named *filename*, with extension *ext*, in subdirectory*subdirb*, in directory *dira*, off the root of drive *c*. On a Windows machine, the path and file specification is not case sensitive.

3.2 Commands to Control a Session

There are commands, which may be entered directly from the console, or which may be lines in a Script, which accomplish the same things as some of the operations described above as being invoked by clicking on a button in a pull down menu from the R Gui. For example

```
source("filename.R")
```

will run the named Script.

4 LINES, CONTINUATION LINES, AND COMMENTS

Generally, each line of **R** code in a Script is interpreted as a command. Multiple commands may be put in a single line if they are separated by ";" but usually there is no good reason

for doing this, and it creates a cluttered-looking code file which is harder to understand by visual inspection.

When entering commands interactively in the **R**-console, hitting the "Enter" key on an incomplete command (uninterpretable by **R**) results in a "+" prompt, which allows you to complete the command. If you "intended" to continue the line, but the command as it stood was interpretable, it is too late once you hit the "Enter" key. The use of parentheses can insure incompleteness, since a command is not complete until the number of left and right parentheses balance. Of course, the parentheses will still have their usual function in arithmetic, so they must be placed in a way that is still consistent with the intended operation. Similarly, an operation may be treated as a "group" of one command (see section [15]) with curly brackets, which then allows a line break, as long as the line break does not break a variable name or function name or operator.

This logic for continuation lines carries over in Scripts, though the prompts are not present in the script itself. If a command is too long to fit on one line, it may be "continued" onto subsequent lines if each continued line is recognizably incomplete (uninterpretable by **R**). The use of parentheses, or curly brackets, can insure such incompleteness, by placing a left parentheses or bracket in the first line that is "to be continued" and not closing that with the paired right-character until the last of the continuation lines.

Everything to the right of a "#" sign in a line of **R** code is ignored when the line runs. This is the way to put comments to yourself in a Script, to help remind you what the various lines of code do. This is a good idea.

Multiple blanks (horizontal space) are interpreted the same as a single blank. Leading and trailing blanks are ignored. Blanks within expressions (e.g. between operators or named quantities) are ignored. This allows considerable freedom to use spaces to create a layout that facilitates visual comprehension of the code.

Unless program "flow" commands are involved (specifying things like loops or branching), a program "runs" (implements) the lines of code in the Script in the sequence corresponding to the order of the lines. The program "remembers" the results of previous calculations until they get overwritten.

5 VARIABLE NAMES AND THE ASSIGNMENT OPERATION

A variable name is something begining with a letter of the alphabet that serves as a "place" where numeric values can be stored. The numeric value may be changed by operations during the course of running the program. The changes may be brought about by arithmetic expressions or by defined functions.

R is case sensitive. The variable X and the variable x are different, and may coexist with distinct values in the same program, or even the same line of code. Case also matters in all the built-in functions.

The fundamental operator for moving a value into a named variable is the "assignment" indicated symbolically by "<-" a "left arrow" made from typing the angle bracket character

"<" followed by the hyphen character "-" from the keyboard. The elementary syntax of an assign command is:

```
AA <- BB
```

which takes the current numeric contents of the variable BB and stores it in AA, overwriting the previous numeric contents of AA. The operation is directional: whatever is on the right gets evaluated, and the result gets stored on the left.

The actual numbers involved depends on context from one line to the next. Thus,

```
BB <- 35.5
AA <- BB
```

will store the number 35.5 in variable name AA; whereas

```
BB <- 21
AA <- BB
```

will store the number 35.5 in variable name AA

When issuing commands interactively in the command window, or when "running" a Script from the Edit window, a line that issues a command to calculate something without specifying an assignment of the result, carries out the calculation and automatically echoes the numeric result (writes it) to the screen; similarly when issuing commands interactively in the command window, or when "running" a Script from the Edit window, a line that simply names a variable without specifying an operation or an assignment automatically echoes the current numeric contents of that variable to the screen. A line with an assignment does not automatically echo to the screen.

These automatic echoes do not occur when running a Script using the "Source R code..." option rather than from the editor. There are other kinds of write (or read) commands which will work both in interactive mode and from running a Script by "Source R code..."

A variable which has been created, but does not yet contain a numeric value, will display "NA" as its contents. A variable which contains the results of an arithmetically undefined operation such as "0/0" will display "NaN" as its contents. A variable which contains results of an arithmetic operation corresponding to infinity will display "Inf" as its contents.

5.1 Special Constants

Some special mathematical constants are already named

```
x <- pi    # loads the value 3.1415..... into x
x <- i     # loads the imaginary square root of -1 into x
```

These special named constants can be over-ridden by operations in a session or Script. That is to say, the variable named "pi" or the variable named "i" can be given values other than their pre-assigned defaults.

5.2 "Objects"

Named variables can be of several different kinds, and can store more complicated information than just a single numeric value. These named things, broadly, are termed "objects." **R** actually keeps track of what kind of thing an object is, as well as keeping track of its numeric contents. The kinds of objects which we will use most are vectors (a single numeric value is the special case of a vector with just one element) and matrices.

