Package Development in Windows

Duncan Murdoch

Department of Statistical and Actuarial Sciences
University of Western Ontario

Originally from August 13, 2008;
updated November 23, 2012

Outline

Outline

R is mostly packages!

- R ships with 14 base packages, and 15 recommended packages.
- CRAN contains about 4100 packages, Bioconductor has many more.
- There are many other packages not in these repositories.

What is in a package?

- Permanent R objects: functions, data, etc.
- Man pages and vignettes documenting these objects.
- External code in C, C++, Fortran, Objective C, etc. to implement some of the functions, or link to external libraries or programs.
- Tests to help to keep the code working as R evolves.

Packages, libraries, repositories?

- We use

  ```
  > library(foo)
  ```

  to load the package and put it on the search list, but a package is not a library.
- A library is a collection of packages installed on your system. Use

  ```
  > dir.create("newlib")
  > .libPaths("newlib")
  ```

  to create a new one, and add it as the first place to look.
- A repository is a collection of packages like CRAN, usually available online. Use `install.packages()` to install a package from a repository into your library.

Not everyone uses packages

Packages are great, but they aren't the only ways to save code and data. There are also

- binary images
- R scripts
- vignettes

Saving a binary image

Use ***save()*** or ***save.image()*** to save R objects to a file.

- Saved images are portable: all equal or newer versions of R on all platforms should be able to read then.
- These are very easy to create: just answer ***Yes*** when quitting!
- ***save()*** on a single large object is easy, and it may be easier to reload it than to recreate it.

What's wrong with saving your workspace?

- Saved images are hard to work with: they are black boxes outside of R.
- It is very easy to save more than you intended, and get bloated saves, and unintended interactions.
- It is easy to forget how some objects were created.

Working with scripts and vignettes

You can put R code in a plain text file, and use copy and paste or `source()` to read it into R.
You can write a vignette, containing a mixture of LaTeX (or other) text and R code.

- These are easy to transport and edit on any platform.
- It is easy to see what's there (if you format your code nicely...)
- You can have a permanent record of how research results were produced.
- Vignettes using Sweave are great for explaining code.

What's wrong with scripts?

- It is hard to re-use parts of scripts.
- Cut and paste is error prone.
- It is hard to remember which earlier part of a script needs to be re-executed, and which doesn't.

So why packages?

- Packages combine the good aspects of saved images and scripts.
- R packages can be distributed to others.
- R tools support quality control checks.

Outline

Windows is not Unix

- Much R development occurs on Unix-like machines using GNU tools.
- Linux and Mac OS X development use the same tools.
- The tools are generally available for MS Windows, but it takes some work to find them.
- Simple packages don't need anything but R to build.
- We've collected all the tools for complicated packages for you.

The Rtools collection

`http://CRAN.r-project.org/bin/windows/Rtools`
has news and downloads of the main tools. We have packaged most of them into an installer, **`Rtools216.exe`**.
Download and run the installer, to get:

1. Unix-like command line tools
2. MinGW64 gcc compilers (C, C++, Fortran, Objective C)
3. Some other files needed to build R itself

Command line tools

These are a number of utilities to make Windows look more like Unix. These tools will run in the Windows shell (CMD), or in a Cygwin shell. (We include the Cygwin DLLs.)

- The GNU make utility: make
- A simple Bourne shell to run shell scripts: sh
- Tools for working with archives of files: tar, gzip, zip, unzip
- Unix-like file and system commands: cat, cp, date, echo, find, ls, mkdir, mv, rm, rmdir
- Tools for text manipulation: cut, diff, egrep, gawk, grep, sed, sort, tidy, touch
- Various others: basename, cmp, comm, expr, ln, makeinfo, md5sum, od, pedump, rsync, texindex, tr, uniq

The MinGW64 compilers

On Windows we build R using the MinGW64 release of the gcc compiler suite, and it's easiest to build packages using the same compilers (so it is the only one we support). These are *not* the Cygwin compilers: those are incompatible with R. Cygwin now distributes a copy of R built with the Cygwin tools; it doesn't work.

