

Writing an R Package

David Harte

Statistics Research Associates

Wellington

email: david@statsresearch.co.nz

19 November 2007

Outline

- What is an R package?
- Construction of a simple R package
- Including FORTRAN or C code within the package
- Object orientated functions
- Further odds and ends
- Example using hidden Markov models
- Resource documents

The R Project

R is a programming language and environment for statistical computing and graphics (see [1], [2] and [5])



Main Features:

- operators for calculations on arrays, in particular matrices
- integrated collection of tools for data analysis
- graphical facilities for data analysis and display
- an effective programming language (called S)

Home page: <http://www.r-project.org>

Software can be downloaded from CRAN (Comprehensive R Archive Network): <http://cran.r-project.org/mirrors.html>

Running R

When R is started, a command line window appears as:

```
[~/]$ R
```

```
R version 2.6.0 (2007-10-03)
```

```
Copyright (C) 2007 The R Foundation for Statistical Computing
```

```
ISBN 3-900051-07-0
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
```

```
You are welcome to redistribute it under certain conditions.
```

```
Type 'license()' or 'licence()' for distribution details.
```

```
  Natural language support but running in an English locale
```

```
R is a collaborative project with many contributors.
```

```
Type 'contributors()' for more information and
```

```
'citation()' on how to cite R or R packages in publications.
```

```
Type 'demo()' for some demos, 'help()' for on-line help, or
```

```
'help.start()' for an HTML browser interface to help.
```

```
Type 'q()' to quit R.
```

```
>
```

```
>
```

R can be terminated by entering `quit()` or `q()` at the command prompt

The R Help Window

A help window can be opened by entering, at the R command prompt:

```
>  
> help.start()  
>
```

Note that:

- we can search for keywords, phrases or function names
- documentation for each function *conforms to a prescribed layout*
- documentation for many functions contain examples that can be executed

We want to utilise this framework (and more) for our own locally written functions

What is an R package?

- Collection of functions and possibly datasets together with documentation for each function and dataset
- Documentation is in a standardised format
- Easily portable between UNIX, Windows and MAC operating systems
- Easily installed onto a computer that contains the R software

A package can be dependent on other packages - these dependencies are automatically loaded when the given package is loaded

There are many packages freely available from CRAN, see:

<http://www.r-project.org>

Installing a Package onto a Computer

In a UNIX operating system*, one can install or remove a package (with superuser authority) as:

```
R CMD INSTALL packagename_version.tar.gz
```

```
R CMD REMOVE packagename
```

Using a Package in an R Session

To utilise an installed package, start R, then at the R command prompt:

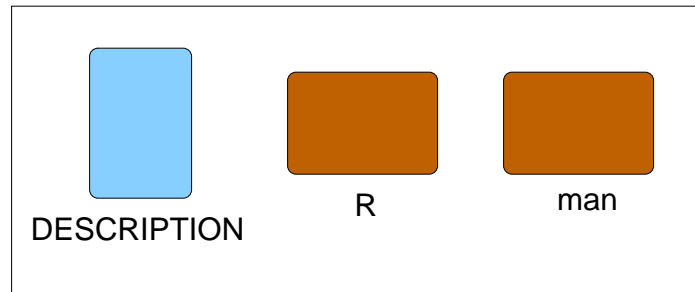
```
>  
> library(packagename)  
>  
>
```

*In Windows, one downloads the compiled binary file (.zip). This is then installed using a pull down menu in an R session. On a MAC OS, download the “.tgz” file.

Construction of a Simple R Package

A package is contained in a folder (directory)

The simplest package will contain two folders: “R” and “man”, and a text file called DESCRIPTION



Files in folder “R” have suffix “.R” and contain the R functions

Files in folder “man” have suffix “.Rd” and contain the manual pages, with a strictly defined layout

The file DESCRIPTION contains the package characteristics (e.g. authors, title, version, license, etc), also with a strictly defined layout

Example

Want our package to contain 2 functions:

```
times2 <- function(x, addnoise=FALSE){  
  x <- 2*x  
  if (addnoise==TRUE) x <- x + rnorm(length(x))  
  return(x)  
}
```

```
times3 <- function(x, addnoise=FALSE){  
  y <- 3*x  
  if (addnoise==TRUE) y <- y + rnorm(length(x))  
  return(list(x=x, y=y))  
}
```

1. Place the above functions into a file called “example.R” say
2. Start R, at command prompt, enter

```
> source("example.R")  
> ls()  
[1] "times2" "times3"
```

The command `ls()` is actually a function, and says to list the current objects

3. Now we tell it to make a package skeleton called “ExamplePackage0” by entering

```
> package.skeleton(name="ExamplePackage0", force=TRUE)  
>
```

