Making R Packages Under Windows:
A Tutorial

P. Rossi 4/05

I don't claim this tutorial to be comprehensive. It is designed for the developer who has only a limited knowledge of R and the R environment.

Why create an R package?

There are three good reasons:

1. Creating an R package forces you to document your code and provide test examples to insure that it actually works. It will also be much easier to use your code as documentation will only be a ? command away and all of your functions and shared libraries will be available for use.
2. If your goal is disseminate your research, this is an ideal way of making sure others have access to your work. It will also increase the probability that eventually your work will be correct. You will also learn more about the properties of your research ideas through the experience of others.
3. Ease your sense of guilt by giving back something to this amazing community of volunteers!

Coding Philosophy

The best way to develop an R package is to start your development work in R, only later adding low-level language (e.g. C/C++/Fortran) *if* necessary. It is harder to start with some massive C routine and write the R wrapper. R is much better for error checking and processing of arguments than low-level languages. R is also easier to follow so others can adapt your code. R is surprisingly fast. In my own package development, I maintained a 10:1 ratio of lines of R code to C/C++.

Reducing Frustration

Some have found the creation of a package under Windows to be frustrating. There is a sense in which the Windows R environment is a house of cards that must be carefully assembled or it won't work!

To help, I have created a very small package consisting of one function and one C routine. You can test out your Windows environment on this simple package first before you attempt to create a package from your own code. After you successfully create this test package, try one of your own by using only a tiny subset of the functions you have written. Then build up your package by adding more functions, one by one.

Don't try to make a package with an older version of the Windows operating system. XP works fine (some say 2000 as well) but Windows 95/98 is reputed to be problematic. If you are running an older version of Windows, you probably want to upgrade anyway.

Getting Started: Making Up for What Windows Lacks

R was designed in a Unix environment which includes a rather complete set of compilers, programming utilities, and text-formating routines. Windows lacks these components. This means that we have to download the free-ware equivalents of the subset of Unix needed to build your package.

Fortunately, these components are available and easy to obtain thanks to the work of Brian Ripley and Duncan Murdoch. See http://www.murdoch-sutherland.com/Rtools/.

You need the following components:

1. A minimal set of Unix utilities (what Ripley/Murdoch call "Rtools").
2. The GNU compiler set (only if your package contains C/C++ /Fortran code)
3. perl (a scripting language used by the R installer and checker)
4. The Microsoft html help compiler
5. a version of TeX (fpTeX)

You only need TeX if you want to submit your package to the Comprehensive R Archive Network (CRAN). TeX is used to format the LaTeX version of your documentation pages. If you don't' have TeX, you will not be able to check for TeX errors generated in the creation of your documentation. I recommend *not* using MikTeX. It will not work with the R package installer/builder without some tinkering (not for the un-initiated!). Below we describe how to obtain fpTeX, a freeware version of TeX which works well with R. If you want to use MikTeX, follow the instructions on the Murdoch web site (see the link above).

I am assuming that you have a decent text editor (MS Word is not designed to be a text editor – avoid it. notepad will save files with the .txt extension by default which can cause confusion). I recommend VIM. There is a Windows version of VIM, which is an "improved" version of the ubiquitous Unix editor, vi. http://www.vim.org/.

Steps:
   Note: these steps to acquire the necessary freeware are also summarized at the Murdoch web site. We include them here as a convenience.

1. Download the set of "Rtools" or Unix utilities.

   In your browser, right click on:
   http://www.murdoch-sutherland.com/Rtools/tools.zip   Save the zip file into a directory. I suggest C:\Rtools (assuming C is your primary disk drive). Unzip the files. I use Winzip. If you have Winzip on your machine, you can simply right click on the tools.zip file in windows explorer and specify "extract here."

2. Download and install the minGW.

"minGW" stands for minimal GNU for Windows. This is the gcc, g++, and g77 set of compilers and other associated utilities. This will have to be done by first downloading the packaged version and then double clicking this file (it is a self-extracting .exe file). Follow the link: http://prdownloads.sf.net/mingw/MinGW-3.2.0-rc-3.exe?download or click on `MinGW-3.2.0-rc-3.exe`.

