Computing Primer for Statistics Courses

S-PLUS

Chad M. Schafer, Spring 2001

S-PLUS

S-PLUS is a powerful computer package for the performance of statistical analysis. It can be thought of both as an object-oriented programming language and an interactive platform.

0.0.1 The `.Data` Directory

Being the object-oriented language it is, working with S-PLUS produces a collection of objects. These objects are stored in a so-called `.Data` directory. It is important to keep track of which `.Data` directory is currently in use. An effective strategy is to create separate directories for each project (e.g. lab) and within each of these project directories create a `.Data` directory as follows:

```
computer.username% mkdir .Data
```

S-PLUS will always look first in the current working directory for a `.Data` directory to use. If it does not find one it defaults to using one in the user's home directory.

0.0.2 The Basics of S-PLUS: Using S-PLUS in Interactive Mode

The simplest introduction to S-PLUS consists of learning how to utilize the program in interactive mode. To start S-PLUS interactively, issue the following command at the UNIX prompt:

```
computer.username% Splus
```

The user should then observe

```
S-PLUS : Copyright (c) 1988, 1996 MathSoft, Inc.
S : Copyright AT&T.
Version 3.4 Release 1 for Sun SPARC, SunOS 5.3 : 1996
Working data will be in .Data
>
```

Note the change in the prompt to the `>`, and that S-PLUS indicates the `.Data` directory that is currently being used. Here the `.Data` directory is within the current working directory at the time S-PLUS was started. (An important distinction: Do NOT start S-PLUS while within the `.Data` directory.) Exiting S-PLUS is achieved by issuing

```
> q()
```

There is no need to "save" objects prior to quitting. Any object created is stored in the `.Data` directory and is only removed with the `rm` function described below.

The structure of the above quit command illustrates an important point regarding S-PLUS functions. Just typing

```
> q
```

will output to the screen the content of the function `q`, while placing the parentheses after it will execute the command. In most cases the function call will require arguments to be placed inside these parentheses. Built-in S-PLUS functions should be thought of as being themselves objects, just like those created by the user. The user can build additional functions. This is taken up in a later section.

Typing the name of any S-PLUS object without the parentheses gives the content of that object, but in the case of most built-in functions this is useless. The `help` function provides detailed information, including examples and related S-PLUS functions. For example,

```
> help(q)
```

explains the quit function in detail.

The following functions are routinely used in interactive mode:

- `rm(object.name)` - Deletes the object `object.name`.
- `objects()` - Lists all user-defined objects in the active `.Data` directory.
- `history()` - Lists the past ten issued commands and gives the user the opporunity to reissue any of them.

For each of these commands there are additional arguments which are very useful in certain situations. See the appropriate `help` file for more information.

Data Structures in S-PLUS: Scalars, Vectors, Matrices, and Arrays

The statement

```
> x <- 5
```

defines `x` as a *scalar* with value 5. Note the assignment operator `<-`. A *vector*, or a one-dimensional *array*, can be defined by any of the following commands:

```
> vec <- vector(mode="numeric",length=5)
> vec <- rep(0,5)
> vec <- c(0,0,0,0,0)
```

Each of these examples produces an object named `vec` which is a numeric vector of length 5, filled with zeros. The `mode` argument to the function `vector` can be any one of `logical`, `numeric`, `complex`, or `character`. An example of the definition of a *matrix*, or two-dimensional array, is as follows:

```
> mat <- matrix(0,nrow=3,ncol=5)
```

The object `mat` is now a three by five matrix filled with zeros. The first argument to the `matrix` function can either be a scalar, in which case the matrix is filled with that value, or it can be a vector with length equal to `ncol` × `nrow`. Arrays of dimension higher than two can be formed using the `array` function. This works just as the `matrix` command except instead of giving the number of rows and columns, a vector with length equal to the dimension desired is supplied. For example,

```
> arr <- array(0,c(2,3,4))
```

defines the object `arr` as a three-dimensional array filled with zeros. The sizes of the dimensions are two, three and four, respectively. The functions `length` and `dim` are useful for finding the sizes of vectors, matrices, and arrays.

The use of square brackets after a data structure allows one to reference subsets. The following examples illustrate the possibilities.

