Statistical Computing with **R**

Eric Slud, Math. Dept., UMCP

October 21, 2009

Overview of Course

This course was originally developed jointly with Benjamin Kedem and Paul Smith. It consists of modules as indicated on the Course Syllabus. These fall roughly into three main headings:

(A). **R** (& **SAS**) language elements and functionality, including computer-science ideas;

(B). Numerical analysis ideas and implementation of statistical algorithms, primarily in **R**; and

(C). Data analysis and statistical applications of (A)-(B).

The object of the course is to reach a point where students have some facility in generating statistically meaningful models and outputs. Wherever possible, the use of **R** and numerical-analysis concepts is illustrated in the context of analysis of real or simulated data. The assigned homework problems will have the same flavor.

The course formerly introduced **Splus**, where now we emphasize the use of **R**. The syntax is very much the same for the two packages, but **R** costs nothing and by now has much greater capabilities. Also, in past terms **SAS** has been introduced primarily in the context of linear and generalized-linear models, to contrast its treatment of those models with the treatment in **R**. Students in this course have often had a separate and more detailed introduction to **SAS** in some other course, so in the present term we will not present details about **SAS**, in order to leave time for interesting data-analytic topics such as Markov Chain Monte Carlo (`MCMC`) and multi-level modeling in **R**.

Various public datasets will be made available for illustration, homework problems and data analysis projects, as indicated on the course web-page.

The contents of these notes, not all of which are posted currently, and which will be augmented as the term progresses, are:

1. `Introduction to R`
Unix and R preliminaries, R language basics, inputting data, lists and data-frames, factors, functions.

2. `Random Number Generation & Simulation`
Pseudo-random number generators, shuffling, goodness of fit testing.

3. `Graphics`

4. `Simulation Speedup Methods`

5. `Numerical Maximization & Root-finding`
(respectively for log-likelihoods and estimating equations)

6. `Commands for Subsetting`
Manipulating Arrays and Data Frames

7. `Spline Smoothing Methods`

8. `EM Algorithm`

9. `The Bootstrap Idea`

10. `Markov Chain Monte Carlo`
Metropolis and Gibbs Sampling Algorithms
Convergence Diagnostics for MCMC
Bayesian Data Analysis applications using `WinBugs`

11. `Multi-level Model Data Analysis`
Linear and Generalized Linear Model Fitting and Interpretation

A few Exercises are contained in these notes, but all formal Homework assignments are posted separately in the course web-page Homework directory.

1 Introduction to R

R and **Splus** are so-called *object-oriented languages*, which means roughly that they are organized to recognize both inputs and outputs (such as numerical data and fitted statistical models) from standard computer-representations, which have the structure primarily of *lists (of basic data structures) with attributes* of several special types. All-encompassing definitions are elusive, but the main idea is that outputs of onestage of analysis can be computed on and then inputted to furtherstages [including further model-fitting, pictures and graphs, etc.] without re-defining their structure. This makes **R** especially suited to *interactive* analysis.

1.1 Unix Preliminaries

Unix commands are typed immediately after a Unix prompt, such as

```
 evs@mary.umd.edu%
```

A useful basic list of commands is:

`mkdir`	`Creates directory, e.g. "mkdir .Data" from home directory.`
`pwd`	`Prints current directory, e.g. /home2/bnk/dirA/.../dirN`
`man`	`Unix help, e.g. "man pwd" gives information about "pwd".`
`cd`	`Change directory, e.g. "cd .Data" moves to subdirectory .Data.`
`ls`	`Lists all files excluding dot files.`
`ls -a`	`Lists all files including dot files.`
`ls -l`	`Lists files in long format. Size in bytes.`
`ls -lt`	`Lists files in long format and sort by time of last change.`
`ls -lut`	`Lists files in long format and sort by time of last access.`
`ls -s`	`Lists files and their sizes.`
`rm`	`Removes a file. E.g. "rm filename".`
`\rm`	`Removes a file no questions asked.`
`cp`	`Creates a copy of a file. E.g. "cp A B" copies A into B.`
`du`	`Size of the working directory in kilobytes.`

```
lpr -Plw2301 filename : prints file "filename" on 2nd floor printer.
"echo $PRINTER" gives the default printer.
```

Text Editors

There are several options such as 'text editor', 'emacs', 'pico', etc. Emacs is convenient. To edit a file, from within Unix type

```
emacs filename &
```

This will open up a window, containing menus, ready for editing.

1.2 R Preliminaries

(a) Get into **R** by typing **R** following a Unix prompt. Do this only after deciding where (i.e., in what Directory) you want your saved data to reside. Then the **R** save-area will be the subdirectory `.RData` within your current directory.

In case you already have a save area, named for a special purpose, e.g. as "Work.RData", then when you invoke **R** and start a session, you can issue the **R** command

```
load(``Work.RData'')
```

to make all of the contents of the workspace `Work.RData` available in the current session.

(b) Exit **R** by typing `q()` following the **R** line-prompt > . If you want to save everything in the current area as an **R** workspace (say `NewSpace.RData`) for future reference, then before you quit, issue a command

```
save.image(``NewSpace.RData'')
```

When you quit, you will be prompted whether you want to save the current workspace: if you say yes, then it will be saved in `.RData`.

(c) Whenever an assignment has been made to an object name, that object is retained in the current workspace until removed or another assigment is made to the same name.

(d) To see what you have in your workspace at any time, from within **R**, type **ls( )** following the **R** prompt.