We will make limited use of a kind of object called a "list." Our main concern with lists generated by particular functions will be to unpack them to obtain the numeric components which interest us.

When using **R** as a statistical package, a kind of object called a "data frame" plays a big role. Essentially, a data frame is a matrix (table) of numerical values, where the columns are distinct "observed data variables," each row constitutes a multivariate "observation," and each column also has a "name" by which it can be selected for analysis or identified in output.

In our simulation programming, we will use a data frame only when it is necessary as the input to a statistical function.

6 COMMANDS TO EXAMINE OR MANAGE WORKSPACE CONTENTS

The command

```
ls()
```

will list the names of all the objects in the current workspace.

The command

```
ls.str()
```

will list the names of all the variables in the current workspace and will print out their contents (i.e., numeric values for all the variables which store numbers).

The command

```
rm(x)
```

where x is the name of a variable in the current workspace, will remove that variable from the workspace.

The command

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

will clear the entire workspace.

7 ELEMENTARY ARITHMETIC OPERATORS

The following symbols function on scalar (non-vector) variables the way you might expect in "normal" arithmetic.

+ addition
- subtraction
* multiplication
/ division of value on left by value on right
^ exponentiation (base on left, exponent on right)

The default order of operation is exponentiation first, multiplication and division next, addition and subtraction last. The default order can be over-ridden by parentheses. Parentheses force their contents to be evaluated first; when parentheses are nested, the innermost is first.

Example:

```
A <- (BB+c)^3    # the quantity (BB+c) is raised to power 3; then
                 #  the numerical value of the result is stored in A
```

8 BUILT IN FUNCTIONS

Further standard operations corresponding to elementary calculation, but requiring parentheses to indicate what is operated on, are:

sqrt() square root
exp() e (base of the natural logarithms) raised to the power
log() natural logarithm
log10() logarithm base 10
sin() sine (angle in radians)
cos() cosine (angle in radians)
tan() tangent (angle in radians)
factorial() factorial
round() round to the nearest integer

Some functions require more than one number to be operated on. An example is:

choose(N,n) the combinatoric expression N!/((N-n)!n!)

9 RANDOM NUMBER GENERATORS

For reproducibility, and control, the seed for the random number generators should be set explicitly, before the first use of a random number generator in the Script. The command is:

```
set.seed(m)   # set the seed to the value of m
```

Changing the seed and re-running the Script will obtain a different ("independent") sample from the random number generators.

There is a long list of random number generators for sampling from various distributions. Most of these have 3 arguments—values that should be specified to control the operation. Generally, the first argument is the number of random values to be drawn from the distribution, and the next two arguments are the further specifications of characteristics of the distribution—for example its mean and variance— with different characteristics (parameters) used for different distributions. Some of the random number generators, and their argument lists, are

rbeta(n,a1,a2) sample of n values from a beta distribution with parameters a1 and a2
rbinom(n,N,p) sample of n values from a binomial distribution with N trials and probability of event p
rexp(n,r) sample of n values from an exponential distribution with rate parameter r
rgamma(n,sh,sc) sample of n values from a gamma distribution with shape parameter sh and scale parameter sc
rlnorm(n,lu,ls) sample of n values from a lognormal distribution with log space mean lu and log space standard deviation ls
rnbinom(n,m,p) sample of n values from a negative binomial distribution with number of events m and probability of event p
rnorm(n,u,s) sample of n values from a normal distribution with mean u and standard deviation s
rpois(n,r) sample of n values from a Poisson distribution with rate parameter r
runif(n,ulo,uhi) sample of n values from a uniform distribution between ulo and uhi

We will describe in a moment how this sample can be stored for use by the program.

10 VECTORS AND SCALARS

Unless specified otherwise (we will get to this later), a named variable in **R** is automatically a vector—a list of numbers, not necessarily a single number. The notation used up till this point in this document ignored this, without any problems arising, because each named variable could be interpreted as a single-element vector, and everything would be fine.

But in other contexts, the variable definitely will function as a vector with many elements. For example:

```
x <- c(5.7,21.0.3.4) # puts the 3  "concatenated" specified values
                     #  into the vector named x
x <- rnorm(n,u,s)    # samples n values from a normal distribution with
                     #  mean u and standard deviation s, and stores the
                     #  values in the vector named x
x <- rep(-99,100)    # stores the value -99 as the 100 elements
```

```
                      #  of the vector x
x <- seq(1,50)        # stores the sequence of integers 1 through 50
                      #  in the vector x
x <- seq(1,50,2)      # stores the sequence of integers 1,3,5,...,47,49
                      #  in the vector x of 25 elements (stops at 49
                      #  because 49+2=51 exceeds the upper limit)
```

All five above examples , referring to the destination vector by name, without specifying particular elements of that vector, "create" the vector, giving it the "length" (number of elements) required. Under these conditions, if the vector had been defined with a different number of elements earlier in the program, the number of elements will be changed–in other words, this kind of operation "replaces" the previous contents and "length" of the pre-exsting vector.