Missing pieces

With the Rtools installed, you can build packages, but you won't be able to build some types of documentation.

PDF help and vignettes You will need a copy of LaTeX to build the PDF documentation and vignettes. I recommend MikTeX, version 2.9, available from `http://www.miktex.org`.

MikTeX is not required, but I recommend getting it.

Running the installers

The Rtools and MikTeX both come with installers. I recommend installing Rtools *last*, because it is quite sensitive to the system PATH, and other installers might mess it up.

The dreaded PATH

- The PATH is a list of directories on your system giving the search order for commands. Because the Rtools include so many commands, it is essential that they appear very early in the PATH, or other versions of those commands will be found instead.
- In Windows, there is a system PATH, which is set for any program started from Explorer. You can change the PATH within a CMD shell, or within R, to affect programs started locally.
- I recommend that for simplicity you let Rtools set the system PATH, but there's the possibility of conflicts with other programs.

How to set the PATH

Rtools will offer to edit the system PATH by adding these directories at the start:

```
c:\Rtools\bin;
c:\Rtools\gcc-4.6.3\bin;
```

You should also add (if its installer didn't) the directory for MikTeX, as well as the directory for the R binaries. For example,

```
c:\texmf\miktex\bin;
c:\R\R-2.15.2\bin
```

Outline

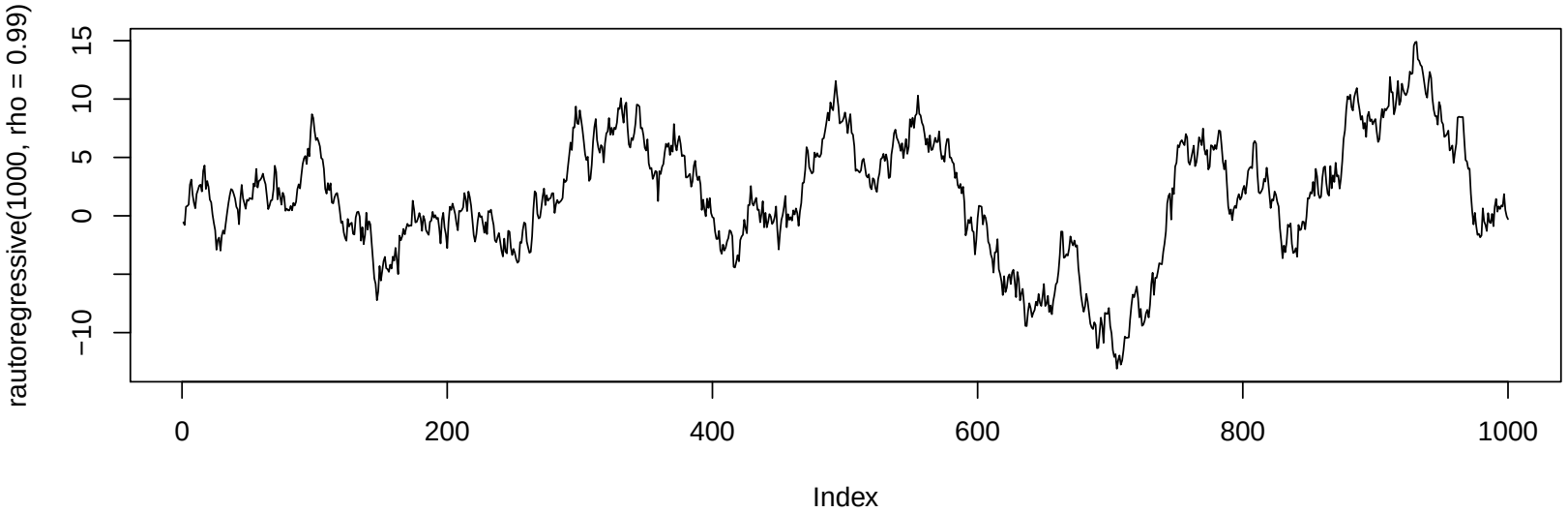
Starting from code

We'll write a package to hold an autoregressive simulator.

```
> rautoregressive <- function(n, rho=0){
+   result <- double(n)
+   innov <- rnorm(n)
+   result[1] <- innov[1]
+   for (i in seq_len(n-1)+1) {
+     result[i] <- rho * result[i-1] + innov[i]
+   }
+   return(result)
+ }
```

Checking our code...

```
> set.seed(123)
> plot(rautoregressive(1000, rho=0.99), type='l')
```



Note: ***arima.sim*** is better!

A skeleton package

```
> package.skeleton("Rauto", "rautoregressive",
+                  path="c:/temp")


Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in
'c:/temp/Rauto/Read-and-delete-me'.
```

What got created?

```
> list.files("c:/temp/Rauto", recursive=TRUE)
[1] "DESCRIPTION"
[2] "man/Rauto-package.Rd"
[3] "man/rautoregressive.Rd"
[4] "NAMESPACE"
[5] "R/rautoregressive.R"
[6] "Read-and-delete-me"
```

Read-and-delete-me

```
* Edit the help file skeletons in 'man',
  possibly combining help files for multiple
  functions.
* Edit the exports in 'NAMESPACE', and add
  necessary imports.
* Put any C/C++/Fortran code in 'src'.
* If you have compiled code, add a
  useDynLib() directive to 'NAMESPACE'.
