How to Make an R Package for Windows With Native Routines in Fortran 95

JOSEPH L. SCHAFER
The Methodology Center
The Pennsylvania State University
University Park, PA

July 21, 2008

Abstract

The standard mechanism for distributing statistical software to R users is to build an R package. However, the usual package-building mechanism assumes that native routines are written in C or FORTRAN 77; code written in Fortran 95 is not supported. This document shows how to create an R package for Windows when the package contains routines in Fortran 95. I show how build and distribute the package in binary form, so that users of R for Windows can download it as a `.zip` file and install it just as they would any package from CRAN. This procedure is not officially sanctioned by the R Development Core Team, because the resulting R package is neither platform-independent (it works only on Windows) nor open-source (the actual Fortran code need not be included). Nevertheless, this procedure may be very useful for researchers who wish to include native Fortran 95 in an R package or who, for whatever reason, do not wish to distribute their Fortran source.

This work was supported by National Institute on Drug Abuse, 1-P50-DA-10075.

1 Preliminaries

1.1 Knowledge assumed

R is a free, open-source statistical computing environment used by methodologically sophisticated researchers around the world. Although R was and is developed on Unix, it runs on other operating systems including MacOS and Windows. With few or no changes, R code should produce identical results on all platforms. R is very flexible. It is a high-level programming language that allows users to write their own functions and interface with native routines written in C or Fortran. The developers of R have also provided a set of utilities for users to create and document their own packages and distribute them through the Comprehensive R Archive Network (CRAN), the network of websites that hold the R distributions and code.

Throughout this document, I assumes that the reader already has basic familiarity with R at, say, the level of *An Introduction to R* by Venables, Smith and the R Development Core Team. That manual is available from CRAN and is included in the basic R distribution. In particular, I assume that the reader knows how to create R functions. I also assume that the reader is familiar with the R function `.Fortran`, which allows R to interface with native routines written in Fortran.

1.2 Packages and libraries

A *package* is a set of routines that is not part of the R base distribution but has been arranged and documented in a standard fashion. Once a package has been installed on an R user's system, it becomes a *library* that can be attached and used in any R session. R functions that are located in libraries look and feel just like the R functions in the base distribution; they have their own help files, for example.

Because of the open-source and platform-independent nature of R, users who create their own packages and submit them to CRAN are expected to provide the source code for all their routines. This source code is written in R, and parts of it may also be written in C or Fortran.

Recent versions of R will download user-contributed packages from CRAN and install them on a computer automatically. The Windows GUI version of R has a "Packages" menu that allows you to do this in different ways. One way interactively prompts you to select the CRAN mirror site and select the package. If the package that you are trying to install depends on (requires) other packages, those packages are downloaded as well. Another way

to install a package is to go CRAN with a web browser (`http://cran.r-project.org/`) and download a precompiled binary version of the package you wish to install. If you are running R on Windows, you would visit CRAN, go to the "Windows" area, then to "Contrib," download the desired package as a compressed (`.zip`) archive and save it to your computer. Then, in an R session, you would go to the "Packages" menu, choose "Install package(s) from local zip files...", and select the `.zip` archive that you just downloaded.

Once a package has been installed on your computer, the functions contained in it will be available for use in that R session and in future sessions. Before using the package, however, you have to attach it with the `library()` function. For example, to use functions in the package `grid`, you have to issue the command

```
> library(grid)
```

in that R session and in any future R session. Alternatively, in the Windows GUI version of R, you can go to the "Packages" menu, select "Load package..." and then select `grid` from the list of locally installed packages.

Note to users of Windows Vista. Downloading and an installing an R package is straightforward if you are running Windows XP. If you are running Windows Vista, however, you may run into trouble for the following reason. By default, R has probably been installed in a subdirectory of `C:\Program Files` or `C:\Program Files (x86)`. By default, packages are installed in subdirectories of the R directory. As a Vista user, you may not have sufficient privileges to create new subdirectories and install files there. If you have trouble installing an R package under Vista, there are two ways to work around it.