(e) Specify a text editor for help and function-editing windows by typing a command (after the line-prompt) such as:

```
options(editor="emacs")
```

(f) What you type following the **R** prompt is always an *expression*. **R** scans to the end of each typed line to make sure that the syntax is (possibly) correct so far, and to check at the end of the line whether the expression is complete or continuation-lines (prompted by a new line on which **R** types '+') are needed. When a syntactically complete expression is reached, **R** evaluates it if possible, issuing error messages if not all variables exist within the directories on the search-list.

(g) Apart from arithmetic operations, **R** commands are given in the form of functions, e.g.: `q()`, `sum(xvec)`, `plot(x,y)`, etc.

(h) Unless an expression specifies an action (such as assignment '<−', or graphical plotting, the result of evaluating the expression is an object (a summary of) which is printed. If after seeing the object (before issuing any other **R** commands) you want to assign and save it, type (after the prompt)

```
newname <- .Last.value
```

and the assignment operator `<-` can also (always) be replaced by `=` .

1.3 R Language Elements

R operates on objects which all have the structure either of *functions* (discussed later) or of *vectors* or *lists* with these basic elements with attached lists of *attributes*.

So what are lists made of ? To begin with, lists are made of the basic data-objects organized as *strings* or *vectors*, and can be of the followingtypes: Numerical (or Complex), Boolean (T/F), and Character (with a string `"XYZname"` allowed to be a *single* vector-element).

```
> x = (1:9) - c(3,1,7)
> x
[1] -2  1 -4  1  4 -1  4  7  2

> c("ABC", "g", "Maryland")
[1] "ABC"      "g"        "Maryland"

> y = ( (1:9) - c(3,1,7) > 0 )
> y
[1] F T F T T F T T T
```

Throughout **R**, there are useful commands to convert types :

```
> as.numeric(y)
[1] 0 1 0 1 1 0 1 1 1

> as.character(x)
[1] "-2" "1"  "-4" "1"  "4"  "-1" "4"  "7"  "2"

> as.numeric(.Last.value)
[1] -2  1 -4  1  4 -1  4  7  2
```

Every **R** object has a 'length', which for a vector is just the number of entries; for a list is the number of components; and for a function is one plus the number of arguments. For each object, there is a list of 'attributes' which may be empty but might include: 'dim' and 'dimnames' for matrices and arrays; 'names' for vectors,, lists, and functions; and 'class' for data-frame and fitted model objects. You can also use these attributes as functions, e.g. after defining the **R** data-frame **LTdata** via the `read.table` command in Section 1.5 below:

```
> names(LTdata)
[1] "Stratum"  "Last10." "Cellct" "Tenure" "Race" "NumPer"
[7] "Ethnic"   "Locale"
```

There are several types of vectors with attributes, which constitute the next stage of **R** objects. These include *matrices and arrays* — which we discuss now — and also *factors*, which are treated later.

A matrix or array should be regarded as a vector, consisting of the entries concatenated in lexicographic order of the array-indices (with the earlier array-indices moving most rapidly), together with a (possibly empty) 'attributes' list giving the dimension (as a vector of integers) and the row and column names.

```
> xvec = runif(50)
> length(xvec)
[1] 50
> ymat = matrix(xvec, ncol=5)
> length(ymat)
[1] 50
> attributes(ymat)
$dim:
[1] 10  5
> sum(abs(c(ymat)-xvec))
[1] 0
```

1.4 Simplest Operations on Vectors and Arrays

As we saw above, you can use function `c( )` to create vectors by concatenation, and two existing vectors can be concatenated to form a new one

```
> xvec = c(1:3, c(7,9,1,4))
> xvec
[1] 1 2 3 7 9 1 4
```

A sub-vector of an existing vector `xvec` can be created as the same object `xvec[ivec]` in either of two ways : `ivec` may be a vector of integer indices of the length of the subvector you want *or* a Boolean or $0, 1$ valued vector of the same length as `xvec`:

```
> xvec[2*(1:3)]
[1] 2 7 1
> xvec[c(F,T,F,T,F,T,F)]
[1] 2 7 1
```

Standard mathematical functions automatically apply componentwise to vectors:

```
> cos(pi*(0:6))
[1]  1 -1  1 -1  1 -1  1
> xvec > 3
[1] F F F T T F T
```

As a result, you can refer to subvectors of a given vector containing all components satisfying a specified condition

```
> xvec[xvec>3]
[1] 7 9 4
```

Note: if you want to use equality in defining Boolean variables, you must use `==` rather than `=` . 'Not equal' is denoted `!=` .

To create a matrix or array from a vector:

```
> ymat = matrix(c(xvec,0), ncol=2, dimnames=list(NULL,c("1st","2nd")))
> ymat
     1st 2nd
[1,]   1   9
[2,]   2   1
[3,]   3   4
[4,]   7   0
> array(c(ymat), dim=c(2,2,2))

, , 1
     [,1] [,2]
[1,]    1    3
[2,]    2    7

, , 2
     [,1] [,2]
[1,]    9    4
[2,]    1    0
```

Note that in the `matrix` function, inserting the final option '`, byrow=T`' before the final right-paren would cause the input vector elements to be created with first row (1,2), second row (3,7), etc.

The objects `as.vector(ymat)` and `c(ymat)` are the same: just the vector of elements (same as `c(xvec,0)` in this case).

Mathematical operations like `ymat^2` applied to a matrix are again applied componentwise, so the resulting object is again a 4×2 matrix.

Some useful functions which apply to vectors are: `sum`, `mean`, `var`, `sd`, `summary`. If they are applied to matrices, the result for all but `summary` is the same as if applied to `as.vector` of the matrix. The function `summary` applied to a matrix is a `"table"` consisting of summaries of the columns. Applying `summary` to a vector gives a standard set of summary-statistic values (min, max, quartiles, and mean) and is often a quick, single-line method of getting an idea what an existing vector contains.