3. Download and install perl.

http://www.activestate.com/Products/ActivePerl/Download.html.
Choose the version for XP (don't use these instructions for anything else). Use the default installation settings.

4. Download and install the Microsoft html compiler.

This will be used by the R package builder to create compiled html help files. This program bundle will install itself in the usual Windows fashion. After downloading the install .exe file, simply double click on it and follow all defaults.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/hwmicrosofthtmlhelpdownloads.asp

5. Download and install fpTeX.

This is done in a somewhat cumbersome way. You first must visit the fpTeX website and then download two small files (a batch program and a utility). http://www.fptex.org/fptexli2.html

Create a directory such as c:\tempfpTeX, download the two files by the usual right-click, save target as method. Open a command prompt or "console" window (start | run |cmd). `CD` to the directory you have created and then invoke the batch file by typing the name of the batch file – "getfptex.bat."

The batch file will start a long process of ftping each of the files of the fpTeX package. This will take a very long time – at least one hour and perhaps more! Be patient. After downloading the files, you can double click on the installer, TeXSetup.exe. Choose the default options.

6. Change your PATH "environmental variable"

Unix uses a search path (as does R) to locate various files to execute at the command prompt. Virtually no Windows programs use this mechanism. For this reason, most Windows users have never had to concern themselves with a path. However, in

order for these tools (and the R install/check/build scripts) to work, you MUST set the correct path.

To set the path, right click on the "My Computer" icon on your desktop. Choose properties and click on the "advanced" tab. Click the environmental variables button and you will see a display like the following:
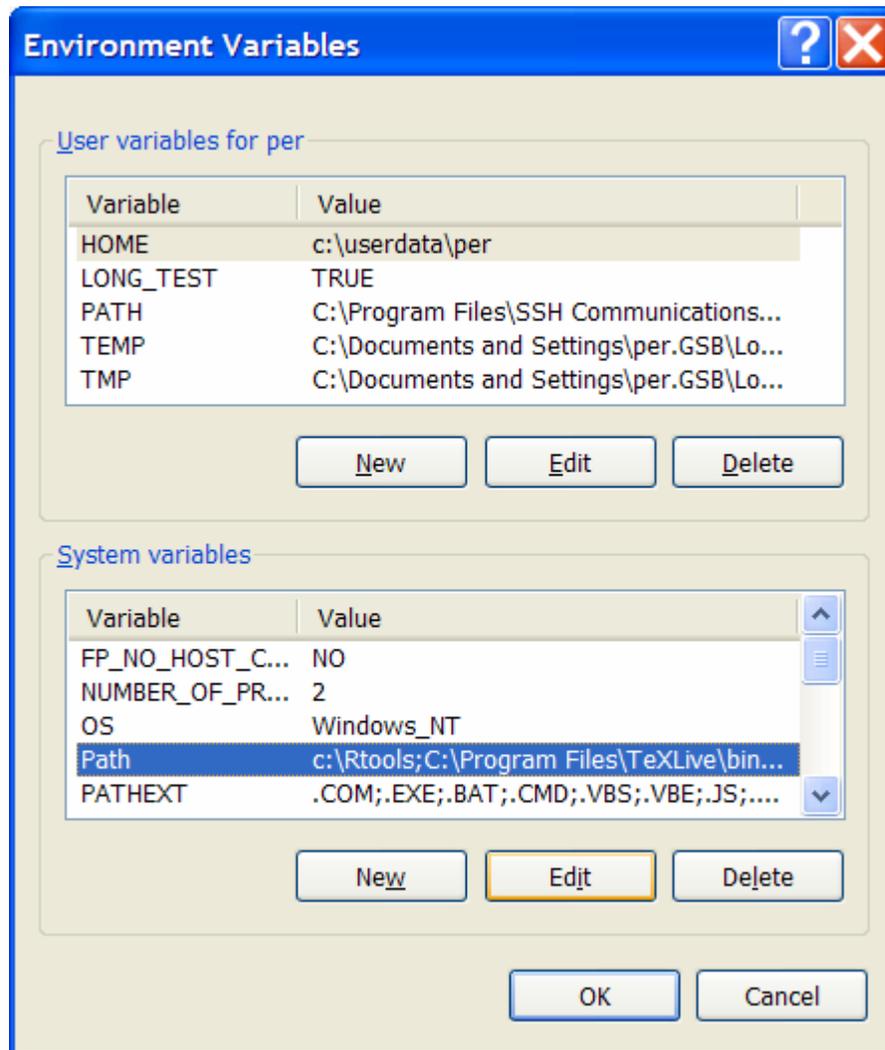
We need to change the "system variable" path. To do so, click on the variable and select the edit button. Change the path so that the directories containing fpTeX, Rtools, minGW, perl, R and the html help compiler are at the *beginning* of the path. This will be somewhat tedious. Once you are "editing" the path variable, you will need to use the left arrow key to get to the beginning. You will need to be very