- Run R with Administrator privileges. To do this, *right-click* an R shortcut and select "Run as administrator". Then you should be able to install packages without any trouble. Once a package has been installed, you do not need Administrator privileges to attach or use it. So you can quit the R session, start another session, and use `library()` in the usual way.
- Another way to work around the Vista security restrictions is to remove R from your computer and then re-install it in an area that Vista considers to be your personal disk space. That area depends on the username of your account. For example, if your account username is `jls`, then your personal disk space is `C:\Users\jls`. When you install R, you will be asked where it should be put. Instead of the default, which is something like `C:\Program Files\R`, you should select `C:\Users\jls`. This method of getting around the Vista security restrictions may be necessary if you are going to be building R packages. The R building process seems to run into trouble under Vista unless R is has been installed in your own personal space. I'll explain more about that in Section 5.

1.3 Why build an R package?

If you have written one or more R functions that you would like to share with others, then you should strongly consider turning them into an R package. Packages are the officially sanctioned way to document and distribute extensions of R. Building an R package is tricky for a novice. In simple cases, you can work around the package mechanism, e.g. by just sending someone the source code for your R function and some notes on how to use it. But the R community has a strong culture, and sooner or later an experienced R user or developer will tell you, "You know, you really should make this into an R package..." The official guide to creating R packages is the document *Writing R Extensions* by the R Development Core Team, which is also available from CRAN. To a novice, this document can be dense and impenetrable, and it does not have many examples. If you want to make R packages, you should have that document available as a reference. But I do not assume that you have read and absorbed all the material in that document.

1.4 Tools you will need to build an R package for Windows

The utilities needed for creating an R package for Windows are called "Rtools." These are essentially the same utilities that you would use to build R from source and install it on a Windows computer. Most users of R for Windows do not build R from source; they download and install a precompiled binary version specifically for Windows. Most users do not make their own R packages, either. So if you are a typical R user, chances are that you do not have Rtools. But you can easily download and install Rtools. Before attempting to make your own package, make sure that you have *the most recent version of R* installed on your computer. Then install *the most recent version of Rtools.* Rtools is a self-installing executable (`.exe`) file which you can simply download and run. The website for Rtools is `http://www.murdoch-sutherland.com/Rtools/`.

Rtools includes a basic ("vanilla") version of Perl, a platform-independent scripting language that is commonly used for manipulating text. It also includes MinGW, a minimal set of development tools from the Free Software Foundation.

Rtools assumes that you have LaTeX, which are needed to create `.dvi` and `.pdf` versions of the documentation. The most popular version of LaTeX for Windows is called MiKTeX, and it is free. Before getting Rtools, you should download and install MiKTeX on your computer.

When you install Rtools, it will be best to put it along a path that has no embedded spaces. Instead of putting it in `C:\Program Files\...`, it's better to put it in `C:\Rtools`.

Rtools utilities are accessed from a Windows command prompt. For this to work properly,

you need to make sure that the directories where the Rtools programs are located are included in the environmental variable `PATH`. Recent versions of the Rtools installer may automatically modify your `PATH` variable to include these directories. If they do not, you will need to alter `PATH` manually. The details of how to do this vary slightly from one version of Windows to the next, but you can usually do it by going to "Control Panel" and "System." If you go to "Control Panel" and "System," look for "Properties" or "Advanced System Settings" and then for "Environment Variables," and you should be able to find and edit the variable `PATH`. There may be two `PATH` variables, one for you (the user) and one for the system. You may edit either one. Go to the beginning of the character string and add each of the Rtools `bin` directories to the search path, followed by semicolons. For example, if you have installed Rtools in `C:\Rtools`, then you will need to add the following strings to the beginning of `PATH`:

```
C:\Rtools\bin;
C:\Rtools\MinGW\bin;
C:\Rtools\perl\bin;
```

If you have already installed MiKTeX then, chances are, the directory containing the MiKTeX executables (such as `latex.exe`) will already be included in the user `PATH` or the system `PATH`. You should check to see if it's there and add it if it's not. For example, you should see something like

```
C:\Program Files\MiKTeX 2.7\miktex\bin;
```

somewhere along the search path. If not, search your computer to find the location of `latex.exe` and add that directory to `PATH` followed by a semicolon.