Some useful functions and operations on matrices are:

`t(ymat)`	*transposed matrix*
`diagonal(xvec)`	*diagonal matrix with diagonal vector* `xvec`
`diagonal(ymat)`	*vector equal to main diagonal of* `ymat`
`solve(zmat)`	*inverse of square matrix*

Submatrices and sub-arrays can be created using the same logic as sub-vectors: refer to vectors of indices in the appropriate dimension, with the convention that leaving a dimension blank means all indices in that dimension are included.

```
> ymat[c(1,3),]
     1st 2nd
[1,]   1   9
[2,]   3   4
> ymat[2,]
 1st 2nd
   2   1
```

Thus the i'th row (respectively j'th column) of a matrix `ymat` is a *vector* `ymat[i,]` (resp. `ymat[,j]`).

1.5 Inputting Data & Recovering Existing Objects

Throughout an **R** session, you will be defining and assigning **R** objects. There are a few main ways for you to get access to existing datasets and (if desired) to save them into your work-area (i.e., your designated `.RData` workspace).

The simplest is to enter (small) datasets from the terminal:

```
> grades = c(85, 73, 44, 97, 65)
> quizzes <- scan()
1: 4 8 7 6 5 9 9 8 7
10:
```

Here we are using the 'scan' command, which inputs a designated (ASCII) file into a vector; in the usage just given, the ASCII file is created from the terminal input. A more elaborate use of the scan command, which first strips the two header lines, then inputs the data as a long vector, follows:

```
> LTvec = scan("/home1/evs/LTdata.asc", skip=2, what=character())
> length(LTvec)
[1] 256
> LTvec[1:7]
[1] "1"     "7267"  "94069" "0"     "NW"    "MP"    "HI"
```

Note: we would not have needed the '`what=...`' entry, except that the data consist both of numbers and character fields. Since we really want the data in a matrix in our illustration below, and want to allow some columns as categorical and others as numerical, a much easier way is

```
> LTdata = read.table("/home1/evs/LTdata.asc", header=T)
```

Many datasets, including this one, are available on the course website in (compressed) ASCII format, and you can execute commands like the previous one after first copying the data from a browser window into a text file in your home directory and saving it.

I will also place some previously existing **R** objects, including data, in the public `/nfs/projects/statdata` directory `Rstf.RData`, and this and other useful **R** workspaces containing data objects can also be found in the course web-page `Data` directory. On the **MathNet** network You can gain access to them by the command

```
> attach("/nfs/projects/statdata/Rstf.RData")
> objects(2)
```

the second line of which will show you the **R** objects available in that **R** workspace. Otherwise, you can download the `Rstf.RData` workspace to a home directory from which you invoke **R** for this course. Then, when you start a new **R** session, you can incorporate the desired data into your session workspace by the command `load("Rstf.RData")` or can make them available by `attach("Rstf.RData")`.

1.6 A Data Illustration

Here is a small dataset concerning the demographics of households which were among the last 10% in their Census Tracts to be enumerated in the 1990 Decennial Census, from a Census report by T. Krenzke (1997). There are 5 binary variable categories:

Tenure of housing unit:	O = Owner, R = Renter
Race of head-of-household:	NW = Nonwhite, WH = White
Number of Persons in household:	MP = Multiple-person, SP = Single
Ethnicity (head-of-household):	HI = Hispanic, NH = Non-Hispanic
Locality:	R = Rural, U = Urban

For each demographic combination, **Last10%** is the number of (enumerated) households, out of the total number **Cellct**, falling among the last tenth enumerated in their Tracts.

```
Stratum  Last10%  Cellct  Tenure  Race  NumPer  Ethnic  Locale

   1      7267      94069      O    NW      MP      HI      R
   2     53420     803461      O    NW      MP      HI      U
   3     67462     842662      O    NW      MP      NH      R
   4    276979    3805838      O    NW      MP      NH      U
   5      1039       9378      O    NW      SP      HI      R
   6      7492      66753      O    NW      SP      HI      U
   7     19648     194929      O    NW      SP      NH      R
   8     75485     775073      O    NW      SP      NH      U
   9     13775     171222      O    WH      MP      HI      R
  10     75581    1205599      O    WH      MP      HI      U
  11    900518   13582241      O    WH      MP      NH      R
  12   1438974   27514002      O    WH      MP      NH      U
  13      2254      24443      O    WH      SP      HI      R
  14     13192     170659      O    WH      SP      HI      U
  15    226360    2730240      O    WH      SP      NH      R
...
```

For convenience, we assume that these data reside in an ASCII file called **/home1/evs/LTdata.asc**, which has 34 lines (two lines of header, as shown). In section 1.5 above, the data were processed via `read.table` into a data-frame **LTdata**. As a side-effect, each of the columns has become a *factor*:

```
> attributes(LTdata[,"Tenure"])
$levels:
[1] "O" "R"

$class:
[1] "factor"
```

