Creating and building R packages: course notes

Matt Nunes

19th April 2010

Why build R packages?

- Convenient code storage and version control
- "Open source" ideology: allows others to reproduce your work
- Facilitates easier code development on collaborative projects
- Con: Sometimes the building process can be tedious and frustrating!

1 Package structure

Package structure

An R package consists of the following components:

DESCRIPTION file with package information (e.g. name, version, dependencies etc)

R directory containing the R code for functions in the package

man directory containing help files for objects in the package

data (optional) directory containing any datasets/dataframes

src (optional) directory containing any external compiled code (e.g. C or Fortran code)

1.1 *DESCRIPTION* file

DESCRIPTION **file fields**

Package the name of the package

Title A single line package title description

Version Version of the package (formats are e.g. 0.1 or 1.1-2)

Description A more detailed single paragraph description of what the package does

Author As many names as required, plus email addresses

Maintainer a name and an email address

License what licence you want to distribute the package under, e.g. "GPL-2" or "Unlimited" (see share/licenses in R home directory)

Optional fields

Date the current build date, e.g. 2010-03-03 or 19/04/10

Depends list of dependent packages required to run your package.

Suggests Similar to Depends, e.g. those packages used for some function examples

LazyLoad If yes, builds package so that functions are loaded when required (more efficient).

- There are other optional fields. See *R* documentation for more information.

Example *DESCRIPTION* file

```
Package: mypackage

Version: 0.1-1

Title: A package for doing something

Date: 19/04/10

Depends: R (<=2.9), anotherpackage (>=0.1.2)

Author: Fred Bloggs <f.bloggs@lancaster.ac.uk>

Maintainer: Fred Bloggs <f.bloggs@lancaster.ac.uk>

Description: Does some really cool stuff

License: GPL
```

1.2 Code and datasets

Code and datasets

- R code can be put into `mypackage/R` as one file containing all code, or individual files. Format is the same as output from `dump()`, must have the extension `.R` or `.r`
- There is usually a "processing function" (traditionally called *zzz.R*) with tasks to be performed when the package is loaded, such as loading libraries and compiled code (using `library.dynam`) .
- Datasets usually have the extension `.rda`, and can be output from *R* using `save()`.

Example R function:zzz.R

```
.First.lib <-function(lib,pkg){

ver <- read.dcf(file.path(lib, pkg, "DESCRIPTION"),

    "Version")

ver <- as.character(ver)

library.dynam("mypackage",pkg,lib)

cat("mypackage", ver, "loaded\n")

}
```

1.3 Help files

Help files

- *.Rd* files are used to generate HTML, pdf and LateX help files for the package itself and any *R* objects in the package
- Use LaTex-like entries to describe functions and datasets
- `prompt` and `promptPackage` will create empty help files to be filled in (similar to `package.skeleton` later)
- There are many different optional fields that can be used – see *R extensions* documentation for details

.Rd file fields

name the name of the function

alias a "topic" with which multiple functions can be grouped together if necessary

Title a one line description of the function. Must start with a capital letter, no full stop at end

description a description of the function.

usage the call syntax to the *R* function

arguments A list environment describing each argument to the function

author similar to the `author` field in the *DESCRIPTION* file

value a list environment describing what the function returns

examples R code giving an example of use. Must be able to be run without errors in *R*

keywords one or two entries from an allowed list in the *R* home `/doc` directory. The list can also be seen using `file.show(file.path(R.home("doc"),"KEYWORDS"))` from *R*

details (optional) More details on how the function works, e.g. the algorithm/technique used.

seealso (optional) links to other *R* objects related to the function. Format: `\code{\link{...}}`.

note (optional) Any warnings or comments for the user

references (optional) Any references to articles in the literature. Format: `\url{...}`.

docType (for datasets, package) either `data` or `package`

format (for datasets only) a description of the format of the data

source (for datasets only) where the data came from (e.g. url).

Example *.Rd* file

```
\name{fifthroot}
\alias{fifthroot}
\title{Compute the Fifth Root}
\description{
   Compute the Fifth Root of a Real Number
}
\usage{
fifthroot(x)
}
\arguments{
   \item{x}{a real number.}
}
\seealso{
   \code{\link{sqrt},\link{cuberoot},\link{fourthroot}}
}
\examples{
# fifth root of 32
fifthroot(32)
}
\keyword{math}
```

1.4 Making life a bit easier...

Shortcut: `package.skeleton`

Some of the work of creating a package (especially for help files) can be done from within R using `package.skeleton`. This function creates

- package structure
- DESCRIPTION file
- skeleton help files
- a few README files to give hints

The files can then be edited using for the specific package using any text editor.