Generally, a vector can be created by an operation (such as each of the five examples above) without explicitly declaring it to be a vector. **R** will figure out automatically that the "object" being created has to be a vector.

10.1 Declaring a Vector

A bare bones declaration of a vector can be done with the command

```
x <- vector(mode="numeric",length=10) # declares x to be a vector of 10
                                      #  elements for numeric values
                                      #  (this will automatically
                                      #  initialize all elements with the
                                      #  value 0)
```

or a vector may be declared as a 1-row matrix or a 1-column matrix (see section [12] below).

10.2 Referring to Specified Elements of a Vector

The syntax

```
x[3]
```

refers to the 3'rd element of the vector. The syntax

```
x[3:7]
```

refers to the 3'rd through 7'th element of the vector (i.e., a subsegment).

If this subset designation appears on the "source" side of an assignment ("arrow") operation, the numerical values in the specified elements will be accessed and used in the operation. If this subset designation appears on the destination side of an assignment operation, the operation will operate on (and overwrite) those elements, but the other elements are not affected. Referring to a specified element of a vector requires that the vector already have been declared.

```
x[j] <- 3   # load the value 3 into element j of vector x (requires
            #  that x already be established as a vector, and that j
            #  already have a numerical value)
```

10.3 "Scalar" Arithmetic Syntax on Vectors

Operations on vectors with commands using "normal" non-vector arithmetic syntax carries out the specified operation one element at a time on every element of the vector. For example, if Y already has been initialized as a vector with n elements, then

```
z <- 1/Y  # stores in vector z the element by element reciprocals of Y
```

or

```
w <- factorial(Y)  # stores in vector w the element by element values
                   #  of the factorials of elements of Y
```

This takes getting used to. It does *not* correspond to "normal" math notation.

A further oddity which must be understood is that **R** will proceed with such an operation even if there is a missmatch in the number of elements of the vectors involved. **R** will "pad" the short vector, by wrapping elements from the beginning of that vector, to get a number of elements sufficient to carry out the operation using all the elements of the longest vector.

So, for two vectors W and S,

```
z <- W*S # stores in vector z the element by element products of W and S
```

First, we note that this does not correspond to the usual rules of vector multiplication in matrix algebra. (There are other **R** commands for such multiplication.) Second, we note that this operation presumes the same number of elements in each of the vectors z, W, and S. If they don't have the same number of elements, the padding and wrapping rules take over. Generally, it is up to you to keep track of the number of elements in the vectors in your program, to ensure that no unintended padding or wrapping takes place.

10.4 Functions to Summarize a Vector

Functions designed to operate on a vector may produce just one number as a result, or may deliver a list of numbers (vector) with a different length from the vector that is summarized. Examples of a single number resulting are:

```
f <- mean(b)    # calculate mean of values in vector b, and store in f
f <- median(b)  # find median of values in vector b, and store in f
f <- var(b)     # calculate "population estimate" of variance of values
                #  in vector b, and store in f; to get the sample
                #  standard deviation (treating the values in b as if
                #  they were the entire population) multiply
```

```
                #  f by (n-1)/n where n is the number of values in b
f <- sd(b)      # calculate "population estimate" of standard deviation
                #  of values in b, and store in f; to get the sample
                #  standard deviation (treating the values in b as if
                #  they were the entire population) multiply
                #  f by sqrt((n-1)/n) where n is the number of values in b
f <- min(b)     # find the minimum of the values in vector b, and store in f
f <- max(b)     # find the maximum of the values in vector b, and store in f
f <- length(b)  # find the number of values in vector b, and store
                #  that count in f
f <- sum(b)     # calculate sum of values in vector b, and store in f
f <- prod(b)    # calculates product of values in vector b, and store in f
f <- which.min(b)  # find the minimum of the values in vector b, and store
                   #  in f the index value (element number in b)
f <- which.max(b)  # find the minimum of the values in vector b, and store
                   #  in f the index value (element number in b)
```

An example of a summary resulting in a vector is:

```
v <- quantile(b,p=c(0.05,0.1,0.9,.0.95))
                  # find the 5%, 10%, 90% and 95% left tails of the values
                  #  in vector b, and report these in the (4-element)
                  #  vector v. (The Q% quantile value is the largest number
                  #  in the smallest Q%, by count, of the numbers in b.)
```

A tabular summary of the distinct values in a vector and the number of occurrences of each is obtained from:

```
t <- table(x)  # store in object t a tabulation, where the first line is
               #  a header which will echo the name of the vector, the
               #  second line is a list of all the distinct values
               #  encountered in the vector, ordered by size, smallest
               #  first, and underneath each listed value will be the
               #  number of occurrences (count) of that particular value
```

The table, in its entirety can be displayed by any command to print t.