We next fit a simple linear regression model to the ratios **Last10./Cellct** in terms of the binary factors without interactions. Some simple non-graphical summaries follow:

```
> fitLT = lm(Last10./Cellct ~ Tenure + Race + NumPer + Ethnic
    + Locale, data=LTdata)
```

```
> names(fitLT)
 [1] "coefficients"  "residuals"     "fitted.values" "effects"
 [5] "R"             "rank"          "assign"        "df.residual"
 [9] "contrasts"     "terms"         "call"

> summary(LTdata[,2]/LTdata[,3])
      Min.   1st Qu.   Median    Mean  3rd Qu.     Max.
 0.0522997 0.0793687 0.107191 0.10301 0.122615 0.159824

> summary(fitLT$fitted)
      Min.   1st Qu.  Median    Mean  3rd Qu.     Max.
 0.0547577 0.0821846 0.10301 0.10301 0.123835 0.151262
> summary(fitLT$resid)
       Min.     1st Qu.      Median        Mean    3rd Qu.      Max.
 -0.0219314 -0.00548602 0.000896611 4.87891e-19 0.00662384 0.0164323
> fitLT$coef
 (Intercept)       Tenure            Race       NumPer          Ethnic
 0.103009888 0.0218158505 -0.00479605427 0.0122276504 -0.00349861644
         Locale
 -0.00591400994
> unlist(lapply(LTdata,levels))
 Tenure1 Tenure2 Race1 Race2 NumPer1 NumPer2 Ethnic1 Ethnic2
 "O"     "R"     "NW"  "WH"  "MP"    "SP"    "HI"    "NH"
 Locale1 Locale2
 "R"     "U"
```

The summary function has been used to display the 32-vectors of response variables, fitted values and residuals. The numerical coding of the binary factors is (-1,1), as can be seen for example from

```
> model.matrix(fitLT)[1:5,]
  (Intercept) Tenure Race NumPer Ethnic Locale
1           1     -1   -1     -1     -1     -1
2           1     -1   -1     -1     -1      1
3           1     -1   -1     -1      1     -1
4           1     -1   -1     -1      1      1
5           1     -1   -1      1     -1     -1
```

We have now gotten to a point where we must talk about lists: how to create them and how to refer to their components. We explain in the following subsection the **R** list-related commands used above .

1.7 Lists

Lists can be created by concatenating **R** objects:

```
> listout = list(name1 = obj1, name2 = obj2, name3 = obj3)
> names(listout)
[1] "name1"   "name2"   "name3"
```

The objects we concatenate will themselves be vectors and lists, possibly with ‘attributes’. Here is a concrete, not too simple, example:

```
> listex = list(x=c(1,4), y=function(x) x^2, z=fitLT)
> listex
$x:
[1] 1 4

$y:
function(x)
x^2

$z:
Call:
lm(formula = Last10./Cellct ~ Tenure + Race + NumPer + Ethnic +
        Locale, data = LTdata)

Coefficients:
 (Intercept)       Tenure           Race       NumPer
 0.103009888 0.0218158505 -0.00479605427 0.0122276504
         Ethnic          Locale
 -0.00349861644 -0.00591400994

Degrees of freedom: 32 total; 26 residual
Residual standard error: 0.00902571784
```

There are two equivalent ways to refer to a list component, by number and by name. In the last example, `listex[[1]]` and `listex$x` both refer to the vector $(1, 4)$; `listex$y` is the function x^2, and `listex[[3]]` is the linear-model fitted object `fitLT` discussed in Section 1.6 above. We saw from `names(fitLT)` that `fitLT` itself was a list with various components (mostly vectors) related to residuals, degrees of freedom, coefficients, etc. Thus `fitLT$coef` is the vector of fitted coefficients. (Often, in **R**, the standard model-object list-components do not need to be spelled out in full — just far enough so that there is no ambiguity with other components.)

A tremendously useful kind of list is the **R** *data-frame*: the elements of a matrix are given the structure of a list whose components are the columns. This has the advantage, as for **LTdata** described above, that the different columns can have different data types. In addition, data-frames retain the '*dim*' attribute along with the convenience of allowing rows, columns and submatrices to be referenced just as though the frame were a matrix. Data-frames will be used frequently in applying **R** statistical analysis functions.

In section 1.6, we used a command `unlist(listname)`: it simply concatenates the elements of the list components as one long vector.

Finally, although **R** functions are not themselves lists, they have a 'names' attribute, which is a quick way to remind yourself of the order of arguments needed for a function.

```
> names(lm)
 [1] "formula"   "data"      "weights"   "subset"    "na.action"
 [6] "method"    "model"     "x"         "y"         "contrasts"
[11] "..."       ""
```

1.8 Digression on Factors

We know already that factors are vectors together with 'levels' attribute giving (as character strings) the distinct values occurring in the vector of elements and the class attribute 'factor'. How can one transform a numeric factor back to a numeric vector ?

```
> smpfac = sample(1:20,30, replace=T)
> smpfac
[1] 6  18 10 6  19 6  3  5  14 11 8  16 20 17 18 7  7  17 7  2
[21] 18 9  2  15 11 12 5  7  4  18
> tmpfac = factor(smpfac)
> levels(tmpfac)
[1] "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "14"
[13] "15" "16" "17" "18" "19" "20"
> as.numeric(tmpfac)
[1]  5 16  9  5 17  5  2  4 12 10  7 14 18 15 16  6  6 15  6  1
[21] 16  8  1 13 10 11  4  6  3 16
> sum(abs(smpfac-as.numeric(levels(tmpfac)[as.numeric(tmpfac)])))
[1] 0
```

Thus the as.numeric version of the factor is the sequence of indices within the (ordered) levels for the vector of factor values.

1.9 Miscellaneous Commands

`seq, rep, replace, ifelse`

```
        > y = replace(x,(1:length(x))[x>90],NA)
        > y = ifelse([x>90],NA,x)
        > z = rep(c(1,2,3),10)
```

`if, for, apply`
`runif, sample` & other pseudorandom variate generators
`sort, order, diff`
`search, .First`

```
> .First = function()
{
        options(editor = "emacs")
        attach(``NewSpace.RData'')
        load(``OtherR.RData'')
        help.start()
}
```

1.10 Loose Ends

(1) *Remark*: within **R** commands like `scan` or `attach` or `get`, the abbreviation for your home directory will not be recognized, so you must use your counterpart to my **/home1/evs**.