Will create a package folder called “ExamplePackage0” containing the R functions `times2` and `times3` in the “R” subfolder, and templates for the manual pages in the “man” subfolder

4. Have copied to “ExamplePackage” to save it from getting overwritten

5. Next we need to add the required information in the files DESCRIPTION and man/*.Rd; the T_EX like markup language is described in [3, §2]
6. Build the portable package tar.gz file:

```
R CMD build ExamplePackage
```

This makes a new “tar.gz” file; is used install on required computers, see section “Installing a Package onto a Computer”

PDF Manual of Package

One can create a pdf file of the package documentation:

```
R CMD Rd2dvi --no-preview --pdf --output="manual.pdf" ExamplePackage
```

Package Checks

It can be checked by running (outside R) at the XTERM prompt:

```
R CMD check ExamplePackage_1.0.tar.gz
```

Will check for various syntax errors, run through all examples, etc

Results placed into folder "ExamplePackage.Rcheck"

Add a Function to an Existing Package

To add a function called `xyz`, add a file called “`xyz.R`” containing the function to the `R` folder and a file called “`xyz.Rd`” to the `man` folder containing the documentation

Example: say we want to add the function:

```
log3 <- function(x)
  log(x, base=3)
```

1. Place into a file called “`log3.R`”, then source into an active R session, then

```
>
> prompt(log3)
>
```

2. This will create a file called “`log3.Rd`”, edit, and place into the “`man`” folder

3. Note that functions already loaded in R can be dumped to a text file by using the `dump` function

Add an Example Dataset to an Existing Package

A package can also contain datasets

Example: say we want to add the following dataset:

```
testdata <- seq(1, 10)
```

1. The data should be loaded into an active R session
2. Then use the `save` function in R to save the data as a binary file:

```
>  
> save(testdata, file="testdata.rda")  
>
```

3. Make a new folder in “ExamplePackage” called “data”, and place the file “testdata.rda” into “data”

4. Make the manual page for `testdata` by using the `promptData` function in an active R session:

```
>  
> promptData(testdata)  
>
```

This will create a file called “`testdata.Rd`”, edit, and place into the “`man`” folder

Note that one may want to embed the test dataset as a text file (space reasons), though this is more tricky; see the package **ssBase** for methodology here

(<http://homepages.paradise.net.nz/david.harte/SSLib>)

Re-Build, Re-Check and Re-Install

When a package is modified, the version number and date should be updated in the files DESCRIPTION and “man/ExamplePackage-package.Rd”

The “package” file (.tar.gz) needs to be rebuilt and re-checked; can be achieved at an XTERM or DOS prompt:

```
R CMD build ExamplePackage
```

```
R CMD check ExamplePackage_???.???.tar.gz
```

As *superuser* in UNIX, remove the old version, and install the new one; can be achieved at an XTERM prompt:

```
R CMD REMOVE ExamplePackage
```

```
R CMD INSTALL ExamplePackage_???.???.tar.gz
```

Including FORTRAN or C Code

Tasks that are numerically intensive are often achieved faster using FORTRAN or C

Such code can be interfaced to an R function

All input and output should be passed through the R function

Embedding it correctly into an R package will ensure that it is correctly compiled and loaded

See [[3](#), §5] for full details

Example Using FORTRAN

Consider a FORTRAN subroutine as follows; there are 3 required components to embed within the package:

1. The FORTRAN subroutine multiplies a real vector of length n by an integer scalar m

```
subroutine scale(m, n, c)
integer n, m
double precision c(n)
i=0
do while(i .LE. n)
    c(i) = c(i)*m
    i = i + 1
enddo
end
```

Make a folder “ExamplePackage/src” with a file called “scale.f” containing the above code

2. The FORTRAN subroutine is called from within the R function `timesm`, which in turn calls the R function `.Fortran` (see R help page for more details)

```
timesm <- function(x, m){  
  n <- length(x)  
  y <- .Fortran("scale", as.integer(m), as.integer(n),  
                as.double(x), PACKAGE="ExamplePackage")  
  names(y) <- c("m", "n", "x")  
  return(y)  
}
```

In the folder “ExamplePackage/R”, make a file called “timesm.R” containing the above code

One should also add the manual page
“ExamplePackage/man/timesm.Rd” for the new R function

3. The R function `.First.lib` is run when a user runs `library(ExamplePackage)`; it tells R that the package uses compiled[†] FORTRAN or C code

```
.First.lib <- function(lib, pkg) {  
    library.dynam("ExamplePackage", pkg, lib)  
}
```

In the folder “ExamplePackage/R”, make a file called “zzz.R”[‡] (conventionally used name) containing the above code

After modifying the package, one needs to again re-build, re-check and re-install

[†]In UNIX, the FORTRAN code will be automatically compiled when the package is loaded onto a computer; in Windows it is compiled when the binary file (.zip) is built.

