Making a package from base R files

John C. Nash * Arkajyoti Bhattacharjee †
based on communications from Duncan Murdoch

11/06/2021, revised 19/01/2022

Background

This article tries to explain an approach to developing alternative versions of functions which are in the distributed base of R. Our interest was in developing improvements to the `nls()` function and related features in R as part of a Google Summer of Code project for which Arkajyoti Bhattacharjee was the funded student. However, `nls()` has many tentacles involving a number of files and functions that may or may not be called as `nls()` is executed.

Part of the difficulty in carrying out such development of alternative versions is that one needs to be able to execute the new variants in parallel with the existing ones. A heavy-effort approach would be to have separate full sets of R code and build each system and run them separately. That is, we want to have two or more versions of R in the same computing system.

There are several ways to do this:

- build or install the two versions of R in separate directory trees and invoke them with separate launch scripts or programs, such as described in https://support.rstudio.com/hc/en-us/articles/215488098-Installing-multiple-versions-of-R-on-Linux;
- use virtual machines with different R versions;
- run the different R versions in containers, e.g., Carl Boettiger and Eddelbuettel (2017);
- move the required functionality to a package that can be loaded as needed to mask the base functionality and replace it with the functions we wish to test or use. This last approach is discussed here.

We did attempt to provide R functions to switch between alternative sets of working functions – e.g., `ReplaceNLS.R` and `RestoreNLS.R` – but this approach was not promising, and not pursued.

Duncan Murdoch suggested, and generously provided an almost-complete prototype of, a package of interlocked functions that provide the set of `nls()` capabilities. This package `nlspkg` was copied to `nlsalt` then modified to provide the new capabilities or new expression of the code. The new capabilities or code, such as a function `Anyfn()` can be tested side by side with a "standard" version (if one exists) by use of the double colon syntax, namely,

```
nlspkg::Anyfn() # existing functionality or code
# and
nlsalt::Anyfn() # New functionality of code.
```

Packaging from base R files

Duncan Murdoch described his process of developing the prototype of `nlspkg`:

*retired professor, Telfer School of Management, University of Ottawa
†Department of Mathematics and Statistics, Indian Institute of Technology, Kanpur

```
Identify the main R functions I wanted in the package, and copy that
code into ./R.

Identify the C/Fortran functions that are referenced, and copy the
source for that into ./src.

Try to build, and get lots of errors about missing functions. Find
those and copy as above.

There was one case (the `%||%` operator) where I thought the code used
an unexported object from utils; the base-internals.R function just used
::: to get it, but a better solution would be to copy the source (which
is pretty trivial), and it turns out, is what stats did (the duplicate
source is in AIC.R).

Some files contained functions or methods that weren't relevant to us
(e.g. confint.lm).  I commented those out so I didn't need to worry
about the functions they used, and imported the generic (i.e. confint)
from stats.

In the C code, we were lucky because the nls code only made use of one C
function that's not in the API:  R_NewEnv. (That was called from
numeric_deriv.) There's an R function to do that, so I called it and
passed the result in.

Finally, once it would build, I tried running R CMD check.  This
identified all the missing help pages, which I copied over.  They didn't
need any modifications.

Because nls isn't really base functionality, all of this was a lot
easier than if I had tried to move something that works more with the
internals.  That's probably true for most functions in stats, but would
not be true for other base packages (except datasets).
```

After we downloaded the ZIP file of Duncan's package from his GitHub repository, we built and checked the package. There were two WARNING messages:

```
* checking DESCRIPTION meta-information ... WARNING
Non-standard license specification:
  GPL-2, GPL-3
Standardizable: FALSE
* checking for missing documentation entries ... WARNING
Undocumented code objects:
  ‘C_bvalus’ ‘C_lowesp’ ‘C_lowesw’ ‘C_nls_iter’ ‘C_numeric_deriv’
  ‘C_port_ivset’ ‘C_port_nlminb’ ‘C_port_nlsb’ ‘C_pppred’ ‘C_rbart’
  ‘C_setppr’ ‘C_setsmu’ ‘C_smart’ ‘C_supsmu’ ‘anova.nls’
  ‘anovalist.nls’ ‘coef.nls’ ‘confint.nls’ ‘deviance.nls’
  ‘df.residual.nls’ ‘fitted.nls’ ‘formula.nls’ ‘logLik.nls’ ‘nlsModel’
  ‘nlsModel.plinear’ ‘nls_port_fit’ ‘nobs.nls’ ‘plot.profile.nls’
  ‘port_cpos’ ‘port_get_named_v’ ‘port_msg’ ‘port_v_nms’ ‘predict.nls’
  ‘print.nls’ ‘print.summary.nls’ ‘profiler’ ‘profiler.nls’
  ‘residuals.nls’ ‘summary.nls’ ‘vcov.nls’ ‘weights.nls’
All user-level objects in a package should have documentation entries.
See chapter ‘Writing R documentation files’ in the ‘Writing R
Extensions’ manual.
```

The first WARNING can be overcome by changing the License line in the package DESCRIPTION file from

`License: GPL-2, GPL-3`

to

`License: GPL-2 | GPL-3`

as noted in https://www.r-project.org/Licenses/.

