Part 4: Computing/Programming

140.776 Statistical Computing

Ingo Ruczinski

Thanks to Thomas Lumley and Robert Gentleman of the R-core group (http://www.r-project.org/) for providing some tex files that appear in part in the following slides.

options

```
options                  package:base                  R Documentation

Options Settings

Description:
     Allow the user to set and examine a variety of global "options"
     which affect the way in which R computes and displays its results

Usage:
     options(...)
     getOption(x)
     .Options

Arguments:
     ...: any options can be defined, using 'name = value'.
          However, only the ones below are used in "base R".
          Further, 'options('name') == options()['name']', see the
          example.
       x: a character string holding an option name.
```

Customizing your R environment

Creating a `.Renviron` file in your root directory is a good idea. For example, when you use the `postscript` command to create a figure, the default for the paper format is 'A4'. You can change it to format 'letter' by typing

```
postscript("test.ps",paper="letter")
```

but you had to do this every time you want to create a postscript file. However, if you put the line

```
R_PAPERSIZE=letter
```

into your `.Renviron` file, R will use the 'letter' format as default. And it works in all subdirectories!

Customizing your R environment

Creating a `.Rprofile` file in your root directory is an equally good idea. For example, the commands

```
load("/home/iruczins/code/R/functions/.RData")
options(width=200,defaultPackages=c(getOption("defaultPackages"),
"nlme","rpart","survival","tree"))
```

ensure that my own R functions are available when I start an R session, that the width of the screen output is 200 characters wide, and that the default libraries plus the libraries `nlme`, `rpart`, and `survival` are loaded. Works in all subdirectories!

Type `?Startup` to see what exactly R is doing.

The BATCH mode

The command `source` reads in R code, but it is generally not a good idea to use it to run large simulations. This should be done using the `R BATCH` mode:

```
R BATCH inputfile outputfile &
```

In R, you have to use the & to run the job in the background. In Splus, this was not necessary.

If you run jobs on a cluster and share CPU time with other users, don't forget to `nice` them!

`system`

The function `system` allows you execute Unix/Linux commands inside an R session. For example,

```
> system("ls")
> system("pwd")
> system("ping www.google.com")
```

lists the files in your current directory, shows you the path, and 'pings' Google.

The function `system.time` (see also `proc.time`) lets you determines how much time the currently running R process has already consumed.

Debugging

R has some built-in program debugging tools. Check out the help files for:

- `browser`
- `debug`
- `trace`
- `traceback`
- `recover`

Defining functions

A function definition looks like

```
median <- function(x, na.rm = FALSE)
{
 ... lots of code...
 ## a return value
 sort(x, partial = half)[half]
}
```

This function has two arguments, `x` and `na.rm`. The second argument has a default value of FALSE, so it is optional. The first argument is required.

The last line of the function computes a return value, which is not assigned to anything.

Functions

```
std.dev = function(x) sqrt(var(x))

t.test.p = function(x,mu=0) {
  n=length(x)
  tv=sqrt(n)*(mean(x)-mu)/std.dev(x)
  return(2*(1-pt(abs(tv),n-1)))
}

t.stat = function(x,mu=0) {
  n=length(x)
  tv=sqrt(n)*(mean(x)-mu)/std.dev(x)
  list(t=tv,p=2*(1-pt(abs(tv),n-1)))
}

> z=rnorm(300,1,2)
> t.stat(z)
$t
[1] 8.019781

$p
[1] 2.420286e-14
```

Function arguments

Suppose the function is called as

```
median(height*width)
```

and `height` and `width` are `c(1,2,3)` and `10` respectively.

- The function argument `x` is now the value of `height*width`, `c(10,20,30)`. It still has this value even if there are new variables `height` and `width` defined inside the function.
- The argument `na.rm` has its default value `FALSE`, since its value wasn't specified.