`> vec[2]`	The second element of the `vec`.
`> mat[1,2]`	The $(1, 2)$ element of `mat`.
`> mat[,2]`	The second column of `mat`.
`> mat[c(1,2),]`	The first two rows of `mat`.
`> arr[,,3]`	A two by three matrix consisting of all elements of `arr` that have the third index three.

S-PLUS can perform arithmetic operations on data structures of the same size, and utilization of this feature will greatly increase speed and efficiency. The standard arithmetic operators, including `+, -, /, *, sqrt()`, *etc.* work on an *element by element* basis. For example

```
> vec1 + vec2
```

produces a new vector of the same size as `vec1` and `vec2`. Be aware that the operator `*` does not perform the standard matrix multiply; the operator for that is `%*%`. The command

```
> sqrt(mat)
```

returns a matrix which has as its elements the square roots of the elements of `mat`. S-PLUS is capable of performing these operations on data structures of different sizes, provided the number of elements of one is a multiple of the number of elements of the other. This is most often useful in operations with scalars, e.g. multiplying a matrix by a scalar.

A higher-level version of a matrix often used in S-PLUS is a *data frame*. The conversion is performed simply, using

```
> dafr <- data.frame(mat)
```

In most cases the columns of a data frame are thought of as variables, while the rows are observations. The `names` function allows for naming of each of the variables:

```
> names(dafr) <- c("Population","Area","Death Rate","Birth Rate","Literacy Rate")
```

Columns of the data frame can be accessed by name using a `$`, e.g.

```
> dafr$"Death Rate"
```

Regardless, the object `dafr` can still be treated as if it were a matrix.

Reading in Data From a File

The function `read.table` is used for reading data from a file. The output of this function is a data frame. Important arguments to this function include

- `file` - Specifies the file in which the data is found.
- `header` - A logical argument, set to `T` if the first line in the file to be read contains the variable names. Otherwise, set to `F`.
- `skip` - Allows the user to specify a number of lines that should be skipped at the top of the file to be read. For example, many data files contain a header with a description of the data set.
- `sep` - Specifies how the data values are separated within a row of the file. Common settings are `sep=","` and `sep=" "`.

For example, assume we have a file named `datafile.dat` which resides in the directory `/home/username/data` and appears as follows:

```
datafile.dat: This file contains data to be used as an example.
There are four columns, and five rows.
1,3,10,2
5,23,8,6
3,8,2,9
12,10,9,1
4,9,2,10
```

The following command would read in this data file into the data frame `testframe`.

```
> testframe <- read.table(file="/home/username/data/datafile.dat",header=F,skip=2,sep=",")
```

Note that `read.table` assumes that the rows of the data file define rows of the data frame.

Creating Graphs in S-PLUS

When creating a graph the first thing S-PLUS requires is a call to the `motif` function. This creates a window in which the graph will be placed.

```
> motif()
```

The window created in this manner has a menu bar across the top which includes the option for printing the graph. To delete the window, issue the function

```
> dev.off()
```

Multiple windows can be active simultaneously.

One of the simplest and most used graphs is the *scatter plot* comparing two variables. For example, if there are two vectors of equal length named `xvec` and `yvec`, the command

```
> plot(xvec,yvec)
```

will create a scatter plot of `yvec` versus `xvec`. The following list includes additional arguments which can be supplied, along with example values.

- `xlab` - Label to place along the x-axis. `xlab="Year"`
- `ylab` - Label to place along the y-axis. `ylab="Price"`
- `xlim` - Specify the range for the x-axis. `xlim=c(1990,2000)`
- `ylim` - Specify the range for the y-axis. `ylim=c(10,20)`
- `main` - Gives a title to the plot. `main="Price versus Year"`

The function `plot` is itself a very generic function, used to create many types of plots. As a result, the `help` file is not too helpful for any one type.

