A Tutorial on R with Examples

Longhai Li

Department of Mathematic and Statistics
University of Saskatchewan
106 Wiggins Road, MCLN 219
Saskatoon, SK, S7N 5E6
Email: longhai@math.usask.ca

January 2008

1 Invoking R

1.1 From unix terminal

`R [options] [<infile] [>outfile]`
or
`R CMD BATCH infile`
Explanations:

- Without specifying anything in `[ ]`, R will be opened from a terminal, waiting for inputing R commands. Output will be printed on screen. To exit from R, type `q()`, then you will be asked whether to save the image or not. If choose to save, the objects created will be saved in file ".RData", and all the commands you have input so far will be saved in file ".Rhistory". Note: these two files are hidden files.
- If "infile" is given, R will execute the R commands in "infile" and output is written to "outfile" or printed on screen if it is empty. You must specify in `[options]` whether to save objects: `--save,--no-save`.

  Demonstration:

  A file "demo-startup.R" is shown as follows:

```
a <- matrix(1:8,2,4)

a

b <- matrix(1:8*0.1,2,4)

b

a + b
```

  Run: `R --no-save --quiet < demo-startup.R > demo-startup.Rout`

  It generates a file named "demo-startup.Rout", shown as follows:

```
> a <- matrix(1:8,2,4)
>
> a
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
>
> b <- matrix(1:8*0.1,2,4)
>
> b
     [,1] [,2] [,3] [,4]
[1,]  0.1  0.3  0.5  0.7
[2,]  0.2  0.4  0.6  0.8
>
> a + b
     [,1] [,2] [,3] [,4]
[1,]  1.1  3.3  5.5  7.7
[2,]  2.2  4.4  6.6  8.8
>
```

- Alternatively, one can type the following commands in R console for running all commands in the file "demo-startup.R":
  `source("demo-startup.R",echo=TRUE).`
  The above content in the file "demo-startup.Rout" will be printed on screen.
- `R CMD BATCH file.R` is similar to
  `R --save < file.R > file.Rout`, but writting all messages printed on screen in "file.Rout", including the error messages.
- Run programs in the background using `nohup`, eg:
  `nohup R BATCH infile &`

1.2 From Windows

- Using Windows command lines: adding the directory containing R.exe to the environment variable `path`. You can run R in the same way as using Unix terminal. But `nohup` facility may not exist.
- Click R on the desktop to start up. You need to change the working directory for the first time. After you save image (file ".RData" will be created), for the second time, click the file ".RData" will open R from this directory.

2 Getting help

- From R console, type `?keyword`, or `help.search(keyword)`
- From internet: http://cran.r-project.org