[‡]One can also put other commands here that must be run when the package is loaded, see [3, §1.1.3].

Object Orientated Functions

There are many *generic* functions in R: `logLik`, `plot`, `print`, `residuals`, `simulate`, `summary`, etc

The *method* used by a generic function depends on the *class* of that object

For example, if `x` has class `"xyz"`, when `print(x)` is executed, the generic function `print` will look for a function called `print.xyz`, if found `print.xyz(x)` is executed, else `print.default(x)` is executed

This allows one to build neater and more powerful code

See [\[1, §10.9\]](#) for further introductory information, [\[3, §7\]](#) for writing packages that include or provide methods to generic functions, and [\[2, §5\]](#) for more advanced aspects

Example - Object Orientated Functions

Consider the following object containing daily rainfall totals:

```
x <- list(station="Kelburn", units="mm",  
          y=c(10, 5, 0, 0, 4, 18, 0, 0, 2, 0, 0, 0,  
              1, 0, 0, 22, 4, 0))  
print(x)
```

It is printed as a "list":

```
> print(x)  
$station  
[1] "Kelburn"  
  
$units  
[1] "mm"  
  
$y  
[1] 10  5  0  0  4 18  0  0  2  0  0  0  1  0  0 22  4  0
```

Now we give it a class, and define a print *method* for that class:

```
class(x) <- "rainfall"
```

```
print.rainfall <- function(x, ...){  
  n <- length(x$y)  
  cat(paste("Daily Rainfall at ", x$station,  
    " (", x$units, ") \n", sep=""))  
  tmp <- cbind(c(1:n), x$y)  
  colnames(tmp) <- c("Day", "Amount")  
  rownames(tmp) <- rep("", n)  
  print(tmp)  
}
```

```
print(x)
```


Similarly, a method for the `plot` function could be:

```
plot.rainfall <- function(x, y, ...){  
  plot.ts(x$y, xlab="Day", type="b",  
          ylab=paste("Daily Rainfall at ", x$station,  
                     " (", x$units, ")", sep=""))  
}
```

```
plot(x)
```

Note that `plot.ts` is a plot method for a time series object

Note that objects within objects can also have their own class

This suggests we should have:

```
x <- list(station="Kelburn", units="mm",
          y=ts(c(10, 5, 0, 0, 4, 18, 0, 0, 2, 0, 0,
                 0, 1, 0, 0, 22, 4, 0)))
class(x) <- "rainfall"
```

```
print.rainfall <- function(x, ...){
  n <- length(x$y)
  cat(paste("Daily Rainfall at ", x$station,
            " (", x$units, ") \n", sep=""))
  print(x$y)
}
```

```
plot.rainfall <- function(x, y, ...){
  plot(x$y, xlab="Day", type="b",
        ylab=paste("Daily Rainfall at ", x$station,
                    " (", x$units, ")", sep=""))
}
```

Note that `class(x$y)` is "ts", hence the above use `print.ts` and `plot.ts`

Further Odds and Ends

It is not a requirement that each function must have its own manual page; similar functions or those with similar argument lists can be placed on the same page

In fact, each manual page in the pdf document is simply a particular “.Rd” file in the “man” folder

Demonstration examples, package vignettes, and package tests can also be added (see [[3](#), §1.1])

One can also define generic functions

Front ends to R (GUIs) can also be constructed, see [[3](#), §8]

Package HiddenMarkov

A package for the fitting of discrete time HMMs (class `dthmm`), Markov modulated Poisson process (class `mmpp`), and a Markov modulated generalised linear model (class `mmglm`)

For each, an object is defined with one of the above classes, which contains the necessary model parameters, data, etc

Generic functions that have methods for each of the above model classes are: `BaumWelch`, `summary`, `logLik`, `residuals`, `Viterbi`

In an object orientated environment, one would tend to group functions according to the generic activity; and in a non-object orientated environment according the model type

Resource Documents

Resource documents are available from <http://www.r-project.org>, then click “Manuals”

Some useful ones are:

- [1] *[An Introduction to R](#)* gives an introduction to the language and how to use R for doing statistical analysis and graphics.
- [2] *[The R Language Definition](#)* documents the language per se. That is, the objects that it works on, and the details of the expression evaluation process, which are useful to know when programming R functions.
- [3] *[Writing R Extensions](#)* covers how to create your own packages, write R help files, and the foreign language (C, C++, Fortran, ...) interfaces.
- [4] *[R Data Import/Export](#)* describes the import and export facilities available either in R itself or via packages which are available from CRAN.
- [5] *[R Installation and Administration](#)*.