There are many general aspects of an S-PLUS plot that can be manipulated with the use of the `par` function. The procedure is to issue the `par` command after calling `motif`, but before the call to the graphing function. To illustrate, the default in S-PLUS is to create plots that are wider than they are long. The `par` argument that controls this is `pty`. If square plots are desired, one sets `pty="s"`. For example,

```
> motif()
> par(pty="s")
> plot(xvec,yvec)
```

will result in a square scatter plot of `yvec` versus `xvec`. Another very common use of `par` is to place more than one plot in one graphing window. This is done using `mfrow`. For example,

```
> motif()
> par(mfrow=c(3,2))
> plot(xvec1,yvec1)
> plot(xvec2,yvec2)
     ⋮
> plot(xvec6,yvec6)
```

gives a matrix of six plots in three rows and two columns.

Other often-used graphing functions include `boxplot`, `hist`, and `contour` for creating boxplots, histograms and contour plots, respectively. See the `help` file for details.

Whenever a graph is printed from S-PLUS it is saved in the current working directory in the form of a postscript file. This file will have a name like `ps.out.0001.ps`. Often one wants to only create the postscript file, without wasting paper. There are many ways of doing this but the simplest, although inelegant, way is to choose the option `Printing...` below the `Options` menu in the window of the graph of interest. Then change where it reads `lp` to read `cat > filename.ps`. Upon pressing the button `Print` the user receives a postscript file `filename.ps` in the current working directory, and nothing is sent to the printer. Note that two indentical postscript files are actually created, the other being the `ps.out.000x.ps` file. If the intended destination of the postscript file is a LaTeX document, the plot should be changed from "landscape" to "portrait" mode before being "printed." See the section "Including Postscript Files" above in the LaTeX section for more information.

0.0.3 Using S-PLUS in Batch Mode

In some instances it is useful to be able to run S-PLUS in batch mode, where a series of commands runs without requiring the user's attention. The procedure for this is simple. All of these operations are performed at the UNIX prompt, in the same directory that contains the `.Data` directory to be used. First, create a file which contains the sequence of S-PLUS commands that are to be executed. These commands should be just as if they were typed during an interactive session. Then, give the command

```
computer.username% Splus BATCH commandsfile outfile
```

where `commandsfile` is the aforementioned file with the commands. The output from the series of commands will be sent to the file `outfile`.

In cases where the sequence of commands will be time consuming (often the rationale for using batch mode), special consideration should be made of the other users of the computer system. To ensure that others will have fair access to the computer on which the commands will run, one should alter the above command to be

```
computer.username% nohup nice +19 Splus BATCH commandsfile outfile &
```

Check the `man` pages if more information about `nohup` or `nice` is required. The `&` appended on the end of the command is a general UNIX operator that causes the given command to run in the "background," allowing other work to be done, or for the user to logout completely. See the above section "Running Commands in the Background" in the UNIX section for more information, including how to cancel such jobs.

0.0.4 Writing Functions in S-PLUS

Building new functions gives the user of S-PLUS the ability to "automate" frequently used command sequences. To illustrate the basics of function writing this section will show the steps needed to build a function to be called `tr`, which will return the trace of a given matrix. Note that the built-in S-PLUS function `trace` does something very different.

To edit a new function, simply use the `fix` function as follows:

```
> fix(tr)
```

An important note: if the user wants to use Emacs as the text editor (the default is vi), give the following command at the S-PLUS prompt.

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

To avoid doing this every S-PLUS session, explore the help file for the `.First` function. This produces a new window in which the text editor is running. The skeleton of the function is visible.

```
function()
{
}
```

The set of parentheses following `function` are where the arguments to be passed to the function will be placed. In this case there is only one argument, the matrix of which the trace is to be found. This is to be given the name `holdmat`. It is important to remember that all objects defined within a function are *local* to that function. Thus, if there were an object at the *global* level also called `holdmat`, this will not overwrite it. However, global objects can be referenced from within a function, as long as a local object of the same name has not been created.

```
function(holdmat)
{
}
```

The curly brackets will hold the body of the function. The first step is to verify that the object that the user has passed is indeed a matrix. This requires the use of an `if` function. This function evaluates the argument passed to it, and if it is true the body of statements following it (surrounded by curly brackets) will be performed. S-PLUS has a function `is.matrix` which returns `TRUE` if the object passed to it is a matrix, and `FALSE` otherwise. In addition S-PLUS has `!` as a negation operator. Simply put, if `logical.statement` is `TRUE`, then `!logical.statement` is `FALSE`, and vice-versa. Thus the function is updated as follows:

```
function(holdmat)
{
   if(!is.matrix(holdmat))
   {
      cat("The argument is not a matrix. Exiting.\n")
      return()
   }
}
```

The `cat` function sends the given string to the screen, with the `\n`. appended on the end forcing a carriage return. `return` forces the function to exit immediately.