The warnings about documentation come from two sources:

- many of the `nls()` features parallel generic functions such as `residuals`, `predict`, `coef`, and so on, so are generically documented in R.
- the rest of the "undocumented" functions are C functions (identifiable easily as their names contain underscore characters). It is these functions that our "Improvements to nls()" project aims to replace where it is sensible to do so with R functions. They may then be documented as needed.

Name issues – IMPORTANT

It is worth mentioning that the NAMESPACE file provided by Duncan is an important part of the packaging. It could be worthwhile to document each element in that file needed for a package such as `nlspkg`, but at the time of writing, this is yet to be carried out. However, we must note that in renaming the package from `nlspkg` to something else, the change of name must be mirrored in the line

`useDynLib(nlspkg, .registration = TRUE, .fixes = "C_")`

where `nlspkg` in this line must be changed appropriately.

We also noted that the `S3method()` entries of the NAMESPACE files are needed to link the generic functions to similar modeling functions in R. Thus `anova`, `coef`, etc. are declared as S3 methods.

As a convenience to get the package running, Duncan used the shortcut

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

to ensure functions were made available to the user. This makes visible all the internal C functions, named `C_(something)`. We have suppressed this blanket declaration and explicitly exported the `nls` and `numericDeriv` functions. One or two other routines may need to be exported to allow full functioning of the package.

We note that name change is ALSO required in `src/init.c` in the line

`void attribute_visible R_init_nlspkg(DllInfo *dll)`

For version control, the file `.gitignore` has two entries: one is `*.0` and the other, initially, was `nlspkg.so`. We changed the latter to `*.so` and this appears to be satisfactory for general use.

Developing an alternative set of functions and an alternative package

We wished to build our alternative package by replacing calls to C functions with R work-alike functions. A preliminary example that already is set up and has passed preliminary checks is the function `numericDeriv()` from the file `R/nls.R` with called elements from the file `src/nls.c`. Note that some code needs to be commented out or deleted from the C file(s) and headers to avoid errors when building and checking the alternative package.

To prepare the `nlsalt` package a directory of that name was created and the contents of the `nlspkg` directory copied over. The DESCRIPTION file was edited to provide the `nlsalt` name and the result checked with build and CHECK.

At this stage, we can begin to substitute for calls to C code and remove that code once replacement R material is present. Frequent build and CHECK cycles are helpful.

For RStudio users, note that each package directory needs its own (name).Rproj file, created using RStudio's "File/New Project" etc. dialog. However, the package directories with their Rproj files can be within the umbrella version control project `improvenls` for the Git version control. (We used a GitLab repository.)

Windows issue

While all the above worked well in a Linux environment, when we tried running the build and CHECK in a Windows environment, we found errors. (JN normally runs in Linux Mint 20.x, but AB uses Windows 10. Each of us has a VirtualBox VM for the alternative OS to allow for checking of our work.) There is a load error for functions from the BLAS (Basic Linear Algebra Subroutines) collection in a 32 bit Windows architecture. This is revealed by the RStudio "Build / Clean and Rebuild" feature. Using the `CMD.exe` terminal window showed that `R CMD build` worked, but that `R CMD INSTALL` gave the load errors.

After some exchange of emails with others and with the R-package-devel mailing list, a Google search using terms suggested in the emails found the posting

https://stackoverflow.com/questions/42118561/error-in-r-cmd-shlib-compiling-c-code.

This suggested that a file `Makevars.win` is needed in the `src/` directory of the package(s), and that this file should contain just the following line.

```
PKG_LIBS = $(LAPACK_LIBS) $(BLAS_LIBS) $(FLIBS)
```

We did, however, note that it is not clearly indicated that this file needs to be in `src/`, nor (until revealed by `R CMD check`) that the file must be `Makevars.win` and not `makevars.win`. There is a lot of material on `Makevars` in the Writing R Extensions manual (R Core Team (2013)), but these points do not seem to be presented, or at least are not obvious.

Our packaged version of `nls()` from July 2021 is at

https://gitlab.com/nashjc/improvenls/-/tree/master/nlspkg.

Note that this is drawn from the version of R of that date (R 4.1.0), so will not necessarily reflect the distributed version of R when this document is read.

Downstream benefit

The approach to replacing functionality in base R suggested here has a follow-on benefit should some or all of the improvements we hope to be able to make are eventually included in R's distribution files. That benefit is that the initial package can be renamed to, for example, `nlslegacy` so that R users can, should they need, use legacy capabilities. Indeed, from preliminary timings of some functions, it appears that pure R code runs 2-3 times slower than the mixed R and C versions. On the other hand, the pure R code in our early cases is much shorter and, we suspect, much easier to maintain. Furthermore, it simplifies the packaging of base R for alternative interpreter or compiler versions of the R language.

Acknowledgements

It is a pleasure to note the help and encouragement of Duncan Murdoch, Dirk Eddelbuettel and Brett Klamer.

References

Carl Boettiger, by, and Dirk Eddelbuettel. 2017. "An Introduction to Rocker: Docker Containers for R." *R Journal of Statistical Software* 9 (2). https://journal.r-project.org/archive/2017/RJ-2017-065/RJ-2017-

065.pdf.

R Core Team. 2013. “Writing R Extensions.” http://CRAN.R-project.org/doc/manuals/R-exts.html.