(2). *Attaching Data-frames*

```
> attach(exampfram)
>objects(2)
 [1] "AGEVAR"  "ALBUMIN" "AUX"     "CCHOL"   "CIRRH"   "COND"    "DTH"
 [8] "EVTTIME" "IDNUM"   "LOGBILI" "OBS"     "TRTGP"
```

Each of the following sets of commands does the same thing !

```
> y = seq(a, b, (b-a)/n)
> y = a + (0:n) * ((b-a)/n)
```

For ASCII data `1, 2, NA, 9, 8, NA, -3, 7` in file **testdat**:

```
> replace( z= scan("testdat", sep=","), is.na(z), -999 )
> as.numeric( ifelse( (w = scan("testdat", sep=",",
        what=character()))=="NA","-999",w) )
```

```
> rep(1:3,10)
> 1 + (0:29) %% 3
```

```
> c(zmat %*% rep(1/ncol(zmat),ncol(zmat)))
> apply(zmat,1,mean)
```

Apply either of the following after: `set.seed(153)` :

```
> sort(w <- runif(100))
> w = runif(100) ; w[order(w)]
> sample(1:10,100, replace=T)
> 1 + trunc(runif(100)*10)        ### equal only in distribution
```

Finally, here are *three* different ways to tabulate, in sorted increasing order, the distinct values occuring in a numeric vector `zv` :

```
> table(zv)
> { szv = sort(zv)
    ind = (1:length(szv))[diff(c(-1.e8,szv))>0]
    cbind(szv[ind],diff(c(ind,length(szv)+1))) }
> { levs = as.numeric(levels(factor(zv)))
    szv = split(zv,levs)
    unlist(lapply(szv,length)) }
```

1.11 Functions

Functions are defined and customized within **R** according to the syntax

```
> fname = function(x,y,z, w=w0) {
```

. . . *body of function (series of commands, usually assignments)*

```
  lastexpr }
```

where `lastexpr` is an **R** expression — the resulting object created by the function — which can involve variables `x,y,z,w` as well as 'local' variables defined within the function body and *not* saved in the **R** work-area afterwards. The function arguments can be any **R** objects, including functions and lists, and may be named in the original function specification (which allows you to enter the function-arguments by name in possibly the wrong order). In addition, the use of named arguments allows you to designate a default value for the named argument (i.e., a value which will be assumed if you do not specify it when calling the function). EXAMPLES FOLLOW:

```
> f1 = function(x, c=3) sqrt(1+c*x*x)
> f2 = function(x,g=f1) g(x)
> f1(12)
[1] 20.80865
> f2 = function(x,g=f1) g(x)
> f2(12)
[1] 20.80865
> f2(12,sin)
[1] -0.5365729
> f2(12,function(x) x)
[1] 12
```

Several very useful **R** commands do exactly this, with functions as arguments, including `apply`, `sapply`, and `uniroot`. EXAMPLES FOLLOW:

```
> bmat = matrix(runif(40), ncol=8)
> apply(bmat,1, function(z) sqrt(var(z)) )
> rtlst = uniroot(function(x) x^2-2, c(0,2)  )
> rtlst$root
[1] 1.414213
> rtlst$f.root
[1] -6.855473e-07
```

You can define functions *either* in-line following **R** prompts as above (including copying in of text from separate text-editor windows) *or* by

```
> fn1 = ed(fn1,editor="emacs")    ### or
> fix(fn1)
```

1.11.1 Vectorizing Function Operations

Functions can always be applied to vectors or matrices. For example, one way to calculate sample variances of the rows of a matrix `bmat` is

```
> n = ncol(bmat)
> v1 = rep(1,n)
> avec = ( bmat^2 %*% v1 - (bmat %*% v1)^2/n )/(n-1)
```

Another way to do it is using `apply`, which also allows complete flexibility in the choice of function, which we exploit here by directing **R** to omit missing values before calculating variances:

```
> avec = apply(bmat,1,var, na.rm=T)
```

But functions written for scalars may not always allow vectors to pass through, or may not work as you expect if you are not careful. For example

```
> yfcn1 = function(v, w, Acond, fnam) {
# Take  Acond  to be booleanl; v,w  vectors of same length
        fnam(if(Acond) v else w) }
> yfcn1(pi,pi/2,T,sin)
[1] 1.224606 e-16
```

```
> yfcn1(pi,pi/2,F,sin)
[1] 1

> yfcn1((1:10)*pi/10, rep(c(pi,pi/2),5), cos(1.2^(1:10)*pi) > 0, sin)
[1] 1.224606e-16 1.000000e+00 1.224606e-16 1.000000e+00 1.224606e-16
[6] 1.000000e+00 1.224606e-16 1.000000e+00 1.224606e-16 1.000000e+00
Warning messages:
  Condition has 10 elements: only the first used in: if(Acond) v else w
```

This function performs correctly if **v, w, Acond** are of length 1 but not otherwise ! To allow them all to be vectors, change the function to

```
> yfcn2 = function(v, w, Acond, fnam)  fnam(ifelse(Acond,v,w))

> yfcn2((1:10)*pi/10, rep(c(pi,pi/2),5), [cos(1.2^(1:10)*pi > 0], sin)
[1] 1.224606e-16 1.000000e+00 8.090170e-01 9.510565e-01 1.000000e+00
[6] 1.000000e+00 8.090170e-01 5.877853e-01 1.224606e-16 1.224606e-16
```