careful to type the path names correctly. See below for an example of the beginning of the path as set on my machine:

```
c:\Rtools;C:\Program Files\TeXLive\bin\win32; c:\mingw\bin;
C:\Perl\bin\;c:\PROGRA~1\R\rw2001\bin\;c:\PROGRA~1\HTMLHE~1\
```

Several points are in order about the path variable:

1. path designations under Windows are NOT case-sensitive.
   `c:\program files\` and `c:\Program Files\` are equivalent!
2. The second item, `c:\program files\TeXLive\bin` was added automatically by the fpTeX install.
3. These "unix-like" utilities use the "bin" directory to store the binary, executable or ".exe" files. DON'T forget the "bin" subdirectory in the path! The Microsoft html help compiler is the exception (see path above).
4. Under XP, you can use either the full path name (including spaces) or the "DOS" short names. That is, `c:\program files\` and `c:\progra~1\` are equivalent.

Now you are ready to "test" your Windows environment to see if you have everything installed correctly.

Open a "Command Prompt" window AFTER you have changed the path environmental variable (the environment variables are initialized and fixed for each invocation of a Command prompt window).

Type "path" at the prompt.

You will see the "path" variable echoed to the window. Check it carefully! If you can, have a colleague check it again!
If there is an error, go back and edit path. Close the command prompt window and start again.
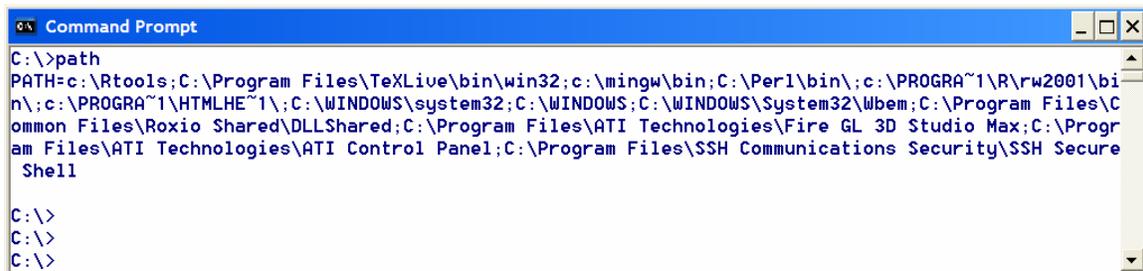
```
Command Prompt                                                    _ □ ✕

C:\>path
PATH=c:\Rtools;C:\Program Files\TeXLive\bin\win32;c:\mingw\bin;C:\Perl\bin\;c:\PROGRA~1\R\rw2001\bi
n\;c:\PROGRA~1\HTMLHE~1\;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\C
ommon Files\Roxio Shared\DLLShared;C:\Program Files\ATI Technologies\Fire GL 3D Studio Max;C:\Progr
am Files\ATI Technologies\ATI Control Panel;C:\Program Files\SSH Communications Security\SSH Secure
 Shell

C:\>
C:\>
C:\>
```

Now we will check to see if the path is set correctly by invoking Rterm (the command prompt version of R).