While you're at it, make sure that the directory containing the R executables (such as `R.exe`) is also in `PATH`. For example, if you have installed R in `C:\Users\jls`, it will be something like

```
C:\Users\jls\R-2.7.0\bin;
```

If you are not sure where the R executables are kept, search your computer for the file `R.exe`.

Finally, to build an R package on a Windows computer, you will also need a utility called the Microsoft HTML Help Workshop. This is a free product from Microsoft that allows developers to create and compile Help files and documentation in the HTML format used by many Windows applications. If you have Microsoft Visual Studio installed on your computer, then you may already have the HTML Help Workshop. If not, you can easily download it from Microsoft and install it on your computer. It's a good idea to install it along a path that has no embedded spaces. For example, instead of installing it in the default place, which is something like `C:\Program Files\HTML Help Workshop`, you can install it

in `C:\HTML-Help-Workshop`. After installing the HTML Help Workshop, the directory that contains the executables (in particular, `hhc.exe`) needs to be added to `PATH`, as described above.

After installing all of these tools, make sure that you have set `PATH` correctly by doing the following. Open a Command Prompt Window and issue the following commands, one at a time.

```
cat --help
perl --help
R CMD --help
gcc --help
latex --help
hhc --help
```

Each of these commands should produce a list of options, like this:

```
D:\jls\software\Rpackage>hhc --help

Usage:   hhc <filename>
    where <filename> = an HTML Help project file
Example: hhc myfile.hhp
```

If you see a message like this,

```
'hhc' is not recognized as an internal or external command,
operable progran or batch file
```

then the software in question has not been installed correctly, or the `PATH` variable has not been set correctly.

2 Using Fortran 95 from R

2.1 Why call Fortran 95 routines from R?

With R, you can develop computational routines very quickly. But routines written in R may not *run* very quickly, especially if they have a lot of nested loops. Computationally intensive procedures can run much faster and handle bigger problems if they are written in a native language like C or Fortran.

At The Methodology Center, we promote the re-use of code by creating procedures in Fortran 95 that can be called from a variety of environments. This is the strategy outlined in the recent book by Lemmon and Schafer (2005). Fortran 95 is an excellent tool for such purposes, and many good commercial Fortran 95 compilers are available. With most compilers, it is easy to encapsulate Fortran routines as as Windows dynamic-link library (DLL) and call them from R, SAS, SPSS, Visual Basic, and other Windows applications.

To call a subroutine in a Fortran 95 DLL from a Windows version of R, you need to use the R function `.Fortran`. Details on how to do this are given in Chapter 6 of Lemmon and Schafer (2005) and in the R help file for `.Fortran`. You just have to make sure that the compiler options are set correctly, that the correct calling conventions are being used, and that actual arguments supplied by R have the correct storage mode. Finally, you have to make sure that any character string arguments being passed to Fortran have length 255, which is admittedly strange and sometimes inconvenient. But if you follow the guidelines in Lemmon and Schafer (2005), you should be able to do it without much difficulty.