To unpack the table, t, into the vector of distinct values and the associated vector (same number of elements) of the counts:

```
vn <- as.numeric(names(t))  # load into vector vn the list of all the
               #  distinct values encountered in the tabulation that
               #  created t, ordered by size, smallest first
vc <- as.numeric(t)         # load into vector vc the counts of the
               #  values encountered in the tabulation that created t,
               #  ordered to correspond to the list of the respective
               #  the distinct values, as in vn
```

Graphics, such as a histogram, which will be discussed later, can also summarize a vector.

10.5 Some Other Functions which Operate on a Vector

Some operations on the entire vector which create a new vector of the same length include

```
f <- cumsum(x)    # cumulative sum of elements 1 through j of the vector x,
                  #  loaded into element j of f
f <- cumprod(x)   # running product of elements 1 through j of x, loaded
                  #  into element j of f
```

10.6 Functions which Operate on More than One Vector

Some functions operate on more than one vector at a time to obtain their result. For example:

```
b <- weighted.mean(v,w)  # mean of elements of the vector v weighted by
                         #  the corresponding elements of vector w,
                         #  loaded into the variable b (single value)
```

10.7 Subsetting a Vector by Subscript

Syntax to specify a particular portion of a vector includes

```
f <- b[j]       # load into f the j-th element of vector b
f <- b[j:k]     # load into vector f the j-th through k-th elements of vector b
f <- b[-j]      # load into vector f all elements of vector b except the j'th
```

10.8 Rearranging Elements of a Vector

```
f <- sort(x)    # order the elements of x in f, by size, smallest first
f <- rev(x)     # reverse the order of the elements of x in f
i <- order(b)   # load into vector f the indices (subscripts) of vector b
                #  ordered according to the values of the corresponding
                #  elements of b, smallest first  (but this does not reorder
                #  the elements of b)
w <- b[i]       # where i is a k-element vector of integer values, loads
                #  k values from vector b into vector w, using the integer
                #  values in vector i as indices of the elements selected
                #  from vector b; in other words, if the integer n is
                #  the j'th element of vector i, then b[n] will be loaded
                #  as the j'th element of vector w.
```

10.9 Some Functions which Operate on a Pair of Vectors

Many of these functions are argument dependent, and will also operate on a matrix or pair of matrices, where they will do something else. Here, operating on a pair of vectors, each returns a single value. The two vectors need to have the same number of elements, to get a result with the usual meaning.

```
b <- cov(x,y)   # estimate of population covariance between x and y
b <- cor(x,y)   # correlation between x and y
```

10.10 Sampling Values From a Vector

Samples of specified size may be drawn from the numerical values that are stored in a vector:

```
f <- sample(x,n)  # randomly sample n values from vector x, without
                  #  replacement, and store these in vector f;
                  #  if n is the length of vector x, the result is
                  #  a random reordering of vector x
f <- sample(x,n,replace=T)  # randomly sample n values from vector x,
                            #  with replacement, and store these in vector f
```

11 READING NUMBERS FROM A FILE INTO A VECTOR

The syntax to read blank-separated numeric values from an ASCII ("plain text" as may be exported—"save as"—from Word) file is

```
b <- scan("filenam.typ")   # reads all the numbers from file named filenam.typ,
                           #  and stores them in sequence in vector b,
                           #  expecting blanks and line breaks to separate
                           #  values in the file
```

where the line breaks in the file also separate numeric values. This read will accept exponential notation in the form 3.4e+05 for example, as well as integers and floating point decimal numbers.

The syntax to read comma-separated numeric values from an ASCII file (such as a csv export from Excel) is

```
b <- scan("filenam.typ",sep=",")  # reads all the numbers from file
                                  #  named filenam.typ, and stores them
                                  #  in sequence in vector b,
                                  #  expecting commas and line breaks to
                                  #  separate values in the file
```

There must be a way to specify that both commas and blanks are valid separators in the read, but I haven't figured it out. In the meantime, you may use the search and replace function in your editor to get consistent separators in the file.

The default path for a read statement is to look for the file to be read in the "active" directory. The path may be specified explicitly, for example

```
b <- scan("c:/projectdata/filenam.typ")
```

Note that the forward slash "/" must be used in place of the normal Windows backslash "\" in path specification inside an **R** command; but the usual Windows convention of backslash holds inside the "file open" selection windows.
Another option for specifying the file and path is the syntax

```
b <- scan(file.choose())
```

which opens a window for interactive selection. If you use this command in a Script, execution will pause at the open window until you respond.

12 MATRICES

A matrix is a doubly subscripted (indexed) array, which is the natural way to store a "table" of numbers. Some operations by their nature create a matrix, and if the result of that operation is assigned to (loaded into) a newly named "object," that object will automatically have the status of a matrix with the required number of rows and columns. In other circumstances, a matrix is "declared," explicitly, before its first use, with a statement that establishes the number of rows and columns and which can also load some numbers into it. The syntax is:

```
M <- matrix(nrow=5,ncol=3)   # declares M to be a matrix of 5 rows
                             #  and 3 columns, but leaves it empty
A <- matrix(x,nrow=n,ncol=m) # declares A to be a matrix of n rows
                             #  and n columns, and loads the first n*m
                             #  elements of vector x into it, columnwise
                             #
B <- matrix(v,nrow=n,ncol=m,byrow=T)    # declares B to be a matrix
                                        #  of n rows and n columns,
                                        #  and loads the first n*m
                                        #  elements of vector v into
                                        #  it, row-wise
```

Because of the padding and wrapping "features" of **R**, it is up to you to make sure that operations involving matrices are carried out on matrices with the appropriate numbers of rows and columns.