Example `package.skeleton` call:

```
package.skeleton(name = "myfirstpackage", list=ls(),path = ".")
```

Some `package.skeleton` arguments:

name the name of the package

list a character vector of *R* objects to include in the package

path where to create the package

code_files a character vector of names of any R code files to base the package around

2 An example R session: building your first package

In this section we will build a small example R package from scratch. The subsections act as a step-by-step guide to building packages in general.

Note: in this section,

matt:/home$ represents the Linux command prompt

> represents the *R* prompt

⊙ represents instructions/commands to do in *R* or Linux.

Getting the course files

⊙ You can `cp` the course files from my home directory, and then uncompress them using:

```
tar -xvf coursefiles.tar.gz
```

2.1 Environment setup

It is useful to create a directory in your workspace to contain built packages, and also a directory where packages are installed (locally). This can be done using the command `mkdir` e.g.

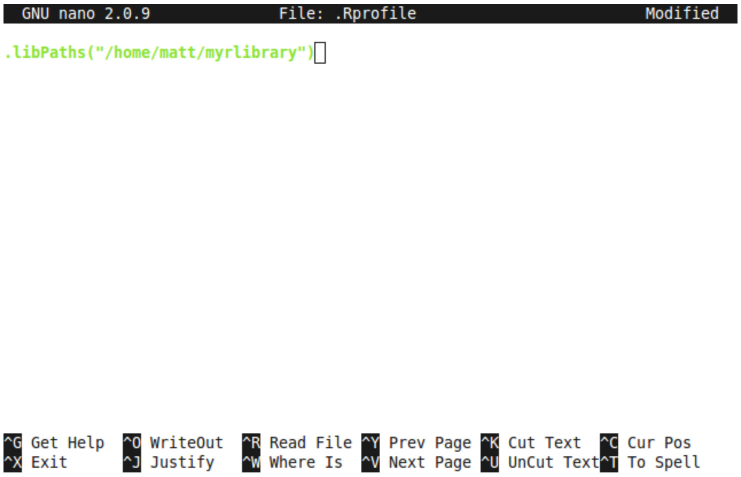
```
matt:/home$ mkdir myrpackages
matt:/home$ mkdir myrlibrary
matt:/home$ ls

myrlibrary
myrpackages
```

2.1.1 .Rprofile

To let R know where your locally installed packages are, you need to put the file .Rprofile in your home directory, containing the path to your chosen directory, using the line: `.libPaths("/home/matt/myrlibrary")` . This file can be done using a text editor (e.g. *pico* or *vi*). For example, using *pico*:

```
matt:/home$ pico .Rprofile
```



```
.libPaths("/home/matt/myrlibrary")
```

The file is saved by `cntrl-x` followed by `y` (yes).

If your package has dependencies, *R* will need to know where locally installed dependent packages are *during the package build*. This is because during the build (and checking process), *R* is run as `--vanilla`. In this case, the environment variable `R_LIBS` might need to be set. One way to do this is to set the `R_LIBS` environment variable within *~/.bashrc*, adding the line : `export R_LIBS=$R_LIBS:~/myrlibrary`. If you do use edit *~/.bashrc*, you need to load the changes by `source ~/.bashrc`.

2.2 creating a package using `package.skeleton`

⊙ Start *R*, and then source the functions for the *roots* package.

```
> source(‘‘Rcode-roots.R’’)
```

If you do a `ls()` command, there should be a number of functions to compute numerical roots of functions. In particular, the functions `fifthroot` and `fourthroot` use exernal compiled code.

⊙ Use `package.skeleton` to create a package named *roots* using the functions in the workspace.

· · · ⋆⋆⋆ · · ·

Your R console should say spurt out something like this:

```
Creating directories ...
Creating DESCRIPTION ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in ’./roots/Read-and-delete-me’.
```

⊙ Quit *R* and have a look at some of the skeleton package files that have been created. For example:

```
matt:/home$ cd roots
matt:/home$ cat DESCRIPTION
```

should show

```
Package: roots
```

```
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2010-04-19
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?
LazyLoad: yes
```

⋆ It is important to have a look at the layout of the .Rd files in `roots/man` since they are a potential source of build problems later.