Type R. If you are told "R is not a recognized as an internal or external command …," your path is not set to the correct directory which contains "R.exe". You should see the usual welcome message to R and get the familiar ">" prompt.

Now quit out of R in the usual way and check to see if minGW, perl, TeX, and Rtools are installed. To do so, type (on separate command lines)

```
gcc --help              (the GNU C compiler)
perl --help
TeX --help
dos2unix --help         (one of the Rtools)
```

For each, you should see a list (sometimes quite long) of options.

If any of these tests fail, check the path carefully. Look into the directory specified in the path for the `gcc.exe`, `perl.exe`, `TeX.exe`, or `dos2unix.exe` files.

If successful, you are ready to learn about package testing and building.

Package Construction

The process of package creation is documented well in Chapters 1 and 2 of "Writing R Extensions" (hereafter referred to as WRE). However, these chapters are designed as a reference and include many features which are not needed for a simple package. While this material is written with remarkable precision, it can be dense for the neophyte.

Most of what is presented below applies to all platforms on which R runs.

Step 1: Create the directory tree

Assume that we wish to create a package entitled "test." We must first create a directory tree upon which the R installer/builder will operate.

The directory tree should contain a top level directory with two files:

    DESCRIPTION
    NAMESPACE

And subdirectories:

    man   <<< for documentation
    R   <<< for R function definitions
    src   <<< for low-level source code (this is optional)
   data   <<< for package datasets (this is optional)

On this website, there is a zip file (`test.zip`) which has this directory tree and files for a very simple test package consisting of one R function and one c file (no datasets).

To create this directory tree for "test," simply unzip the `test.zip` file.

To create this directory tree for your own package, simply use Windows Explorer to create the main and subdirectories. Copy your R function code into the R directory and your source code (if any) into the `src` subdirectory. Copy the `test\DESCRIPTION` and `test\NAMESPACE` files into your package main directory to serve as templates.

Another way to create the basic directory structure is to use the R command, `package.skeleton( )`. See the help file on this function for further information.

Below is the Windows Explorer display of the tree structure for the test package.
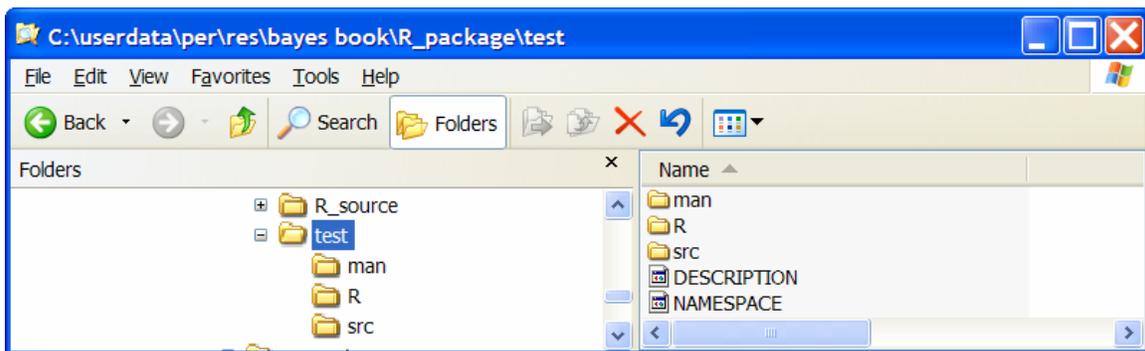
Step 2: Modify/Create the DESCRIPTION FILE

The DESCRIPTION file tells R the basic characteristics of your package as well as providing a short summary. Section 1.1.1 of WRE describes this file. For a simple package, all that is necessary is to provide the following fields. Below the < > notation means textt supplied by you.

```
Package:        < name of the package>
Version:        < number of the form n.n  -- start with 0.0>
Date:           <  date in yyyy-mm-dd format >
Title:          < short title – try to be a descriptive as possible
                – no more than one line >
Author:         <your name (s – use comma) followed by your email address
                enclosed in < brackets >>.
Maintainer:     <  one name – don't end in comma,period >
Depends:        < version of  R – use a recent version – it's free, why not upgrade?
                e.g. R (> = 2.0.1)
Description:     < 5-10 line description. Highlight major features and
                Intended audience >
License: GPL   (version 2 or later)
```