Example

We could complete the `median` function as

```
median <- function (x, na.rm = FALSE){
    if (na.rm) {remove missing values}
        x <- x[!is.na(x)]
    else if (any(is.na(x)))
        return(NA)
    n <- length(x)
    half <- (n + 1)/2
    if (n%%2 == 1) { ## odd n
        sort(x)[half]
    }
    else { ## even n
        sum(sort(x)[c(half, half+ 1)])/2
    }
}
```

It sorts `x` and then returns the middle observation or the average of the middle two.

Scope

There may be variables with the same name (eg `x`, `n`) inside and outside a function. Rules for working out which one to use are called scoping rules. R's are:

- First look for a variable inside the function.
- Then look for a variable inside the function that the function was defined in (if any), and so on up.
- Finally look in the global environment, the variables visible at the command line.

This is different from S, where the second step doesn't happen, but the difference only matters in some specialized cases. The third step is usually accidental for looking up variables (it's important for looking up functions).

Scope example

The value of this function is itself a function

```
power <- function(lambda){
  function(x) {x^lambda}
}

square <- power(2)
cube <- power(3)

> square(1:2)
[1] 1 4

> cube(1:2)
[1] 1 8
```

Inside the `square` function, what is `lambda`? It isn't a local variable, so R looks at the function where `square` was defined. Here `lambda` exists. Its value was 2.

Unevaluated arguments

Earlier we said that in

```
median(height*width)
```

the function argument x just stored the value of `height*width`. This isn't quite true. Until you look at x it stores the whole expression `height*width`. Graphics commands use this to get plot labels, since the `substitute` function lets you copy the expression without looking at it.

```
> label <- function(x) {list(value=x,actual=substitute(x))}
> label(1+1)
$value
[1] 2
$actual
1 + 1
```

This lazy evaluation is also used in handling model formulas.

Avoiding iteration

The canonical bad R program looks like this:

```
## multiply two vectors
for(i in 1:n){
  d[i]<-a[i]*b[i]
}

## compute the inner product
s<-0
for(i in 1:n){
  s<-s+d[i]
}
```

The right way to do this is:

```
s<-sum(a*b)
```

Multiplication, like many operations and functions, is vectorized: it works elementwise on whole objects.

Why avoid iteration?

There are two reasons to replace loops with vectorized calculations:

- Speed: the `for()` loop is much slower since the expression must be evaluated by the interpreter every time.
- Clarity: it is much easier to see what `sum(a*b)` does.

Vectorized functions

These include

- The operators &, `|`, `!`, `+`, `-`, `*`, `/`, `^`, %%.
- Mathematical functions such as `log`, `sin`, `pnorm`, `choose`, `gamma`, and many more.
- Random number generators such as `rnorm`, `rpois`, `...`
- `ifelse` for vectorized conditionals.

Vector recycling

Recall the recycling rules:

- If a and b are vectors of the same length n, then `a*b` is the element-by-element product.
- `2*b` should be the vector whose elements are twice those of b, so the 2 must be repeated n times.
- Generalizing this, the shorter of the two arguments is always repeated to make it as long as the longer argument. If this is not an exact multiple, a warning is given.

Matrices and vectors

One of the few examples of vector recycling where the shorter vector isn't of length 1 is in vector-matrix operations.

```
> a <- matrix(1:12,nrow=3)
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

>  b <- c(1,10,100)

> a*b
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]   20   50   80  110
[3,]  300  600  900 1200
```

Matrices and vectors

If a is a matrix and b is a vector then `a*b` multiplies each column of a by b.

This works because a matrix is stored as a vector with the columns stacked on each other. Vector recycling produces a separate copy of b for each column of a.

A common application is in regression models that involve computing WX, for a design matrix X and diagonal weight matrix X.

The code in `lm` and `glm` uses `X*wts` where X is the design matrix and `wts` is the vector diagonal of W.