2.3 File preparation

⊙ The next step in building the *roots* package is to edit the DESCRIPTION, *.Rd* files (in `roots/man`) with a text editor, e.g. *pico*. To save files, use `cntrl-x` followed by `y` (yes).

⋆ This is the tedious bit. To help you out, I have done a couple for you. You can copy (`cp`) or move (`mv`) them across from the `coursefiles` directory.

```
matt:/home$ cp coursefiles/man/*  /roots/man
```

· · · ⋆⋆⋆ · · ·

⊙ If necessary, you need to create the file *zzz.R* and put it in the `roots/R` directory.

2.4 Compiled code

⊙ Next we put any compiled code in the package, by creating the `roots/src` directory and moving code files to the directory. Again, these are contained in the `coursefiles` directory.

```
matt:/home$ cp coursefiles/src/*  /roots/src
```

If you want to, you can have a look at the source code using e.g. `pico, cat` or `more`.

· · · ⋆ ⋆ ⋆ · · ·

2.5 After all the hard work...

Building and checking your package

- To build your package use `R CMD build path_to_packdir`, e.g. `R CMD build roots`
- You now have a useable package!!
- The built package can be checked for errors using `R CMD check [-l path_to_local_libdir] mypkg.tar.gz`. The `-l path_to_local_libdir` flag is optional, but is necessary on some systems when `$R_LIBS` is not set in *∼/.bashrc*.
- This should be done (especially before submitting to CRAN). A proper package should pass **all** checks!

Installing and using your package

- To install your package, use the command:

  `R CMD INSTALL -l path_to_local_libdir mypkg.tar.gz` For example:

  `R CMD INSTALL -l /home/matt/myrlibrary roots_0.1-1.tar.gz`

- After installation, your package can be used by calling it as you would any other package: `library(roots)`

3 Further examples and debugging

Possible `R CMD check errors and warnings`

Below are a few common problems with passing the `R CMD check` and possible solutions (useful for debugging more complicated packages).

`T/F` warnings. `R CMD check` does not like shorthand for boolean values. Use explicit `TRUE/FALSE` in R code instead.

S3 method inconsistency. There are probably two S3 methods which have different arguments in their `usage` calls/arguments lists.

undefined global variables. This is often a result of package dependence not working properly, or using a variable within R code that doesn't exist.

compiler warnings. This is obviously to do with your compiled code (independent) from *R*. Any errors should be fixed (or ignored if you know what you are doing).

examples code not executable. The example given in a help file does not run independently. Try copy and pasting the offending example code from the *.Rd* file to see where it fails and correct it.

code/documentation mismatches. The `usage` section of an *.Rd* file does not match the argument list of the corresponding *.R* file.

undocumented code. Is there a *.Rd* file missing? Check the *R* functions list against the `man` directory.

package dependency error. Has the required dependent package been `INSTALL`ed and does R know where to find it? Check *.Renviron/.bashrc* for correct path to local/global library.

4 R sesson II: building a more complex package

In this section, the task is to successfully build and debug a more complicated package, resembling a more realistic situation. The package is named *mattpkg* and is in the coursefiles directory.

Task description

The package mattpkg uses a C routine to perform variable manipulation for a certain technique, and depends on the packages *roots* and *secondpkg* (found here: `coursefiles/secondpkg_1.2.tar.gz`). The task will be to create and install a fully functional package *mattpkg* (passing all `R CMD check` tests). There are intentional errors in some package files which need to be debugged. In other words, you will need to:

- install any necessary packages (locally)
- make sure all files in the *mattpkg* directory are ok
- build the package
- check the package for errors
- install the clean package

⋆ You can test the package within *R* by `source`ing the file `coursefiles/mattpkg-script.R`.

· · · ⋆⋆⋆ · · ·

5 And lastly...

Credits

- Notes and files from this course will be available from:

  `http://nunes.homelinux.com/~matt/mathstuff/Rhelp.html`

- For more information about building *R* packages, see the *R extensions manual*: `cran.r-project.org/doc/manuals/R-exts.pdf`

  It has lots of information in it, especially about including compiled code in packages, syntax etc (though it's not always the clearest document for beginners).

- Alternatively, you can always email me...

  `anotheremail@inbox.com`