* Run R CMD build to build the package
  tarball.
* Run R CMD check to check the package
  tarball.

Read "Writing R Extensions" for more
  information.
```

man/rautoregressive.Rd

```
\name{rautoregressive}
\alias{rautoregressive}
%- 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{
rautoregressive(n, rho = 0)
...
```

The edited version

We now edit the `*.Rd` files, producing something like this:

```
\name{rautoregressive}
\alias{rautoregressive}
\title{ Simulate an AR(1) process }
\description{
This function simulates a Gaussian AR(1)
process, started from zero.
}
\usage{
rautoregressive(n, rho = 0)
}
\arguments{
...
```

The NAMESPACE file

- The **NAMESPACE** file describes which functions in your package are visible to others. The default file contains just one line,

  ```
  exportPattern("^[[:alpha:]]+")
  ```

  which says that all visible objects are exported. That's reasonable for a start, but in a more complicated package you'll want to hide some of the implementation.
- A **NAMESPACE** guarantees the search order for functions. Without one, your package may behave differently depending on which packages are attached first.

The DESCRIPTION file

The **DESCRIPTION** file is key: you need to edit it too.

```
Package: Rauto
Type: Package
Title: Simple demo package
Version: 1.0
Date: 2012-11-23
Author: Duncan Murdoch
Maintainer: Duncan Murdoch <murdoch@stats.uwo.ca>
Description: A simple demo of building
    a package in Windows.
License: GPL (>= 2)
```

Installing your package

- There are two ways to install a custom package: from the directory, or by building a tarball first, and installing from that.
- Start by installing directly from the directory.

Installing...

Use

```
> install.packages("c:/temp/Rauto", repos=NULL,
+                  type="source")
```

You can also run `R CMD INSTALL Rauto` in a command shell. In either case, **`Rauto`** is now available:

```
> rm(rautoregressive)
> library(Rauto)
> rautoregressive(5)
[1] -0.99579872 -1.03995504 -0.01798024
[4] -0.13217513 -2.54934277
```

Testing... I

One of the strengths of R is its quality control system. Use it! To check for common errors in a package, use

```
C:\temp R CMD check Rauto

* using log directory 'C:/temp/Rauto.Rcheck'
* using R version 2.15.2 Patched (2012-11-19 r61131)
* using platform: i386-w64-mingw32 (32-bit)
* using session charset: ISO8859-1
* checking for file 'Rauto/DESCRIPTION' ... OK
* checking extension type ... Package
* this is package 'Rauto' version '1.0'
* checking package namespace information ... OK
* checking package dependencies ... OK
* checking if this is a source package ... OK
```