1.12 Working with Tables & Arrays

Whether because you are working with categorical data, or because you want to summarize cross-tabulated characteristics of key explanatory variables, it is helpful to summarize quickly cross-tabulations arising from conditions. The primary command for this is `table`:

```
> table(LTdata$Last10./LTdata$Cellct>0.10,ifelse(LTdata$Cellct>100000,
+   "Big",ifelse(LTdata$Cellct>30000,"Med","Sm")),  LTdata$Locale)
, , R
      Big Med Sm
FALSE   4   1  1
 TRUE   5   2  3

, , U
      Big Med Sm
FALSE   8   0  0
 TRUE   7   1  0
```

```
> dim(.Last.value)
[1] 2 3 2
```

The output from `table` is an array; numeric or character variables are treated as factors, with 'levels' arising from distinct occurrences appearing as `dimnames`. Note that the same command can be used to provide dimnames in transforming a data-frame like **LTdata** into an array, which can also be useful:

```
> LTarray = array(c(LTdata[,2], LTdata[,2]/LTdata[,3]), dim=c(rep(2,6)),
    dimnames=dimnames(table( rep(c("Cell","LTfrac"), 16), LTdata[[4]],
    LTdata[[5]], LTdata[[6]], LTdata[[7]], LTdata[[8]])))
> unlist(dimnames(LTarray))
 [1] "Cell"   "LTfrac" "O"      "R"      "NW"     "WH"     "MP"     "SP"
 [9] "HI"     "NH"     "R"      "U"
```

Another operation which can be very handy is to collapse a data-frame by combining certain table-values (such as cell-totals). The command is `aggregate` or `aggregate.data.frame`:

```
> LTcomb = aggregate.data.frame(LTdata[,2:3], by=LTdata[,4:7], sum)
> dim(LTcomb)
[1] 16  6
> names(LTcomb)
[1] "Tenure"  "Race"    "NumPer"  "Ethnic"  "Last10." "Cellct"
> LTcomb[1,]
  Tenure Race NumPer Ethnic Last10. Cellct
1      O   NW     MP     HI   60687 897530
> LTdata[1:2,]
  Stratum Last10. Cellct Tenure Race NumPer Ethnic Locale
1       1    7267  94069      O   NW     MP     HI      R
2       2   53420 803461      O   NW     MP     HI      U
```

The output is another data-frame with columns `Cellct` and `Last10.`, obtained by summing these column entries over all records corresponding to each distinct combined value of the columns in the list 'by'. There is an important option `(drop=T)` in the argument list which can be used to delete any occurrence combination for the 'by' list which does not actually occur.

What **R** does with the columns in the 'by' list is to treat them as factors and then to create unique combinations by making the factor which is the *interaction* of those columns.

```
> interaction(factor(rep(c(2,5),3)),factor(rep(c("A","B","C"),2)))
[1] 2.A 5.B 2.C 5.A 2.B 5.C
> levels(.Last.value)
[1] "2.A" "5.A" "2.B" "5.B" "2.C" "5.C"
```

In case a specified `by` list is made up of a few columns (say 3), each of which has a large number of levels (say n_1, n_2, n_3), but for which relatively few of the $n_1 \cdot n_2 \cdot n_3$ factorial combinations actually occur, **R** still creates a factorial array of length $n_1 \cdot n_2 \cdot n_3$ before deleting anything: this wastes time and space and occasionally causes **R** to get stuck. There is an old computer-science trick, called **hash-coding**, which can be used to circumvent this problem. Instead of creating the by-list from separate columns, which we may think of as the first 3 columns of a data-frame `sampfram`, one could create a single factor (in an ordering which has very low probability of being any different from the lexicographical ordering of the 3 factors) with exactly the same number of distinct levels occurring, as follows:

```
 by = list( hashcod = cbind(as.numeric(sampfram[[1]]),
      as.numeric(sampfram[[2]]), as.numeric(sampfram[[3]]) ) %*%
        (runif(ncol(sampfram))*10^(5*(1:ncol(sampfram)))) )
```

The idea here is that the pseudo-random numbers generated via `runif` are given to double precision so that there is virtually no chance of a tie when the numerically indexed factor-levels are linearly combined with pseudo-random coefficients. Moreover, if the `sampfram` columns are of moderate size then there is a very small chance that the lexicographical factor-ordering is modified by this method of re-coding into a single factor.

The weights and the random numbers play different roles here: (i) the weights accomplish a (near-) lexicographical ordering in case the columns being manipulated are numeric with roughly the same range of orders of magnitude; (ii) in case there is no particular desire to maintain the lexicographical ordering, and there are possibly many columns with possibly very different dynamic ranges, the hash-coding trick with or without weights re-codes the multiple columns into a single one in such a way that with high probability the number of distinct values in the re-coded column is equal to the number of distinct combinations of values in the columns actually occurring in the columns being combined.

The same trick (with weights) can be used (and this was its original application) in sorting based on multiple categories: the syntax is `order(vec1, vec2, vec3)`, e.g., to find the index-permutation based on sorting `vec3` within `vec2` within `vec1`.

1.13 Functions in an Illustrative Simulation

Several aspects of vectorization and organizing **R** computations in terms of functions can be illustrated effectively in terms of a statistically meaningful example, a simulation to show the effect on logistic-regression coefficients of a random intercept coefficient.

