8.0 Matrices, Lists and Functions in `Splus`

Notes based on those of Professor Joe Schafer

Main subjects covered today:

- More about functions
- Basic programming constructs (if-then, iteration)
- Principles of efficient programming

8.1 More about functions

First, a note about `ls()`.

In the last lecture, we discussed using `ls()` to see what objects are stored in your workspace (the .Data directory). It turns out that the `ls()` command is rather old-fashioned. The producers of S and Splus now recommend that you use the function `objects()`, which is newer.

If you type

```
> help(ls)
```

you will find the statement: `This function is deprecated; use objects instead.` A deprecated function is a function from older versions of S and Splus that has been replaced by a newer function. Deprecated functions still work, but they may eventually disappear from future releases of the software.

8.2 Strategies for writing functions

Writing functions often involves some trial and error. There are various strategies that you may find helpful.

Strategy 1 (my favorite way): Use a separate editor. Compose your Splus commands in an editor window. As you write the commands, copy and paste them to another window running Splus. Verify that each command works properly as you go along. When you are satisfied that each command works as it should, make a function out of them.

Strategy 2: Use the `fix()` function. This function opens an editor window that allows you to modify an existing function. For example, suppose you already created a function called `plotvec`. Typing

```
> fix(plotvec)
```

will open an editor window that contains the current definition of plotvec. If you have defined your default editor to be Emacs by typing

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

then it will be an Emacs window; otherwise it will be a vi window. Modify the function as you see fit, and then exit the editor in the usual way. Upon exiting, Splus will update the function. (If there are any syntax errors in your code, Splus will notify you at this point.) If you don't already have a function named `plotvec` to modify, then you can easily create one. For example,

```
> plotvec <- function()
+   NULL
```

will create a function that does nothing.

8.3 Default values for arguments

We already learned the basic syntax for creating functions. A function with two arguments (say, x and y) has the following form:

```
> myfunction <- function(x,y)
+   put commands here
+   put value to be returned here
```

The method above assumes that whenever you call the function, you will provide two arguments. If you don't give two arguments, then Splus will give you an error message. Suppose, however, that you had written the function like this:

```
        > myfunction <- function(x,y=default)
+   put commands here
+   put value to be returned here
```

In this situation, you no longer need to provide the second argument when calling the function. If you call the function with only a single argument,

```
> myfunction(x)
```

then Splus will substitute `default` for the missing second argument.

8.4 Conditional (if-then) statements

The basic syntax for conditional statements is:

```
> if(condition) expression
```

Here, `condition` is some kind of expression which, when evaluated, returns a single logical value (T or F). For example, `condition` could be `x>2` where x is a scalar. `Expression` is a single command or a group of commands enclosed by curly brackets. When Splus encounters this statement, it will first evaluate `condition`. If condition=T then Splus will execute `expression`, and if condition=F then Splus does nothing.

The basic if-statement can be extended with `else`:

```
> if(condition) expression1
> else expression2
```

When Splus encounters these statements, it first evaluates `condition`. If condition=T then it executes `expression1`, and if condition=F it executes `expression2`.

If multiple if-statements are present, each `else` will be attached to the most recent `if`. For example:

```
> if(condition1) expression1
> else if(condition2) expression2
> else if(condition3) expression3
> else expression4
```

In this example, if condition1=T then Splus will perform expression1, otherwise it will proceed to evaluate condition2. If condition2 is evaluated then Splus will perform expression2 if condition2=T, otherwise it will proceed to evaluate condition3. If condition3 is evaluated then Splus will perform expression3 if condition3=T, otherwise it will perform expression4.

Here are some concrete examples. Suppose you want to write a function called `matprint` which will print out an object only if it is a

matrix. Such a function might look like this:

```
> matprint <- function(x)
+  if(!is.matrix(x)) stop("Hey dummy! Give me a matrix.")
+  else print(x)
+  invisible()
```

(Note: The function `is.matrix()` returns T if its argument is a matrix and F otherwise. The function `stop()` prints out an error message and stops the function from executing.)

