Statistical Computing with **R** & SAS

Eric Slud

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 of the two are very much the same, but **R** is free and at least as capable. Also, in past terms there has been greater emphasis on SAS than there will be now: at present, SAS will be introduced primarily in the context of linear and generalized-linear models, and SAS's treatment of those models will be contrasted with the treatment in **R**.

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

1 Introduction to R

Splus is a so-called *object-oriented language*, which means roughly that it is organized to recognize both inputs and outputs (such as numerical data and fitted statistical models) from standard computer-representations, whch have the structure primarily of *lists with attributes* of several special types. All-encompassing definitions are elusive, but the main idea is that outputs of one stage of analysis can be computed on and then inputted to further stages [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 -Plw2303 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 Splus prompt > . If you want to save everything in the current area as an R workspace (say `NewSapce.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 the command (after the Splus prompt) :

```
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 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 attached lists of *attributes*.

So what are lists made of ? To begin, data organized at its lowest level into strings or vectors, and can be of the following types: 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
> attributes(xvec)
list()
> 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**, **summary**. If they are applied to matrices, the result is the same as if applied to **as.vector** of the matrix.

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 .Data directory).

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 consists 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. 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.

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
16   472353   7034242    O   WH   SP   NH   U
17     8784     72336    R   NW   MP   HI   R
18   135168   1452680    R   NW   MP   HI   U
19    33065    310296    R   NW   MP   NH   R
20   485423   4419920    R   NW   MP   NH   U
21     1631     10205    R   NW   SP   HI   R
22    34662    246480    R   NW   SP   HI   U
23    13796    101362    R   NW   SP   NH   R
24   260072   1861864    R   NW   SP   NH   U
25     9688     74080    R   WH   MP   HI   R
26   133658   1239623    R   WH   MP   HI   U
27   306437   2624507    R   WH   MP   NH   R
28  1204371  11154455    R   WH   MP   NH   U
29     2183     15053    R   WH   SP   HI   R
30    43611    345677    R   WH   SP   HI   U
31   141658   1045221    R   WH   SP   NH   R
32   954350   7948841    R   WH   SP   NH   U
```

For purposes of illustration, 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)) }
```