3 Objects and operations

3.1 Numbers and vector

Below I use example to demonstrate how to use numeric vector.

```
> a <- 1
>
> a
[1] 1
>
> b <- 2:10
>
> b
[1]  2  3  4  5  6  7  8  9 10
>
> #concatenate two vectors
> c <- c(a,b)
>
> c
 [1]  1  2  3  4  5  6  7  8  9 10
>
> #vector arithmetics
> 1/c
 [1] 1.0000000 0.5000000 0.3333333 0.2500000 0.2000000 0.1666667 0.1428571
 [8] 0.1250000 0.1111111 0.1000000
>
> c^2
 [1]   1   4   9  16  25  36  49  64  81 100
>
> c^2 + 1
 [1]   2   5  10  17  26  37  50  65  82 101
>
> #apply a function to each element
> log(c)
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
>
> sapply(c,log)
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
>
>
> #operation on two vectors
> d <- (1:10)*10
>
> d
 [1]  10  20  30  40  50  60  70  80  90 100
>
> c + d
 [1]  11  22  33  44  55  66  77  88  99 110
>
> c * d
 [1]   10   40   90  160  250  360  490  640  810 1000
```

```
>
> d ^ c
 [1] 1.000000e+01 4.000000e+02 2.700000e+04 2.560000e+06 3.125000e+08
 [6] 4.665600e+10 8.235430e+12 1.677722e+15 3.874205e+17 1.000000e+20
>
> #more concrete example: computing variance of 'c'
> sum((c - mean(c))^2)/(length(c)-1)
[1] 9.166667
>
> #of course, there is build-in function for computing variance:
> var(c)
[1] 9.166667
>
> #subsetting vector
> c
 [1]  1  2  3  4  5  6  7  8  9 10
>
> c[2]
[1] 2
>
> c[c(2,3)]
[1] 2 3
>
> c[c(3,2)]
[1] 3 2
>
> c[c > 5]
[1]  6  7  8  9 10
>
> #let's see what is "c > 5"
> c > 5
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
>
> c[c > 5 & c < 10]
[1] 6 7 8 9
>
> c[as.logical((c > 8) + (c < 3))]
[1]  1  2  9 10
>
> log(c)
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
>
> c[log(c) < 2]
[1] 1 2 3 4 5 6 7
>
> #modifying subset of vector
> c[log(c) < 2] <- 3
```

```
>
> c
 [1]  3  3  3  3  3  3  3  8  9 10
>
> #extending and cutting vector
> length(c) <- 20
>
> c
 [1]  3  3  3  3  3  3  3  8  9 10 NA NA NA NA NA NA NA NA NA NA
>
> c[25] <- 1
>
> c
 [1]  3  3  3  3  3  3  3  8  9 10 NA NA NA NA NA NA NA NA NA NA NA NA NA NA  1
>
> #getting back original vector
> length(c) <- 10
> c
 [1]  3  3  3  3  3  3  3  8  9 10
>
> #introduce a function ‘‘seq’’
> seq(0,10,by=1)
 [1]  0  1  2  3  4  5  6  7  8  9 10
>
> seq(0,10,length=20)
 [1]  0.0000000  0.5263158  1.0526316  1.5789474  2.1052632  2.6315789
 [7]  3.1578947  3.6842105  4.2105263  4.7368421  5.2631579  5.7894737
[13]  6.3157895  6.8421053  7.3684211  7.8947368  8.4210526  8.9473684
[19]  9.4736842 10.0000000
>
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
>
> #seq is more reliable than ":"
> n <- 0
>
> 1:n
[1] 1 0
>
> #seq(1,n,by=1)
> #Error in seq.default(1, n, by = 1) : wrong sign in ’by’ argument
> #Execution halted
>
> #function ‘‘rep’’
> c<- 1:5
>
> c
[1] 1 2 3 4 5
```

```
>
> rep(c,5)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
>
> rep(c,each=5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5
>
```

3.2 Missing values

```
> a <- 0/0
>
> a
[1] NaN
>
> is.nan(a)
[1] TRUE
>
> b <- log(0)
>
> b
[1] -Inf
>
> is.finite(b)
[1] FALSE
>
> c <- c(0:4,NA)
>
> c
[1]  0  1  2  3  4 NA
>
> is.na(c)
[1] FALSE FALSE FALSE FALSE FALSE  TRUE
>
```

3.3 Character vectors

Character strings are entered using either double (") or single (') quotes, but are printed using double quotes (or sometimes without quotes). They use C-style escape sequences, using \ as the escape character, so \\ is entered and printed as \\, and inside double quotes " is entered as \". Other useful escape sequences are \n, newline, \t, tab and \b, backspace.

```
> A <- c("a","b","c")
>
> A
[1] "a" "b" "c"
>
> paste("a","b",sep="")
[1] "ab"
```

```
>
> paste(A,c("d","e"))
[1] "a d" "b e" "c d"
>
> paste(A,10)
[1] "a 10" "b 10" "c 10"
>
> paste(A,10,sep="")
[1] "a10" "b10" "c10"
>
> paste(A,1:10,sep="")
 [1] "a1"  "b2"  "c3"  "a4"  "b5"  "c6"  "a7"  "b8"  "c9"  "a10"
>
```