This next version of `matprint` will print out any object, but gives a warning message if the object is not a matrix:

```
> matprint <- function(x)
        +  if(!is.matrix(x))
        +  warning("Hey dummy! This is not a matrix.")
+  print(x)
+  invisible()
```

(Note: The function `warning()` prints out a warning message but does not stop the function from executing.)

8.5 Unconditional iteration

Unconditional iteration means to perform an operation a fixed number of times. With languages like Basic and Fortran, this is often called a "do loop". Unconditional iteration in Splus looks like this:

```
> for(i in vec) expression
```

Here, `i` is a dummy variable, `vec` is a pre-existing vector, and `expression` is a single Splus command or a group of commands enclosed by curly brackets. Splus will create a temporary variable called `i`, set

it equal to vec[1], and perform `expression`; it will then set i equal to vec[2] and perform `expression` again; and so on. In this way, `expression` is performed length(vec) times.

Here's an example. If x is a vector, then

```
> y <- numeric(length(x))
> for(i in 1:length(x)) y[i] <- sqrt(x[i])
```

does exactly the same thing as `y <- sqrt(x)`.

8.6 Conditional iteration

Conditional iteration means to perform an expression repeatedly until some condition is no longer satisfied. It looks like this:

```
> while(condition) expression
```

When Splus encounters this, it first evaluates `condition`. If condition=T it performs `expression` again and again until condition=F. Obviously, `expression` will probably be a group of commands, one of which will eventually modify `condition`; otherwise `condition` will always be T and we will get caught in an infinite loop.

8.7 Principles of efficient programming

As you begin to program in Splus, you will quickly find that there are many ways to perform any given task. Also, you will find that not all ways are equally efficient. There are two principles that you should follow.

(1) Try to minimize the number of expressions evaluated by Splus. Splus is an "interpreted" language, not a "compiled" language. This means that each time Splus encounters an expression, it has to decipher it and convert it to machine-readable microcode. If you write a loop, then Splus has to decipher the expressions within the loop again and again. So you should try to use as few loops as possible. For example, suppose that x is a vector. We can calculate the square roots of all the elements of x with just a single expression:

```
> y <- sqrt(x)
```

This will execute very quickly, even when x is a very long vector. Alternatively, we can write a loop to do the same thing:

```
> y <- rep(0,length(x))
> for(i in 1:length(x)) y[i] <- sqrt(x[i])
```

The second method requires Splus to decipher the expression `y[i] <- sqrt(x[i])` over and over again. If x is a very long vector, this can take a long time.

(2) Try to minimize the number of times memory needs to be allocated. To "allocate memory" means to set aside a block of memory to hold data. Every time you create a new object, Splus has to figure out the size of the object in bytes and set aside that much memory. When you redefine an object and change its size (for example, by saying `x <- sum(x)`), Splus has to de-allocate the memory that held the old version of the object and allocate memory for the new version. If you change a single element of a vector, however (for example, by typing `x[i] <- 2`), then Splus does not need to allocate memory,

because the size and structure of the vector has not changed. Allocating memory takes a lot of time, and so we want to minimize the number of times that this happens. The following example computes the square root of a vector named x. This example is very inefficient, because memory needs to be allocated within each loop:

```
> y <- sqrt(x[1])
> for(i in 2:length(x)) y <- c(y,sqrt(x[i]))
```

It is far more efficient to allocate the memory once at the beginning, like this:

```
> y <- rep(0,length(x))
> for(i in 1:length(x)) y[i] <- sqrt(x[i])
```

Of course, the best way to do it is to eliminate the loop entirely:

```
> y <- sqrt(x)
```

As you gain experience with Splus, you will find that eliminating unnecessary loops and memory allocations can result in enormous savings in computer time.