The file for "test" package is:

```
Package: test
Version: 0.0
Date: 2005-04-01
Title: test package
Author: Peter Rossi <peter.rossi@ChicagoGsb.edu>.
Maintainer: Peter Rossi <peter.rossi@ChicagoGsb.edu>
Depends: R (>= 2.0.1)
Description: test is a simple package that implements a
  naive method for drawing inverted Chi-squared random variables.
  A simple C function is called.  The C function is written to
  illustrate many of the C statements needed for use with R.
License: GPL (version 2 or later)
```

Note: there are other possibilities for the "license" field, see WRE for details.

Step 3: Create/Modify NAMESPACE file

The NAMESPACE file explains what libraries should be loaded with the package as well as
what functions should be "exported" or made publicly available, e. g.

   UseDynLib(test)   <<< load the test shared object (your compiled code, if any)

   export (function1, function2, …) <<< list of functions in your package.

The file for "test" is:

```
useDynLib(test)

export(riChisq)
```

Step 4: Check your R functions

For each function you wish to allow users to use, construct an example file which runs the function and provides test output which you can check to see that the function is working properly.

Tips:
1. use the `return` function to return the last expression in the function definition. The R installer may complain if you simply end the function definition with an expression, e.g.
```
myfun=
function(args)
{
….
return(expression)
}
```

   not

```
myfun=
function(args)
{
….
expression
}
```

2. If your function requires data, simulate data using R's extensive random number generation methods. Use `set.seed(666)` to set the seed to 666 so that you can exactly reproduce the results for the purpose of testing.
3. Comment your function definitions. This will allow others to adapt your function code for other related uses.
4. Adopt a reasonable and descriptive naming convention for your functions. Don't use short names than can be confused with local variables (e.g X1). Don't define function with the same names as in the R "base" package. I like the JAVA–like naming convention of lowercase with capitals for each new concept, e.g. myFunPrint.

Source your functions and then source your examples to make sure they run correctly. You can re-use your example files later when you create documentation.

Step 5: Check Your Source Code

If you have source code, it is imperative that you check the code thoroughly. Use the GNU compilers to avoid frustration. If you are using gcc, use the flags `-Wall -pedantic` and eliminate all warnings. Test the code by creating a DLL using `R CMD SHLIB` and `dyn.loading` it into R. This tutorial doesn't pretend to cover interfaces with low-level languages and coding.

Most text files created under Windows end in the CR/LF (carriage return-line feed) combination. Under Unix, lines end in CR only. Some compilers will complain when they encounter source code files ending in CR/LF. To remove the CR/LF combinations from your source code files, use the `dos2unix` command, as in `dos2unix test.c`. `dos2unix` is in the Rtools already installed.

Step 6: Create Documentation Files

This is one of the most difficult, but most important, steps. If your documentation is poor, no one will use your package, including yourself!

For each function in the `R` directory, we must create an associated R documentation file (with the extension Rd) in the `man` directory. That is, if `myfun.R` is a file in the `R` directory, we must create the file `myfun.Rd` in the `man` directory.

R uses a "text mark-up" language to create these "Rd" or documentation files. The R installation scripts translate the generic Rd language into html, plain text and LaTeX files. The R documentation markup language bears a superficial resemblance to TeX and LaTeX. Does this mean that you have to know LaTeX to write R documentation? NO!

The best way to start is with a template. Create a template for each of your functions and then edit the template to add the information required.

Creating the Template

The R function `prompt` will create a template file for a function that is loaded into the current R workspace. To use prompt, source the function and use the command, `prompt(myfun,file="myfun.Rd")` to create a template file. This file will have all of the required sections. Move this file into the `man` directory and then edit it with your favorite text editor (VIM).