3.4 Matrice

```
> A <- matrix(0,4,5)
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0
[4,]    0    0    0    0    0
>
> A <- matrix(1:20,4,5)
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
>
>
> #subsectioning and modifying subsection
>
> A[c(1,4),c(2,3)]
     [,1] [,2]
[1,]    5    9
[2,]    8   12
>
> A[c(1,4),c(2,3)] <- 1
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1   13   17
[2,]    2    6   10   14   18
```

```
[3,]    3    7   11   15   19
[4,]    4    1    1   16   20
>
> A[4,]
[1]  4  1  1 16 20
>
> A[4,,drop = FALSE]
     [,1] [,2] [,3] [,4] [,5]
[1,]    4    1    1   16   20
>
> #combining two matrices
>
> #create another matrix using another way
> A2 <- array(1:20,dim=c(4,5))
>
> A2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
>
> cbind(A,A2)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1   13   17    1    5    9   13    17
[2,]    2    6   10   14   18    2    6   10   14    18
[3,]    3    7   11   15   19    3    7   11   15    19
[4,]    4    1    1   16   20    4    8   12   16    20
>
> rbind(A,A2)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    1    1   16   20
[5,]    1    5    9   13   17
[6,]    2    6   10   14   18
[7,]    3    7   11   15   19
[8,]    4    8   12   16   20
>
> #operating matrice
>
> #transpose matrix
> t(A)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    1    6    7    1
[3,]    1   10   11    1
```

```
[4,]   13   14   15   16
[5,]   17   18   19   20
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    1    1   16   20
>
> A + 1
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    2    2   14   18
[2,]    3    7   11   15   19
[3,]    4    8   12   16   20
[4,]    5    2    2   17   21
>
> x <- 1:4
>
> A*x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1   13   17
[2,]    4   12   20   28   36
[3,]    9   21   33   45   57
[4,]   16    4    4   64   80
>
> #the logical here is coercing the matrix "A" into a vector by joining the column
> #and repeat the shorter vector,x, as many times as making it have the same
> #length as the vector coerced from "A"
>
> #see another example
>
> x <- 1:3
>
> A*x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3   13   34
[2,]    4   18   10   28   54
[3,]    9    7   22   45   19
[4,]    4    2    3   16   40
>
> A^2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1  169  289
[2,]    4   36  100  196  324
[3,]    9   49  121  225  361
[4,]   16    1    1  256  400
>
```

```
> A <- matrix(sample(1:20),4,5)
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    7    1    9    3   20
[2,]   11   10   12   15    4
[3,]    6   16    5    8   19
[4,]    2   17   14   13   18
>
> B <- matrix(sample(1:20),5,4)
>
> B
     [,1] [,2] [,3] [,4]
[1,]   11   13    3   16
[2,]    2    4    7   10
[3,]    9   12    1   14
[4,]    5   15   17    8
[5,]   19    6   20   18
>
> C <- A %*% B
>
> C
     [,1] [,2] [,3] [,4]
[1,]  555  368  488  632
[2,]  400  576  450  636
[3,]  544  436  651  732
[4,]  589  565  720  826
>
> solve(C)
              [,1]         [,2]        [,3]        [,4]
[1,]  0.0113721434 -0.011305781 -0.03593289  0.03184764
[2,]  0.0037070714 -0.006309519 -0.03288573  0.03116506
[3,] -0.0008241436 -0.012304505 -0.02598824  0.03313549
[4,] -0.0099265186  0.023103180  0.07077051 -0.07169985
>
> #solving linear equation
>
> x <- 1:4
>
> d <- C %*% x
>
> solve(C,d)
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
>
```

```
> #altenative way (but not recommended)
> solve(C) %*% d
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
>
> #SVD (C = UDV') and determinant
>
> svd.C <- svd(C)
>
> svd.C
$d
[1] 2332.552515  204.076932   98.799790    7.614026

$u
           [,1]        [,2]       [,3]       [,4]
[1,] -0.4430091  0.41432949  0.7886463  0.1005537
[2,] -0.4432302 -0.84065092  0.2206557 -0.2194631
[3,] -0.5143024  0.34835579 -0.3848769 -0.6826500
[4,] -0.5854766  0.01689212 -0.4256969  0.6897202

$v
           [,1]        [,2]        [,3]       [,4]
[1,] -0.4492025  0.45643331  0.66652400  0.3816170
[2,] -0.4172931 -0.83456029  0.09103668  0.3479769
[3,] -0.5024522  0.30793204 -0.73787822  0.3290219
[4,] -0.6096108 -0.01885767  0.05471573 -0.7905854

>
> #calculating determinant of C
>
> prod(svd.C$d)
[1] 358092916
>
>
```

3.5 List

An R list is an object consisting of an ordered collection of objects known as its components.
There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on.

```
> a <- 1:10
>
> b <- matrix(1:10,2,5)
```

```
>
> c <- c("name1","name2")
>
> alst <- list(a=a,b=b,c=c)
>
> alst
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$b
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

$c
[1] "name1" "name2"

>
> #refering to component of a list
>
> alst$a
 [1]  1  2  3  4  5  6  7  8  9 10
>
> alst[[2]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
>
> blst <- list(d=2:10*10)
>
> #concatenating list
> ablst <- c(alst,blst)
>
> ablst
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$b
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

$c
[1] "name1" "name2"

$d
[1]  20  30  40  50  60  70  80  90 100
```

```
>
```

A list is usually used to return the results of a large program, for example those of a linear regression fitting.