Next it is necessary to check that the matrix passed in is square. This requires another `if`, as follows:

```
function(holdmat)
{
   if(!is.matrix(holdmat))
   {
      cat("The argument is not a matrix. Exiting.\n")
      return()
   }
   if(nrow(holdmat)!=ncol(holdmat))
   {
      cat("The given matrix is not square. Exiting.\n")
      return()
   }
}
```

The operator `!=` represents "not equal to," whereas `==` is "equal to." The functions `ncol` and `nrow` return the number of columns and rows, respectively, in the matrix.

Finally, the trace can be calculated. This requires the use of a `for` loop. This takes the form

```
for (i in vec) { statements }
```

where `vec` is an arbitrary vector and the body of statements to be performed in the loop are placed inside the curly brackets. These statements are performed a number of times equal to the length of the vector `vec`. The j^{th} time through the loop `i` takes on the value `vec[j]`. Consider the updated function:

```
function(holdmat)
{
   if(!is.matrix(holdmat))
   {
      cat("The argument is not a matrix. Exiting.\n")
      return()
   }
   if(nrow(holdmat)!=ncol(holdmat))
   {
      cat("The given matrix is not square. Exiting.\n")
      return()
   }
```

```
    holdtrace <- 0
    for (i in 1:nrow(holdmat))
    {
       holdtrace <- holdtrace + holdmat[i,i]
    }
    holdtrace
}
```

Note how `holdtrace` is initialized to be zero, and then at each step through the loop the next element of the diagonal is added on.

The last step of the function, shown in the version above, is to pass back the value of the trace. The last line of the function always represents the information that will be passed back to the user of the function. Thus defined, a user can now use this function as follows

```
> traceofX <- tr(X)
```

to obtain the trace of any square matrix.

This function was built for illustrative purposes. A useful exercise would be to try to reduce the number of lines in the function using more built-in S-PLUS functions.

0.0.5 Useful Functions

The following list contains some of the most useful S-PLUS functions. See the `help` files for information on how to use them. In some cases it is necessary to put the function name in quotes when requesting the `help` file.

- *Data Manipulation:* `split`, `apply`, `table`, `factor`
- *Statistical Analysis:* `lm`, `loess`, `anova`, `smooth`, `mean`, `var`
- *Mathematical Functions:* `deriv`, `ms`, `eigen`, `solve`, `t`
- *Function Writing:* `while`, `else`, `list`
- *Input/Output:* `scan`, `sink`

0.0.6 Other Things

Practical Use of S-PLUS

Many users have found that the best way to utilize S-PLUS is to create a text file (e.g. using Emacs) and fill it with a list of the commands needed to complete a given assignment, analysis, *etc.* Then copy and paste the commands from the file into the S-PLUS interactive session. In this way one can check line by line that the commands are working, and still be able to make simple adjustments without retyping everything. Performing statistical analysis and creating graphs often involves a lot of small changes, and keeping the commands in a file will make this easier.

Disk Space with S-PLUS

Using S-PLUS can consume large amounts of disk space. The following steps can help free up space.

1. Keep directories clear of unneeded postscript files.
2. Some objects, such as output from `lm`, can be large. Use `rm` to get rid of unneeded objects.
3. From the UNIX prompt, issue the command

   ```
   computer.username% Splus TRUNC_AUDIT
   ```

to truncate the so-called "audit" file. This will often free up a lot of space. Note that this will only truncate the audit file associated with the `.Data` directory in the current working directory.

4. If netscape is used from the account, keep the cache clear. The cache is located in the directory `~/.netscape/cache/`.

0.0.7 For More Information...

- Spector, Philip. *An Introduction to S and S-PLUS.* 1994. Wadsworth Publishing.
- Venables, Bill and Brian Ripley. *Modern Applied Statistics with S-PLUS.* 1999. Springer Verlag.