For example, consider the function, `riChisq` defined in our "test" package. If we run `prompt` on this function, we will obtain:

```
\name{riChisq}
\alias{riChisq}
%- Also NEED an '\alias' for EACH other topic documented here.
\title{ ~~function to do ... ~~ }
\description{
  ~~ A concise (1-5 lines) description of what the function does. ~~
}
\usage{
riChisq(nu)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{nu}{ ~~Describe \code{nu} here~~ }
}
\details{
```

```
  ~~ If necessary, more details than the __description__  above ~~
}
\value{
  ~Describe the value returned
  If it is a LIST, use
  \item{comp1 }{Description of 'comp1'}
  \item{comp2 }{Description of 'comp2'}
  ...
}
\references{ ~put references to the literature/web site here ~ }
\author{ ~~who you are~~ }
\note{ ~~further notes~~ }

 ~Make other sections like Warning with \section{Warning }{....} ~

\seealso{ ~~objects to See Also as \code{\link{~~fun~~}}, ~~~ }
\examples{
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--  or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function(nu)
{
# purpose: draw from inverted Chi-squared distribution using a
#          "brute" force approach.  for illustration purposes only.
# arguments:
#           nu -- degrees of freedom
# output:
#           list of (nu,iChisq)
#
# create a vector to hand-in to C routine
#
vector=double(nu)

# call interface to c

Chisq=.C("mychisq",as.double(vector),res=double(1),as.integer(nu))$res

return(list(nu=nu,iChisq=1./Chisq))
  }
}
\keyword{ ~kwd1 }% at least one, from doc/KEYWORDS
\keyword{ ~kwd2 }% __ONLY ONE__ keyword per line
```

Format of Rd Files

Rd files are in a "text-markup" language, consisting of a series of commands followed by arguments. Each command is of the form:

```
      \command{arg}
```

For example,

```
      \title{ this is my title }
```

Certain commands are required, many more are optional. The template created by `prompt` will contain the required sections.

Key Sections in the Rd file

The output of our prompt file identifies 12 sections. Only the header sections – `\name\alias\title\description\usage` - and the footer section – `\keyword` are required. The commands for these sections are highlighted in blue above.

In addition to the required sections, we recommend adding sections for arguments, value returned by the function, author and examples. The commands for these are marked in red above.

Thus, an archtypical Rd file would have 10 sections, each starting with the commands below:

```
\name
\alias
\title
\description
\usage
\arguments
\value
\author
\examples
\keyword
```

In addition, I have used a details, reference, and seealso sections. I recommend that you start simple and then add more sections later after you have your Rd files debugged.

Now let's modify the template file for `riChisq`. To do so requires some additional knowledge of the Rd format.

Key Rd Formatting Commands

There is an extensive menu of formatting commands documented in chapter 2 of WRE. I recommend starting with just a few. After you have built-up confidence, then you can read WRE and try them out.

Switch to "typewriter" font (R code):	`\code{ riChisq }`
Insert URL:	`\url{ http://www.r-project.org/ }`
Insert email:	`\email{ peter.rossi@ChicagoGsb.edu }`

The `\arguments` command uses the sub-command `\item` to organize a list. If your return value from the function is also a list you will need to use the `\item` command as well. For example, `riChisq` returns a list of two items. This means in the value section you will insert

```
        \item{nu}{ degrees of freedom value}
        \item{iChisq}{ draw from inverted Chi-squared distribution}
```

NOTE: there are NO spaces between the bracket that closes the item name and the open bracket for the argument text string!