Matrix multiplication

The `*` operator performs elementwise multiplication, so another operator `%*%` is needed for matrix multiplication. Together with `t()` for matrix transposition this allows many statistical formulas to be written without loops.

$$\hat{\beta} = (X^T W X)^{-1} X^T W Y$$

could be written as

```
betahat <- solve(t(X)%*%(w*X))%*%t(X)*(w*Y)
```

However, the QR decomposition approach

```
whalf <- sqrt(w)
betahat <- solve(whalf*X,whalf*y)
```

is faster, more accurate, and uses less memory!

Loops

The main loop construct in R is `for`. The commonest use, as in C and other languages, is to count from 1 to n.

```
for(i in 1:n){
  ##do something
}
```

But `for()` can iterate over any sequence:

```
for(i in (1:10)*4)
for(j in c(3,1,4,1,5,9,2,7))
for(variable in names(data))
for(f in c(sin,cos,tan))
```

Unobvious facts about `for()`

- If the sequence has length zero, the body of the loop is never executed:

  ```
  n <- 0
  for(i in seq(length=n)){
     print(i)
  }
  ```

  Note that `i in 1:n` would be wrong here. `1:0` is the sequence `1,0`. The command `seq(length=n))` should be used defensively in programming.

- In R the loop counter variable exists after the loop finishes:

  ```
  for(i in 1:10){
    ## do something
  }
  print(i)  ## 10
  ```

  In Splus the variable doesn't exist after you leave the loop.

Leaving loops

The `break` and `next` commands allow the flow of a loop to be altered:

- `break` jumps out of the loop. For example, in `glm.fit`:

  ```
   if (abs(dev - devold)/(0.1 + abs(dev)) < control$epsilon) {
      conv <- TRUE
      break
    }
  ```

  jumps out of an iterative optimization when the optimum is (basically) reached.

- `next` jumps to the next iteration of the loop.

Neither of these is very commonly used though.

Other loops

- `while()` repeats an expression while a condition is true. Nearly all the occurences in base R could equally well use `for()`, but `while` may be used to emphasize that the loop is intended to `break` rather than finish.
- `repeat()` repeats an expression forever, so `break` or an interrupt is needed to terminate it. This is not used at all in base R, but can be useful for demos.

A "why not loop?" example

Suppose a is a vector of 5 column numbers, indexing a large dataframe d, and you want to compute the mean of each of the indicated columns.

Since `for()` can iterate over anything, you begin

```
means <- numeric(length(a))
for(i in a){
   means[i]<-mean(d[,i]) ## wrong!
}
```

This doesn't quite work, because `means` is only 5 entries long, and `i` may be larger than 5.

A “why not loop?” example

Try again, using explicit indexing:

```
means <- numeric(length(a))
for(i in seq(length=length(a))){
   means[i] <- mean(d[,a[i]])
}
```

This works perfectly well, but is a bit ugly.

A better solution is

```
means <- sapply(a,function(i) mean(d[,i]))
```

or

```
means <- apply(d[,a],2,mean)
```

The `apply` commands

`lapply`, `sapply`, `apply`, `tapply` all replace loops that iterate over entries or collections of entries in an object, computing the same function on each.

This style of programming is unfamiliar to most people who haven’t used Lisp. It has the advantage of making it clear to the reader (and the computer) that you are doing the same thing to each entry, and that the order of the computations doesn’t matter.

The increased clarity should make the program easier to read and write. In theory it could make it faster, and in fact `lapply` and `sapply` are a bit faster than the corresponding loops.

lapply, sapply

These two are very similar:

- `lapply(x,FUN,...)` applies FUN to each element of `list`. Other arguments to FUN can be supplied, and returns the list of results.
- `sapply` works the same way, but simplifies the result to a vector or matrix if possible.

```
## From anova.lm, 'objects' is a list of models
ns <- sapply(objects, function(x) length(x$residuals))
if(any(ns != ns[1]))
    stop("models were not all fitted to the same size of dataset")
```

apply

`apply()` applies a function to slices of an array (eg columns of a matrix). The syntax is a little tricky: `apply(X,MARGIN,FUN)` where:

- X is an array,
- FUN is the function to apply,
- `MARGIN` is the dimensions that will be kept, `1` is rows, `2` is columns and so on.

```
> data(trees)

> apply(trees,2,mean)
   Girth   Height   Volume
13.24839 76.00000 30.17097
```

returns the mean of each column. This is easy in two dimensions, harder in three.