The dimensioning of an existing matrix can be revealed with the command:

```
b <- dim(W)        # loads into vector b the number of rows declared for
                   #  matrix W (as element 1 of b) followed by the number
                   #  of columns (as element 2 of b)
```

or the number of rows and columns can be revealed separately by

```
n <- nrow(A)   # load into n the number of rows that matrix A has
m <- ncol(B)   # load into m the number of columns that matrix A has
```

Syntax to specify a particular portion of a matrix includes:

```
z <- y[i,j]                 # loads into z the element from row i column j
                            #  of matrix y
v <- y[i,j:k]               # loads into vector v elements j through k
                            #  of row i of matrix y
```

12.1 "Scalar" Arithmetic Syntax on Matrices

Operations on matrices with commands using "normal" non-matrix arithmetic syntax carry out the specified operation one element at a time on every element of the matrix. For example, if Y already has been initialized as a matrix with n rows and n columns, then

```
z <- 1/Y  # stores in matrix z the element by element reciprocals of Y
```

If an assignment creates some variable which has not already been defined in the session, **R** tries to guess what kind of "object" is appropriate to receive the results of an assignment operation. In the above example, if "z" has not been previously defined, **R** will make it a matrix with the same number of rows and columns as Y. To be safe, you could have declared it to be a matrix in a previous command.

As with "normal looking" vector arithmetic syntax in **R**, the arithmetic operations on matrices in **R** do not always correspond to "normal" math notation. And, as with vector operations, **R** will proceed with such an operation even if there is a missmatch in the number of elements, or rows or columns, by padding and wrapping. (Though, unlike the situation with vectors, the missmatch may generate a "warning" message.) Generally, then, it is up to you to keep track of the number of rows and columns in the matrices in your program, to ensure that no unintended padding or wrapping takes place.

12.2 Functions which Operate in Special Ways on Matrices

```
z <- rowSums(x)   # form sum of each row, and load into vector z
z <- colSums(x)   # form sum of each column, and load into vector z
z <- rowMeans(x)  # form average of each row, and load into vector z
z <- colMeans(x)  # form average of each column, and load into vector z
C <- cov(X)       # form covariance matrix C of the columns of matrix X
                  #  (estimate of population covariance); if X has
                  #  n rows and m columns, the covariance matrix
                  #  will be m by m
C <- cor(X)       # form correlation matrix of the columns of matrix X
```

12.3 Matrix Algebra

R does have commands to carry out the operations of matrix algebra according to their usual mathematical definitions (provided the numbers of rows and columns are appropriate). Notably

```
C <- A %*% B   # apply matrix A (of n rows and m columns) in matrix
               #  multiplication to matrix B (of m rows and k columns)
               #  to obtain matrix C (of n rows and k columns)
C <- t(A)      # transpose of matrix A
d <- det(A)    # determinant of matrix A
C <- solve(A)  # invert matrix A and load into C
L <- eigen(A)  # loads into "list" L the eigenvalues and eigenvectors of
               #  matrix A; the name of L is user-specified; then
               #  L$val is the vector of eigenvalues, and L$vec is
               #  the matrix of eigenvectors (as columns?)
```

The interpretation of an **R** vector (not declared as a matrix) as either a row or column in mixed matrix-vector expressions is context dependent. It will be more clear to explicitly use declared matrices with 1 row or 1 column for such purposes.

The operation of some "matrix functions" can depend on the status of their argument:

```
v <- diag(A)  # for A a matrix, loads its diagonal into vector v
              #  (this would be the usual math interpretation)
A <- diag(v)  # for v a vector, forms diagonal matrix A
I <- diag(k)  # for k a scalar, forms k by k identity matrix in I (weird)
```

13 OPERATIONS ON ROWS OR COLUMNS OF A MATRIX

The "apply" function carries out a named operation on all the rows, or all the columns, of a specified matrix. The result is a vector. There are 3 arguments to an apply command: the first names the matrix to be operated on, the second is an integer 1 or 2 specifying whether the operation is on rows (1) or columns (2), the third defines that operation, which of course should be an operation appropriate for applying to a vector. For example:

```
z <- apply(X,1,sd) # calculate population estimate of standard deviation
                   #  of each row of matrix X, and load into vector z
```

This offers considerable flexibility if you define your own "function" which gets applied to each row. (See section [18]).