Testing... II

```
* checking if there is a namespace ... OK
* checking for executable files ... OK
* checking whether package 'Rauto' can be installed ...
* checking installed package size ... OK
* checking package directory ... OK
* checking for portable file names ... OK
* checking DESCRIPTION meta-information ... OK
* checking top-level files ... OK
* checking for left-over files ... OK
* checking index information ... OK
* checking package subdirectories ... OK
* checking R files for non-ASCII characters ... OK
* checking R files for syntax errors ... OK
* checking whether the package can be loaded ... OK
* checking whether the package can be loaded with stated
```

Testing... III

```
* checking whether the package can be unloaded cleanly .
* checking whether the namespace can be loaded with stat
* checking whether the namespace can be unloaded cleanly
* checking loading without being on the library search p
* checking for unstated dependencies in R code ... OK
* checking S3 generic/method consistency ... OK
* checking replacement functions ... OK
* checking foreign function calls ... OK
* checking R code for possible problems ... OK
* checking Rd files ... OK
* checking Rd metadata ... OK
* checking Rd cross-references ... OK
* checking for missing documentation entries ... OK
* checking for code/documentation mismatches ... OK
* checking Rd \usage sections ... OK
```

Testing... IV

```
* checking Rd contents ... OK
* checking for unstated dependencies in examples ... OK
* checking examples ... OK
* checking PDF version of manual ... OK
```

Success!

Other R CMDs

Some other related commands:

R CMD build Rauto

Build a `*.tar.gz` source tarball, for use on any system.

R CMD INSTALL --build Rauto

Install, and build a `*.zip` binary package for use on Windows only.

R CMD Rd2dvi --pdf Rauto

Collect manual pages into a PDF manual.

R CMD --help

Show the full list of CMDs.

Linking external code

- R code is convenient, and can be very fast when operations are done on large vectors, but it is slow in some operations (e.g. loops).
- Compiled C, C++, or Fortran are much less convenient, but are much faster in loops.
- There are several interfaces for external code: `.C()`, `.Fortran()`, `.Call()`, and `.External()`.
- `.C()` and `.Fortran()` are the easiest to use and are very similar; we'll rewrite our function using `.C()`.

Rauto/R/rautoreg.R:

```
rautoreg <- function(n, rho=0) {

    # Pass the innovations in the results vector,
    # return the results in the same place

    .C("rautoregC", as.integer(n),
                    as.numeric(rho),
           results = rnorm(n))$results
}
```

Rauto/src/rautoreg.c:

```
#include <R.h>

void rautoregC(int *n,
               double *rho,
               double *results)
{
    /* Use Rprintf for debugging messages */
    Rprintf("In C, n=%d, rho=%f\n", *n, *rho);
    for (int i=1; i < *n; i++) {
        results[i] = (*rho)*results[i-1]
                     + results[i];
    }
}
```

The **NAMESPACE** file needs to mention the external code, which the installer will have compiled into **Rauto.dll**.

```
exportPattern("^[[:alpha:]]+")
useDynLib(Rauto)
```

Shut down R, re-install the package, and see the results:

```
> library(Rauto)
> set.seed(123)
> system.time(x <- rautoregressive(1000000, 0.99))
   user  system elapsed
   4.10    0.00    4.11
> set.seed(123)
> system.time(y <- rautoreg(1000000, 0.99))
In C, n=1000000, rho=0.990000
   user  system elapsed
   0.11    0.01    0.13
```

Outline

Going further

- R is very flexible: I have only shown one workflow. There are many others!
- The *Writing R Extensions* manual is the authoritative reference.
- Once things are set up properly, all platforms are very similar.