1.13.1 *Description of Simulation*

The model which we propose to simulate and analyze repreatedly is the *Logistic regression* model

$$Y_j \sim \mathit{Binom}(30,\ \frac{e^{\beta_1+\beta_2 X_j}}{1+\ e^{\beta_1+\beta_2 X_j}}\ )\ \ ,\quad j=1,\ldots,50$$

The idea is that each Y_j represents a number of positive responses among 30 independent individuals who share a common value of the explanatory variable X, which we take to have *iid* unit-exponential components, and that all 50 groups of 30 share the same coefficients β, which we take to be $(-0.8, 0.3)$. A generalization of this model to include a *random intercept* would allow β_1 to be replaced by a (normally distributed) random variable, consisting of the previous constant value common to all groups of 30 and a random 'error' which is *iid* with one value for each group of 30. We do a 500-iteration simulation for each of two settings: first, with $\beta_1 = -0.8$, and second with $\beta_{1j} \sim \mathcal{N}(-0.8,\ 0.16)$.

```
> X1 = rexp(50)
  ymat1 = matrix(rbinom(25000,30,plogis(-0.8 + 0.3*X1)),
       ncol=50, byrow=T)
  ymat2 = matrix(rbinom(25000,30,plogis(-0.8 + 0.3*X1+
       0.4*rnorm(25000))), ncol=50, byrow=T)
  outmat1 = apply(ymat1,1, function(yrow,xvec)
        c(glm(cbind(yrow,30-yrow) ~ xvec, family=binomial)$coef,
        sum(yrow)/1500), xvec=X1)
# Note: could have omitted xvec argument & substituted X1 directly
> dim(outmat1)
[1]   3 500
> outmat1 = t(outmat1)
  dimnames(outmat1) = list(NULL,c("beta1","beta2","ptest"))
  outmat2 = outmat1
  for(i in 1:500) outmat2[i, 1:2] =
     glm(cbind(ymat2[i,],30-ymat2[i,]) ~ X1, family=binomial)$coef
  outmat2[,3] = ymat2 %*% rep(1/1500,50)
```

We can compare the estimated coefficients in various ways:

```
> apply(outmat1[,1:2],2,mean)
      beta1     beta2
 -0.7972393 0.2996552
> sqrt(apply(outmat1[,1:2],2,var))
      beta1      beta2
 0.07332403 0.05828725
> c(apply(outmat2[,1:2],2,mean), apply(outmat2[,1:2],2,sd))
      beta1    beta2     beta1      beta2
 -0.7692242 0.289693 0.1085505 0.09157609
```

The third components of `outmat1` and `outmat2` were included so that we can check in a coarse general way that the simulated data behaves as expected. For example, we know for the first simulation that the average response rate is given by

$$\int_0^\infty \frac{e^{-0.8+0.3x}}{1+e^{-0.8+0.3x}}\, e^{-x}\, dx$$

so we can check

```
> integrate(function(x) exp(-x)/(1+exp(0.8-0.3*x)), 0, Inf)$integral
[1] 0.3791168                                  ### error < 4.e-07
> c(summary(outmat1[,3]),sqrt(var(outmat1[,3])))
   Min. 1st Qu. Median   Mean 3rd Qu.  Max.
 0.3427   0.364 0.3727 0.3729  0.3807 0.408 0.01202048
> summary(outmat2[,3])                ### Note more spread out !
   Min. 1st Qu. Median   Mean 3rd Qu. Max.
 0.3313  0.3667 0.3773 0.3773  0.3887 0.43
```

and we could do a similar (but more laborious, because double-) integration to check `outmat2`. We will discuss next time a vectorization-trick to speed up the function-evaluations needed to make double or multiple integrations feasible. One could also provide a simple graphical output to indicate that the group-wise means fall within appropriate confidence-bands:

```
> pvec = 30*plogis(-0.8+0.3*X1)
  inds = order(pvec)           ### needed for plotting lines
  plot(pvec, apply(ymat1,2,mean)-pvec,
         xlab="Theoretical Mean Response for Group",
         ylab="Observed Groupwise Responses")
  abline(0,0)
  title(paste("Plot of Response Rates vs Theoretical",
            "Means & Confidence Bands"), cex=1)
  lines(pvec, -qnorm(0.95)*sqrt(pvec*(1-pvec)/15000), lty=2)
  lines(pvec,  qnorm(0.95)*sqrt(pvec*(1-pvec)/15000), lty=5)
```

1.13.2 *Passing Arguments to Functions within Functions*

So far, our simulation was done by looping within the primary **R** *work-frame*. More complicated operations, or simulations which would be re-done for each of several different parameter-settings, might be organized as the result of a custom-defined function, which itself would call other **R** functions. It turns out there are subtleties concerning passing of arguments to functions within functions. Although this material is very hard to understand on a first or second reading of Venables & Ripley (or any other **R** book !), the issue is that within a k-level nested functions all arguments must be identified either from the ordinary (frame 1) permanent-frame search-list *or* from the temporary frame (frame 0, where **.Options** and **.Devices** reside), *or* from the function-level frame $k + 1$.

```
> Fexamp =
function(nstrat=50, nrep=500, np = 30, beta=c(-0.8,0.3), sig=0)
{
 Xv = rexp(nstrat)
 ym = matrix(rbinom(nstrat * nrep, np, plogis(beta[1] +
        beta[2] * Xv + if(sig > 0) sig * rnorm(nstrat * nrep)
        else 0)), ncol = nstrat, byrow = T)
 outm = t(apply(ym, 1, function(yrow, xvec, nstr0, np0)
        c(glm(cbind(yrow,np0-yrow) ~ xvec, family=binomial)$coef,
        sum(yrow)/(nstr0*np0)), xvec=Xv, nstr0=nstrat, np0=np))
# Note: without this use of ‘apply’, direct use of Xv,
#      nstrat etc would not have been recognized !!
 dimnames(outm) = list(NULL,c("beta1","beta2","ptest"))
 outm
}
```