13.1 Merging Matrices

To merge compatible matrices row-wise

```
Z <- rbind(X,Y)             # assemble matrix Z with the same number of
                            #  columns as matrices X and Y, where matrix
                            #  makes up the first set of rows of matrix Z,
                            #  and matrix Y makes up the remaining rows
```

To merge compatible matrices column-wise

```
Z <- cbind(X,Y)              # assemble matrix Z with the same number of
                             #  rows as matrices X and Y, where matrix
                             #  makes up the first set of columns of matrix Z,
                             #  and matrix Y makes up the remaining columns
```

The concatenate operator can also be used to merge several matrices into a larger matrix, provided the larger matrix has been declared (otherwise the concatenation will form a vector made by stringing columns of the matrices in sequence). The matrices get concatenated column-wise. This can be done inside a matrix declaration in order to keep control of rows and columns. For example, if X has already been initialized as a matrix of n rows and m columns, and Y has already been initialized as a matrix of n rows and k columns, then:

```
Z <- matrix(c(X,Y),n,(m+k))  # assemble matrix Z as n rows and (m+k) columns
                             #  where the first m columns are matrix X
                             #  and the remaining k columns are matrix Y
```

14 CONVERTING BETWEEN MATRIX AND VECTOR STORAGE

An existing matrix can be "unspooled" as a vector by redimensioning it. The resulting vector will consist of the columns of the previous matrix strung together. The syntax is

```
dim(A) <- nm   # where A was a matrix of n rows and m columns,
               #  and nm is the value n*m, this command reconfigures
               #  A to be a vector, of nm elements, where former
               #  columns of the matrix are strung together
```

The process can be reversed, to repack an existing vector as a matrix:

```
dim(V) <- c(n,m)   # where V was a vector of k elements, and n*m=k,
                   #  this command reconfigures V to be a matrix
                   #  of n rows and m columns, where n-element
                   #  segments of the former vector now make up
                   #  the columns of the matrix
```

15 GROUPED COMMANDS

Blocks of commands may be created by "grouping" several ";"-separated commands inside curly brackets. The curly brackets allow a "group" to be continued over more than one line. The left curly bracket initializes a group, which then continues until a right curly bracket is encountered, which may be on a subsequent line.

Blocks of this sort are useful when conditional branching, or looping, over-rides the usual line-by-line sequence of implementation of the code in a Script. Then, the branching can direct flow to entire blocks, or the branching can skip entire blocks; or the loop can repeat an entire block.

16 CONDITIONAL COMMANDS

The conditional is indicated with an "if." The syntax is

```
if (a) b  # if the expression in parentheses is "TRUE," then this line
          #  executes b (which may be grouped commands in curly brackets);
          #  if (a) is not "TRUE" the line has no effect
```

Examples of expressions which could have the value "TRUE" are relationships designated by:

```
 x<y    #  x less than y
 x<=y   #  x less than or equal to y
 x>y    #  x greater than y
 x>=y   #  x greater than or equal to y
 x==y   #  x equal to y
 x!=y   #  x not equal to y
```

These may be combined with "&" for a logical "and," and with "|" for a logical "or," and "!" for a logical "not." The "&" and "|" logical connectors require that the logical variables being connected have the same dimension. If the logical variables are vectors, and the interest is in combining the truth status only of their respective first elements, the logical connectors "&&" and "||" will accomplish that.

Parentheses can establish order of evaluation of the status of the condition. This allows the construction of quite complicated expressions, which overall can still be evaluated as "TRUE" or not.

A more elaborate conditional program flow control commands creates a "bifurcation" in the flow of execution with an "if" that has an "else" alternative. The syntax is:

```
if (a) b  else c  # if the expression in parentheses is "true," then this line
                  #  executes b (which may be grouped commands in curly
                  #  brackets) and resumes on the next command after c;
                  #  if (a) is not "true" then this line executes c (which
                  #  may be grouped commands in curly brackets) and resumes
                  #  on the next command after c
```

There are layout restrictions for breaking a "if else" construction over several lines in a Script. The construction is more than one expression so, while breaks within an expression (such as the conditional or the command that is executed if the condition is "TRUE," or the command that is executed in the condition is not "TRUE") may be accomplished according to the rules for continuation lines, breaks after the "if" or after the "else" are treated differently.

Such an architectural line break may be inserted:

right after the "if";
right after the right parenthesis of the conditional expression;
right after the "else";

but the "else" may not begin a new line; the "else" must appear on a line which includes the termination of the command, or group of commands, that executes if the condition is "TRUE."

The following is an example of valid layout with line breaks:

```
if                    # Start the "if else" construction.
   (a<75)             # This is the condition that controls branching.
     {a <- log(a)     # Branch to here if condition is "TRUE."
      b <- 15         # After here skip to "confluence."
 } else               # Note "}" closing the previous command.
     {a <- a+214      # Branch to here if condition is not "TRUE"
      b <- 0}         #
c <- a*b              # confluence: reach here regardless of branch taken
```

The indenting in this example makes no difference to **R** but is intended to help visual parsing.