3.6 Data frames

A data frame may for many purposes be regarded as a matrix with columns possibly of differing modes and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions.

```
> name <- c("john","peter","jennifer")
>
> gender <- factor(c("m","m","f"))
>
> hw1 <- c(60,60,80)
>
> hw2 <- c(40,50,30)
>
> grades <- data.frame(name,gender,hw1,hw2)
>
>
> grades
      name gender hw1 hw2
1     john      m  60  40
2    peter      m  60  50
3 jennifer      f  80  30
>
> #subsectioning a data frame
>
> grades[1,2]
[1] m
Levels: f m
>
> grades[,"name"]
[1] john     peter    jennifer
Levels: jennifer john peter
>
> grades$name
[1] john     peter    jennifer
Levels: jennifer john peter
>
> grades[grades$gender=="m",]
   name gender hw1 hw2
1  john      m  60  40
2 peter      m  60  50
>
> grades[,"hw1"]
```

```
[1] 60 60 80
>
> #divide the subjects by "gender", and calculating means in each group
> tapply(grades[,"hw1"], grades[,"gender"],mean)
 f  m
80 60
>
>
```

3.7 Reading data from external files

File "grades" is shown as follows:

```
      name gender hw1 hw2
1     john      m  60  40
2    peter      m  60  50
3 jennifer      f  80  30
```

Creating a data frame `grades`:

```
> grades <- read.table("grades")
> grades
      name gender hw1 hw2
1     john      m  60  40
2    peter      m  60  50
3 jennifer      f  80  30
```

Other functions that read data from external files of different format: `read.csv,read.delim`. They are specific form of `read.table`. For more details, type `?read.table`.

4 Writing your own functions

In file "demo-fun.R", a function is defined:

```
#looking for the maximum value of a numeric vector x
find.max <- function(x)
{
   n <- length(x)

   x.m <- x[1]
   ix.m <- 1

   if(n > 1)
   {
      for( i in seq(2,n,by=1) )
      {
         if(x[i] > x.m)
         {
            x.m <- x[i]
            ix.m <- i
         }
      }
   }
   #return the maximum value and the index
   list(max=x.m,index.max=ix.m)
}
```

We now can use this function:

```
> #sourcing functions in file "demo-fun.R"
> source("demo-fun.R")
>
> x <- runif(12)
>
> x
 [1] 0.06779191 0.09266746 0.63309784 0.85986312 0.81900862 0.80315468
 [7] 0.71691262 0.35424083 0.59253821 0.23433190 0.60891787 0.35025762
>
> #calling "find.max"
> find.max(x)
$max
[1] 0.8598631

$index.max
[1] 4

>
```

5 Making graphics with R

5.1 Drawing plots on screen

Typing plotting commands, plots will be shown in a seperate window.

5.2 Making graphics in a file

In order to demonstrate how to use R to produce plots and save plots in a file, I use R to draw plots to illustrate the following two functions:

$$f_1(x) = \frac{a_0 + a_1\, x + a_2\, x^2 + \exp(x)}{x^2} \quad (1)$$

$$f_2(x) = \frac{a_0 + a_1\, (10 - x) + a_2\, (10 - x)^2 + \exp(10 - x)}{(10 - x)^2} \quad (2)$$

where $a_0 = 1, a_1 = 2$ and $a_2 = 3$.
The R commands are shown as follows:

```
demofun1 <- function(x)
{
   ( 1 + 2*x^2 + 3*x^3 + exp(x) ) / x^2
}

demofun2 <- function(x)
{
   ( 1 + 2*(10-x)^2 + 3*(10-x)^3 + exp(10-x) ) / (10-x)^2
}

#open a file to draw in it
postscript("fig-latexdemo.eps", paper="special",
          height=4.8, width=10, horizontal=FALSE)

#specify plotting parameters
par(mfrow=c(1,2), mar = c(4,4,3,1))

x <- seq(0,10,by=0.1)

#make "Plot 1"
plot(x, demofun1(x), type="p", pch = 1,
     ylab="y", main="Plot 1")

#add another line to "Plot 1"
points(x, demofun2(x), type="l",  lty = 1)

#make "plot 2"
plot(x, demofun1(x), type="b", pch = 3, lty=1 ,
     ylab="y", main="Plot 2")
```

```
#add another line to "Plot 2"
points(x, demofun2(x), type="b", pch = 4, lty = 2)

dev.off()
```

The file produced contains the resulting plots:

