STA122 Lab Session # 4: Flow Control in R

Course Instructor: Prof. Merlise Clyde
Teaching Assistant: Debdeep Pati (dp55@stat.duke.edu)

February 3, 2009

1 Grouping, loops and conditional execution

1.1 Grouped expressions

R is an expression language in the sense that its only command type is a function or expression which returns a result. Even an assignment is an expression whose result is the value assigned, and it may be used wherever any expression may be used; in particular multiple assignments are possible. Commands may be grouped together in braces, $expr_1; ...; expr_m$, in which case the value of the group is the result of the last expression in the group evaluated. Since such a group is also an expression it may, for example, be itself included in parentheses and used a part of an even larger expression, and so on.

1.2 Control statements

1.2.1 Conditional execution: if statements

The language has available a conditional construction of the form

```
> if (expr_1) expr_2
 else expr_3
```

where `expr_1`must evaluate to a single logical value and the result of the entire expression is then evident. The short-circuit operators `&&` and `||`are often used as part of the condition in an if statement. Whereas `&`and `|`apply element-wise to vectors, `&&` and `||`apply to vectors of length one, and only evaluate their second argument if necessary. There is a vectorised version of the if/else construct, the ifelse function. This has the form `ifelse(condition,`a, b) and returns a vector of the length of its longest argument, with elements a[i] if condition[i] is true, otherwise b[i].

1.2.2 Repetitive execution: for loops, repeat and while

There is also a for loop construction which has the form

```
> for (name in expr_1) expr_2
```

where name is the loop variable. `expr_1` is a vector expression, (often a sequence like 1 : 20), and `expr_2` is often a grouped expression with its sub-expressions written in terms of the dummy name. `expr_2` is repeatedly evaluated as name ranges through the values in the vector result of `expr_1.`As an example, suppose ind is a vector of class indicators and we wish to produce separate plots of y versus x within classes. One possibility here is to use `coplot()`which will produce an array of plots corresponding to each level of the factor. Another way to do this, now putting all plots on the one display, is as follows:

```
> xc <- split(x, ind)
> yc <- split(y, ind)
> for (i in 1:length(yc)) {
plot(xc[[i]], yc[[i]]) abline(lsfit(xc[[i]], yc[[i]])) }
```

Note the function `split()`which produces a list of vectors obtained by splitting a larger vector according to the classes specified by a factor. This is a useful function, mostly used in connection with boxplots. See the help facility for further details.

2 Examples

Consider, for instance, the following code. (It is not terribly important what the code does, but it implements a version of Newtons method for calculating the square root of y.)

```
> y <- 12345
> x <- y/2
> while (abs(x*x-y) > 1e-10) x <- (x + y/x)/2
> x
[1] 111.1081
> x^2
[1] 12345
```

Notice the while(condition) expression construction, which says that the expression should be evaluated as long as the condition is TRUE. The test occurs at the top of the loop so that the expression might never be evaluated. A variation of the same algorithm with the test at the bottom of the loop can be written with a repeat construction:

```
> x <- y/2
> repeat{
+ x <- (x + y/x)/2
+ if (abs(x*x-y) < 1e-10) break
+ }
> x
[1] 111.1081
```

Actually, while and repeat are quite rarely used in R. Much more frequent is for, which loops over a fixed set of values as in the following example, which plots a set of power curves on the unit interval.

```
> x <- seq(0, 1,.05)
> plot(x, x, ylab="y", type="l")
> for ( j in 2:8 ) lines(x, x^j)
```

Notice the loop variable j, which in turn takes the values of the given sequence when used in the lines call.

3 The Normal-Bayes Example

```
spf <- read.table("ex0430.csv",sep=",", header=T)
attach(spf)
```

```
Y=log(SUNSCREEN/CONTROL)
ybar = mean(Y)
s2 = var(Y)
n = length(Y)

solve.HPD.logt = function(y, frac, max = 4,plot=T, ...){
n = length(y)
ybar=mean(y)
s2 = var(y)
dlogt = function(x){
sqrt(n/s2)*dt((log(x)-ybar)/sqrt(s2/n),n-1)/x}
  maxx = ybar -n/2 + sqrt(n^2/4 - s2*(n+1)/n)
#  maxx = ybar #  print(c(ybar, maxx))
  maxh = dlogt(exp(maxx))
#  maxh = dlogt(exp(ybar))
  h = frac*maxh
  lt = uniroot(f=function(x){dlogt(x) - h},
               lower=exp(ybar - 6*sqrt(s2/n)),upper=exp(maxx))$root
  ut = uniroot(f=function(x){ dlogt(x)- h},
               lower=exp(maxx),  upper=exp(ybar + 6*sqrt(s2/n)))$root
   coverage = pt((log(ut) - ybar)/sqrt(s2/n), n-1) -  pt((log(lt) - ybar)/sqrt(s2/n), n-1)
  if (plot) {
    z = seq(-max, max, length=500)
    mu = exp(z*sqrt(s2/n) + ybar)
    plot(mu, dlogt(mu),
         t="l", lty=1,xlab=expression(exp(mu)),
         ylab="posterior Density", ...)
    abline(h=h)
    segments(ut,0,ut,dlogt(ut))
    segments(lt,0,lt,dlogt(lt))
    title(bquote(paste("p(", .(round(lt, 2))," < ", exp(mu), " < ",
                       .(round(ut,2)), " | " , y, ") = ",
                       .(round(coverage, 2)), ")")))
  }
return(c(lt,ut,coverage,h)) }

solve.HPD.logt(Y, frac = .1)
```