16.1 Conditional Subsets of a Vector

A conditional expression inside the square brackets designating a vector subset will pick out the elements of the vector for which the condition is "TRUE." For example:

```
W <- V[V<5&V>0]  # loads into vector W, in the order encountered, all
                 #  the elements in vector V which are larger than 0
                 #  and smaller than 5; if there are k such values,
                 #  W will then have k elements
```

Similarly, to identify the index numbers (subscript) for which a condition on the vector elements is true:

```
J <- which(x>2)  # loads into vector J, the index numbers (subscript
                 #  values) of the elements of vector x which are
                 #  larger than 2, in the order encountered
```

And, in the same vein, to count the number of elements in a vector for for which a condition is true:

```
n <- sum(V==3)   # loads into variable n, the numbers of elements
                 #  (count) of vector V which are equal to 3
```

17 LOOPS

Loops may be specified to repeat certain blocks of code.

A "for" loop increments a named index (counter) in a specified range, and repeatedly executes a specified expression using each value of that index. For example:

```
for (i in j:k) b   # repeatedly does "b" (which may be grouped commands
                   #  in curly brackets), for the number of times the
                   #  counter i goes from j thrugh k, where i itself
                   #  may be used (but not changed) in the expression b
```

A "while" loop repeatedly executes a specified expression for as long as a specified condition holds. For example:

```
while (c) b   # repeatedly does "b" (which may be grouped commands in
              #  in curly brackets), for as long as c is "true,"
              #  where operations inside the expression "b" can
              #  change the condition "c"
```

Though syntactically valid, explicit loops are not a graceful fit to the inner workings of **R** and therefore may use up lots of memory or computing time. If the number of passes through the loop is large, this may bog down, which is a disapointing practical limitation of **R** for certain kinds of applications. Repeated operations carried out through the vector and matrix functions in **R** will generally run faster, sometimes much faster, so that will prove to be the better option if it makes algorithmic sense.

18 USER DEFINED FUNCTIONS

A new function may be defined in the session, with a "function" command. The syntax is

```
newname <- function(a,b,c) d
             # defines a new function, called "newname," which will
             #  take on the value of the expression "d" (which may be
             #  grouped commands in curly brackets), when the
             #  values of (a,b,c) which are used in "d" are communicated
```

The syntax for using the new function then would be

```
w <- newname(x,y,z)  # loads into w the result of using (x,y,z) as
             #  the values for (a,b,c) in the expression "d" specified
             #  in the function statement which defined "newname"
```

Only the value of the expression is automatically communicated back from the function. Interim values inside the calculation of that expression are ephemeral. If the expression calculated by the new function involves grouped commands involving more than one assignment, the last assignment constitutes "the" evaluated expression.

The object corresponding to the expression calculated by the new function may be a single value, a vector, a higher order array, a list, etc., as determined by the context and explicit declarations.

An alternative for communicating all the interim values from a defined function to the workspace (where it will then be accessible to the rest of the session, is to designate their assignments with the operator "<<-" in place of "<-" inside the expression defined in the function statement. This runs the risk of overwriting objects in workspace which have the same name, so it limits the portability of the function.

As long as all the assignments inside the expression defined in the function statement are done with the usual "<-" operator, the names are "local" and will not interfere with names in the active workspace, even if, coincidentally, the names are the same. In this way, the statement using the function has complete control over the selection of named objects which may get overwritten.

19 MORE CONTROL OVER READ AND WRITE

19.1 Better Print Layout

The print command by itself offers limited control, and clutters the output with leading numbers in square brackets. For more flexibility and control of printing a line, an illustration of a more elaborate command is

```
cat(paste(a,b,"message",c,"\n")) # write contents of b, contents of c,
                                 #  the text string that is in double
                                 #  quotes, the contents of c, and then
                                 #  end the output line
                                 #  (the "\n" codes for a new line)
```

This combination of "paste" inside "cat" eliminates the leading count of items appearing in square brackets, allows the printing of several items on the same line, allows the printing of both character and numeric variables on the same line, and prints the character variable contents as is rather than putting it in quotation marks in the printed line.

20 GRAPHICS

20.1 The Graphics Window

A command to display a graphic will display in its own graphics window. If no graphics window is open at the time the command executes, the command will automatically open a

graphics window. If a graphics window is already open, the command will display the graphic in that window (over-writing the display of anything that was displayed there previously). In order to open a new graphics window, in addition to any graphics windows already open, so as to keep multiple graphics on display (the next graphics display command will display the graphic in the new window), the command is:

```
dev.new()
```

The graphics window may be partitioned into panels which will take the next several graphics (until each panel has a graphic in it). For example:

```
par(mfrow=c(3,2))  # partitions the window into 6 panels as
                   #  3 rows and 2 columns
```

20.2 Plot Values that are in a Vector

This is a "scatter plot" of the values in one vector against their sequence (order, index or vector subscript).

```
plot(b)   # graph the values in vector b, in sequence, against
          #  sequence number
```

20.3 Scatter Plot

A scatter plot can be made from 2 vectors, assuming they constitute paired values (x,y).