`riChisq.Rd` *Modified*

Below is an edited version of the `riChisq.Rd` file. Note that I have added a simple example of using the function. Text that has been added/replaced is in red.

```
\name{riChisq}
\alias{riChisq}
\title{ Draw from the Inverted Chi-squared Distribution }
\description{
  \code{riChisq} draws from the Inverted Chi-squared Distribution.
  A simple C function is called which uses R internal functions to
  generate normal random variables, square them and add them up.
  This function is inefficient and for illustration purposes only.
}
\usage{
riChisq(nu)
}
\arguments{
  \item{nu}{ degrees of freedom parameter }
}
\value{
  a list containing:

  \item{nu }{degrees of freedom parameter }
  \item{iChisq }{Inverted Chi-squared with nu d.f. draw }
}
\author{ Peter Rossi \email{peter.rossi@ChicagoGsb.edu}}}
\examples{
##
  set.seed (666)
  riChisq(10)
}
\keyword{ distribution }
```

The keywords are found in the file, `C:\Program Files\R\rwYYYY\doc\keywords.db`, in the standard installation of R. You have to choose one or more appropriate for this function.

Debugging the documentation files

The error messages generated by R for the documentation files can be cryptic. For this reason, great care must be exercised writing the documentation.

There are two key ideas: 1). Create the Rd file incrementally. That is, don't try to write an elaborate file in one fell swoop. Write the file incrementally, testing each incremental change. 2. work on only one function at a time. Resist the temptation to write many Rd files and then expect to debug them simultaneously.

It is a bad idea to use the `R CMD check` script to check Rd files at first. Use `R CMD Rd2txt` first. If I have documentation file, `myfun.Rd`, use the following command at the prompt in the command window:

```
R CMD Rd2txt myfun.Rd
```

The file will be translated and displayed in that window.

If this works with no errors, use the command:

```
R CMD Rdconv –t=html –o=myfun.html myfun.Rd
```

to produce an html version. Use your internet browser to display the file. Inspect the output from these commands closely to see if all of the sections you have defined are displayed!

Tips on Writing Rd files

1. Like TeX/LaTeX, the Rd formatter has its own ideas of how a document should be formatted. Don't try to over-ride this too much with `\cr` (new lines). Insert a blank line to separate groups of text.
2. Don't forget that % is the comment command in Rd files (like # in R functions). This means you have to be careful about using matrix multiplication in an example definition. Use the escape character "\" to insert %, e.g. C=A\%*\%B.
3. Avoid the $, #, _, <, >, and | characters.
4. Don't forget to close your "{ }". If you are using VIM, you can use the "%" key to toggle between left and right braces to insure you have enough. Invariably, you forget the closing brace.
5. My own view is that insertion of LaTeX equations in the document is more trouble than it is worth, particularly in light of the fact that the documents displayed via the help or ? command are plain text versions. The LaTeX equations don't translate very well into plain text. This can be avoided with some thought. If you really have elaborate equations to refer to, include a link to a pdf file with them.
6. The Rd processor is very sensitive about spaces between the command and the beginning of the argument,
   e..g \author{ Peter Rossi } will work, but \author { Peter Rossi} will not!
7. If you have a set of related functions, use the \code{\link{function_name}} to insert links between the documentation files.

Step 7: Check the Package

In theory, checking the package is simple. Open the command prompt window and CD to the directory level above the start of the package directory tree.

Use `R CMD check` to "check" your package.

`R CMD check test` at the command prompt will fire up a script which will create the document files in text, LaTeX, and html formats, compile the source code and create the DLL, check for inconsistencies and various errors, run the examples, and finally build a manual in dvi format.

The best way to avoid R CMD check errors is to check your code beforehand by:

1. sourcing all functions and running examples of each.
2. compiling and creating a DLL as in step 5 above
3. running R CMD Rd2txt and R CMD Rdconv on all Rd files

If these three steps are taken, R CMD check errors will be greatly limited.

R CMD check will build the package in a directory called <package>.Rdcheck which will e created at the same directory level as the R CMD is invoked.

Finally, make sure that the time to run the examples is less than 2 minutes. This can be checked by looking at the file <package>-Ex.Rout in the Rdcheck directory.

Step 8: Build the Package

Use the command `R CMD build <package>` to build the package. This creates a tar file which can be submitted to CRAN. You can also use `R CMD INSTALL <package>` to install your package to your local R program files location.

Step 9: Upload to CRAN

Follow the instructions on the CRAN web page and upload the tar file created by INSTALL.