3-d array example

```
> data(iris3) ## Anderson’s iris data, in 3-d array
> dimnames(iris3)
[[1]]
NULL

[[2]]
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."

[[3]]
[1] "Setosa"     "Versicolor" "Virginica"

> apply(iris3,c(2,3), mean) ## keep the variable and species
         Setosa Versicolor Virginica
Sepal L.  5.006      5.936     6.588
Sepal W.  3.428      2.770     2.974
Petal L.  1.462      4.260     5.552
Petal W.  0.246      1.326     2.026
```

`tapply`

`tapply` applies a function to the cells of an implicit table defined by one or more factors. One example is creating a table from already grouped data.

```
> data(esoph)
> tapply(esoph$ncases,esoph$agegp,sum)
25-34 35-44 45-54 55-64 65-74   75+
    1     9    46    76    55    13
## could also do xtabs(ncases~alcgp+agegp,data=esoph)

> tapply(esoph$ncases,list(esoph$agegp,esoph$alcgp),sum)
      0-39g/day 40-79 80-119 120+
25-34         0     0      0    1
35-44         1     4      0    4
45-54         1    20     12   13
55-64        12    22     24   18
65-74        11    25     13    6
75+           4     4      2    3
```

`tapply` (continued)

A common data analysis task is to compute tables of means and variances for subsets of the data.

```
> data(colon)
> tapply(colon$age,list(colon$rx,colon$extent), mean)
           1        2        3        4
Obs     61.0 60.28947 59.58635 55.60000
Lev     53.0 60.25000 60.16988 60.25000
Lev+5FU 55.3 61.90625 59.48606 62.18182

> tapply(colon$age,list(colon$rx,colon$extent),
         function(x) sqrt(var(x)))
                1        2        3         4
Obs     10.745542 13.04742 11.90884 10.565546
Lev     15.257785 10.59730 11.88737  7.531095
Lev+5FU  8.285053 12.84921 12.20495 13.492984
```

Speed and apply functions

The `apply` functions in Splus were at one time substantially faster than the corresponding loops, so people often think of them as a speed optimization.

In fact they may not be much faster in any of the S implementations, and they should really be considered as clarity optimizations.

Optimizing for speed is a separate topic.

sweep

```
sweep                   package:base                   R Documentation

Sweep out Array Summaries

Description:
     Return an array obtained from an input array by sweeping out a
     summary statistic.

Usage:
     sweep(x, MARGIN, STATS, FUN="-", ...)

Arguments:
       x: an array.
  MARGIN: a vector of indices giving the extents of ’x’ which
          correspond to ’STATS’.
   STATS: the summary statistic which is to be swept out.
     FUN: the function to be used to carry out the sweep.
     ...: optional arguments to ’FUN’.
```

sweep

```
> data(attitude)
> attitude
   rating complaints privileges learning raises critical advance
1      43         51         30       39     61       92      45
2      63         64         51       54     63       73      47
3      71         70         68       69     76       86      48
4      61         63         45       47     54       84      35
5      81         78         56       66     71       83      47
...

# subtract the column medians
> med.att <- apply(attitude, 2, median)
> sweep(data.matrix(attitude), 2, med.att)
   rating complaints privileges learning raises critical advance
1   -22.5        -14      -21.5    -17.5   -2.5     14.5       4
2    -2.5         -1       -0.5     -2.5   -0.5     -4.5       6
3     5.5          5       16.5     12.5   12.5      8.5       7
4    -4.5         -2       -6.5     -9.5   -9.5      6.5      -6
5    15.5         13        4.5      9.5    7.5      5.5       6
...
```