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

More control may be achieved with additional arguments:

```
plot(x,y,'o')          # connect the point symbols with lines
plot(x,y,type='l')     # connect the point coordinates with lines, but
                       #  do not show any symbol for the points
plot(x,y,col='blue')   # specify the color for the points
plot(x,y,ylim=c(-1,2)) # specify the y-axis window
plot(x,y,xlab="lbl")   # specify the x-axis label
plot(x,y,main="title") # specify the title (head)
```

These arguments may be incorporated in various combinations.

Additional points or lines specified in other vectors may be added to the plot, with the command:

```
points(z,w)                    # add the points whose x and y coordinates
                               #  are in the vectors z and w
points(r,s,type='l',col='red') # add the red line connecting the points
                               #  whose coordinates are in the vectors
                               #  r and s
```

20.4 Box Plot

A box plot is a way to represent a distribution of values.

```
boxplot(x)   # graph the distribution of values in vector x as
             #  a vertical spread
```

The box plots of several sets of values which are in separate vectors may be displayed in the same graphic. For example:

```
boxplot(x,y,horizontal=T)   # graph the distribution of values in vector x
                            #  as a horizontal spread, and stack the graph
                            #  of the distribution of values in vector y
                            #  as a horizontal spread above it, on the
                            #  same scale
```

20.5 Histogram

The numerical values in a vector may be tallied in a histogram

```
hist(b)   # make a histogram of count of the values in vector b,
          #  where R automatically determines the bin width and
          #  the number of bins, to accomodate the range of values
```

More control over a histogram can be achieved with additional arguments:

```
hist(x,breaks=100)           # forces 100 equal-width bins
hist(x,breaks=c(-10,0,5))    # forces 2 bins, with boundaries -10 to 0,
                             #  and zero to 5; error if any values fall
                             #  outside the defined bins; bin height
                             #  gives the count in that bin (but bin
                             #  area is meaningless because of unequal
                             #  bin widths)
hist(x,breaks=seq(0,101,5))  # forces bins of width 5, with the
                             #  left boundary of the lowest bin being 0,
                             #  with the right boundary of the highest
                             #  bin being the largest increment that does
                             #  not exceed 101 (this will come out to be 100)
hist(x,freq=F)               # scales the vertical axis in "density" so
                             #  that the area under the "curve" equals 1
                             #  (as if this were a probability density graph
                             #  of a continuous distribution; this is not the
                             #  same thing as a "bar graph" of the histogram
                             #  where the height of bar give the probability
                             #  of that bin)
```

The numerical results of a histogram tally can be stored as a "list" object, and the numerical components can be unpacked from the list and then used as vectors of values for any purpose:

```
h <- hist(x)                    # loads the histogram as a "list" in h
                                #  without displaying it
```

This would offer a way to archive the graphic itself as an "object" to be recalled (and possibly displayed) later from workspace or from a "workspace image" saved in a ".RData" file.

Once a histogram has been stored in the "list" h:

```
h                               # displays the histogram in a graphics
                                #  window
c <- h$counts                   # loads into vector c the counts by bin
f <- h$density                  # loads into vector f the "density" for each
                                #   bin, as if this were a representation
                                #   of a continous probability density as
                                #   a step function
p <- h$mids                     # loads into vector p the midpoints by bin
b <- h$breaks                   # loads into vector b the bin boundaries
```

The "$" operator illustrated above is used to specify the "component" of a "list" object.

The vector of counts can be normalized directly, which then would allow plotting the probability in each bin, with a block of code such as:

```
b <- seq(0,100,10)      #  establish the bin boundaries
h <- hist(x,breaks=b)   # loads the histogram as a "list" in h
c <- h$counts           # loads into vector c the counts by bin
s <- sum(c)             # sum the counts
c <- c/s                # normalize the counts to fraction of
                        #  counts in each bin, by bin
m <- h$mids             # loads the midpoint value identifying each bin
plot(b,c)               # plot the probability of each bin
```

A cumulative histogram (to the resolution of the bin width) can be formed from the nornalized count vector returned from a simple histogram, as in the block of code:

```
v <- hist(x,breaks=100) # stores bin information and vector
                        #  of by-bin counts in "list" v
c <- v$counts           # loads into vector c the counts by bin
s <- sum(c)             # sum the counts
c <- c/s                # normalize the counts to fraction of
                        #  counts in each bin, by bin
plot(v$mids,cumsum(c))  # plot cumulative probability (fraction)
                        #  against bin midpoint
```

If you wanted to know how many of the values in vector v are smaller than a cutoff value CT, there is a way to specify the bin boundaries for a histogram and then access the counts by bin. This requires two commands:

```
h <- hist(v,breaks=c(0.0,CT,1000.0))   # store the 2-element histogram in
                                       #  as a "list"
v <- h$counts                          # load the 2 respective counts
                                       #   of the number of values per
                                       #   bin into vector v
```