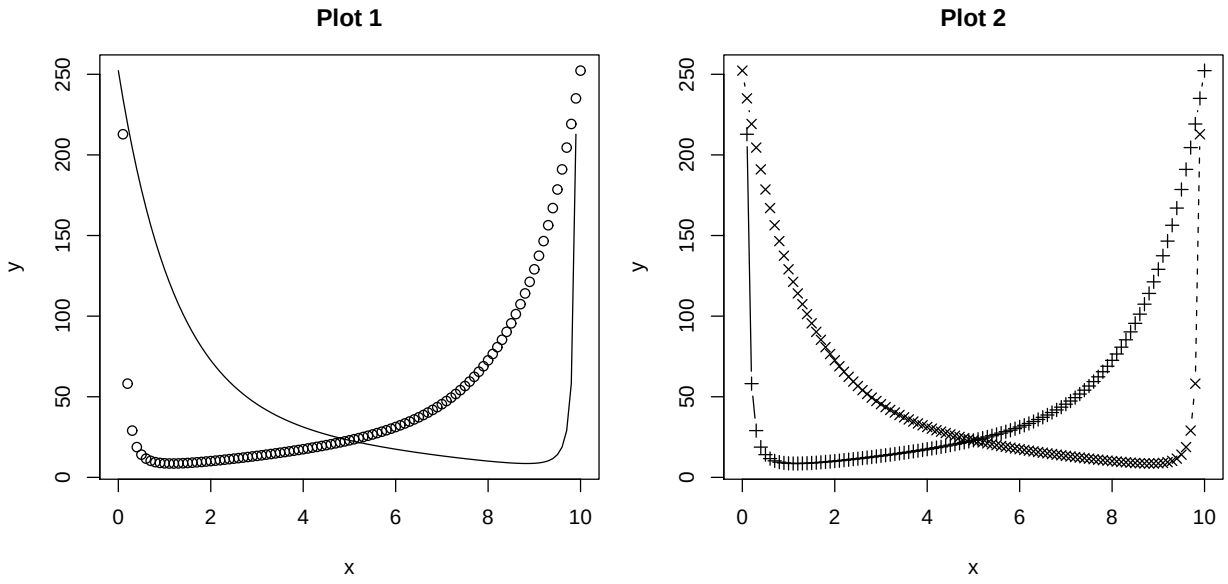

Figure 1: This graph demonstrates two non-linear functions using different lines and points

6 Installing new packages and loading new packages

6.1 Installing a new package from source files

- download a source file, say `apacakge.tar.gz`.
- Run

  `R CMD INSTALL apackage.tar.gz -l /my/rlibrary`

  to compile the source file and install it to directory `/my/rlibrary`.

  You must have GCC compiler installed, which is not available for windows system without doing much effort. The package needs not to be available from CRAN.

6.2 Installing a new package from CRAN

If the package is available from CRAN, you can install a new package without compilation. The precompiled package can be downloaded. To do this, type

`install.packages("apackage",lib="/my/rlibrary")`.

6.3 Loading a package

After installing new package `apackage`, type

`library("apackage",lib.loc="/my/library")`.

Then you can use the functionalities provided by this package.

7 Calling C functions (for unix system only)

R is slow. We better use C codes to do intensive computations. Pointers of R vectors (matrice will be coerced to vectors) are passed to C codes such that C program can use them. This is realized by function ".C".
Below is a simple demonstration.
File "sum.c" is shown as follows:

```
void newsum(int la[1], double a[], double s[1])
{
    int i;
    s[0] = 0;
    for(i=0;i<la[0];i++)
       s[0] += a[i];
}
```

Then compile the C program above:

```
R CMD SHLIB sum.c
```

.
Two files, `sum.o` and `sum.so`, will be produced.
Now we can load "sum.so" into R environment and call the C function with .C:

```
> dyn.load("sum.so")
>
> a <- c(1,2,3,4)
>
> .C("newsum",length(a),a,sum=0)
[[1]]
[1] 4

[[2]]
[1] 1 2 3 4

$sum
[1] 10

>
```

Typically, people write a "wrapper" function to ease calling C functions regularly in R, for example:

```
newsum <- function(a)
{
    .C("newsum",length(a),a,sum=0)$sum
}
```

8 Creating your own package and submitting to CRAN

It is for advanced users. However, it is not difficult. Following this procedure:

- Create a directory "apackage", which has 3 directories: R (containing all R sources), src (containing all C or Fortran files), man (documentation files for each R function), and a file "DESCRIPTION". There are required formats for writing documentation files and "DESCRIPTION".
- (suggested) Run `R CMD check apackage` to check the package and make suggested changes if some come up. A package accepted by CRAN must have no warnings and no errors in this step.
- Run `R CMD build apackage`

To submit to CRAN, just post it (with anonymous ftp) to `ftp://cran.r-project.org/incoming/`. Somebody will run your source file and if it passes `R CMD check` without warnings and errors, it will be posted online as CRAN contributed packages. Mac and Windows binaries will be created.