When the arguments `xvec=Xv`, etc. were dropped in this function within `apply`, and the arguments `np`, `Xv`, and `nstrat` were placed directly into the body of the function-argument of `apply`, the result was

```
> Fexamp(3,100,30)
Error: Object "np" not found
```

There is another way, even harder to digest from the **R** books and help-files but even more generally applicable than `apply`, to pass arguments to functions within functions. If, within the `Fexamp` function, we had defined `outm` as a *nrep* × *nstrat* matrix and then wanted to fill its rows with a *for*-loop, then we could have written

```
  for(j in 1:nrep)
     outm[j, 1:2] = eval(glm(cbind(ym[j,  ], np - ym[j]) ~ Xv,
              family = binomial)$coef, local = sys.parent(1))
```

This command tells **R** to evaluate the `glm` expression by finding variable-names within the *frame* of variables defined within the function-level from which `glm` was called.

1.14 Managing Longer Runs: BATCH and nohup

There are a few **R** and Unix commands which make the submission of longer simulation-runs more manageable, especially if you are logging on remotely and want to run your **R** in background after you have logged off.

These are: `source` and `BATCH` in **R**, and `nohup` in unix.

We begin by illustrating `source`: suppose that you have little text file called `scomm` containing the lines:

```
xx = c(1:37, list(labs=c("AB","BC"), dmat=matrix(1:9, ncol=3)))
cat(7 + 53*exp(-4))
```

Then the **R** command `source("scomm")` causes execution of all commands within the lines of the file in **R**.

Next, for longer 'batch'-style runs, one can use the `R CMD BATCH` command following a Unix prompt. The source-file plays the same role as a file called by the `source()` command. But a file-name to contain the echoed commands and printed output (if any) must also be given — the 'output' file. We illustrate the command together with the use of the Unix `nohup` command.

```
% nohup R CMD BATCH Sstuff/infil.src Sstuff/outfil &
```

1.15 *Why Vectorize ?*

While this issue was much more important in **Splus** than it is is in **R**, the short answer is: to save (a lot of) time. The simplest way to understand the need for vectorizing is to time and compare operations on a large matrix which can be done either by linear algebra versus looping. The specific timing numbers change as computers get faster, but the comparison remains generally valid.

```
> ymat = matrix(runif(500000), ncol=50)
> unix.time({ysum = apply(ymat,1,sum)})
   user  system elapsed
   0.56    0.00    0.56
> unix.time({ysum = apply(ymat,2,sum)})
   user  system elapsed
   0.06    0.00    0.06
> unix.time({ysum = ymat %*% rep(1,50)})
   user  system elapsed
   0.02    0.00    0.01
> unix.time({ysum = rep(1,10000) %*% ymat})
   user  system elapsed
      0       0       0
```

The times given are user, system, and elapsed time in seconds to perform the requested **R** expression-evaluation.

1.15.1 *A Trick to Vectorize Function Evaluations*

We already encountered a use for the numerical-integration routine `integrate`. We were calculating the probability of positive response in a logistic-regression simulation, over all (randomly generated) X values. We take this further by designing a function to provide, for the random-intercept case of the simulation, the probability of response within a cell as a function of the observed explanatory variable X for that cell. We want to calculate

$$\int_{-\infty}^{\infty} plogis(\beta_1 + \beta_2 X + \sigma z)\, e^{-z^2/2} \frac{dz}{\sqrt{2\pi}}$$

and we want its value for a possibly long vector of X values, for fixed $(\beta_1,\, \beta_2,\, \sigma)$. Here is a function to calculate it:

```
> lgstne = function(a, b , acc = 1e-06)  {
# b  must be either scalar or of same length as  a
   bv = if(length(b) < length(a)) rep(b, length(a)) else b
   avec = ifelse(a > 12 + 5 * bv, 1 - exp( - a + bv^2/2) +
      exp(-2 * (a - bv^2)), ifelse(a < -12 - 5 * bv,
      exp(a + bv^2/2) - exp(2 * (a + bv^2)), -1))
   ni = (1:length(a))[avec < 0]
   for(i in ni)
      avec[i] = integrate(function(xp,c1,c2) dnorm(xp) *
      plogis(c1 + c2 * xp)  , -5, 5, rel.tol = acc, c1 = a[i],
      c2 = bv[i])$value
   avec  }
```

In the case where the input argument `a` is a vector, the output of `lgstne` is also a vector, which has been created by a (necessarily slow) for-loop. But the function is very smooth, so we can vectorize the evaluations by using a function defined by **R** as a `smooth.spline` object:

```
> npts = 50
  sig = 0.4
  xv = c( ((-4):(-1)) * 5, -5 + ((1:(npts - 1)) * 10)/npts, (1:4) * 5)
> unix.time({lgsplin = smooth.spline( xv ,
      lgstne(xv, sig), spar = 1e-06, all.knots = T)})
   user  system elapsed
   0.05    0.00    0.10
> unix.time(cat(summary(lgstne(U1,0.4)),"\n"))
0.382 0.4516 0.5432 0.5964 0.7173 0.9943    ### summary line
   user  system elapsed
   0.42    0.00    0.42
> unix.time(cat(summary(predict(lgsplin,U1)$y),"\n"))
0.382 0.4516 0.5432 0.5964 0.7173 0.9944    ### summary
   user  system elapsed
   0.02    0.00    0.01
```

Smoothing splines are actually piecewise polynomial functions closely approximating (a smoothed version of) the designed input set of points (here, the 50 evaluations of `lgstne`). That is why they are so quick to evaluate, and the `smooth.spline` object allows **R** to *vectorize* evaluation at many different new points.