2.2 A simple example using Intel Fortran

Here is an example of a very simple subroutine written in Fortran 95 that we will call from R. The source code for this routine is placed in a file called `test.f90`.

```
!############ File: test.f90 ##########################################
subroutine fortran_test( x, power, y, string )
   !DEC$ ATTRIBUTES DLLEXPORT,C,REFERENCE,ALIAS:"fortran_test_" :: fortran_test
   use program_constants
   implicit none
   ! declare arguments
   real(kind=our_dble), intent(in) :: x
   integer(kind=our_int), intent(in) :: power
   real(kind=our_dble), intent(out) :: y
   character(len=255), intent(out) :: string
   ! begin
   y = x**power
   string = "Hello R user."
end subroutine fortran_test
!######################################################################
```

The comment line `!DEC$...` is a directive to Intel Fortran, our particular compiler, that we want to create a DLL and that the subroutine `fortran_test` is to be exported (made callable from other applications). The `ALIAS` option is necessary because R implicitly adds an underscore to any external symbol (see Chapter 6 of Lemmon & Schafer, 2005).

The subroutine above makes use of the module `program_constants` whose code is placed in another file:

```
!########### File: constants.f90 ######################################
module program_constants
   ! Programming constants used throughout the program.
   ! Unlike most modules, everything here is public.
   implicit none
   public
   ! Define compiler-specific KIND numbers for integers,
   ! single and double-precision reals to help ensure consistency of
   ! performance across platforms:
   integer, parameter :: our_int = selected_int_kind(9), &
        our_sgle = selected_real_kind(6,37), &
        our_dble = selected_real_kind(15,307)
end module program_constants
!######################################################################
```

The rationale for defining the KIND parameters for integer and real variables by creating a module of constants is given by Lemmon and Schafer (2005), Sections 2.1.7, 4.1.2 and 4.2.2.

We create the DLL with Intel Fortran Version 10.1 by issuing the following commands at the Command prompt:

```
ifort /c /nothreads /align:sequence /names:lowercase /assume:underscore constants.f90
ifort /c /nothreads /align:sequence /names:lowercase /assume:underscore test.f90
link /dll test.obj constants.obj
```

(While your routines are still under development, you may wish to include the additional compiler directives `/check:bounds` and `/traceback` in the `ifort` commands, which may help you to debug your code if something goes wrong.)

If everything is done correctly, you should now see a file called `test.dll`. To call the subroutine from an R session, first load the DLL by issuing the command

```
> dyn.load("test.dll")
```

at the R prompt. Depending on where the file `test.dll` is located and where you started the R session from, you may need to supply a full path in the character string along with the filename. But be aware that the backslash "\" is inserted into a character string as "\\". You can see if the subroutine `fortran_test` loaded correctly by typing

```
> is.loaded("fortran_test")
```

to which R should respond `TRUE`. Then create actual arguments of the correct storage mode and length as follows.

```
> x <- as.numeric(10)        # double precision real
> power <- as.integer(2)     # integer
> y <- as.numeric(0)         # double precision real, to hold the result
> # create a blank string of length 255
> string <- ""
> for(i in 1:255) string <- paste( string, " ", sep="")
```

Now the actual call to Fortran:

```
> # call the Fortran routine, naming the arguments
> tmp <- .Fortran("fortran_test", x=x, power=power, y=y, string=string )
> # look at the resulting list
> tmp
$x
[1] 10

$power
[1] 2

$y
[1] 100

$string
[1] "Hello R user.
```

If you don't understand what is happening here, see the R help file for `.Fortran`.

The interface by which R communicates with Fortran DLL's is not sophisticated. If each argument that you supply to `.Fortran` does not exactly match the corresponding argument expected by the Fortran subroutine with respect to data type and dimensions, R will probably crash, and the error message that you get will be uninformative. R may also crash if you make any of the myriad programming mistakes in Fortran that lead to run-time errors (logarithm of a negative number, division by zero, subscript out of bounds, trying to access arrays that have not been allocated). A feature of DLL's is that they run within the same virtual memory space as the application that invokes them, so if something goes wrong inside the DLL, the whole process may be corrupted. For this reason, you need to be extra careful to follow good programming practices when you write the Fortran code, so that users of your package will not experience frustration.

2.3 Why is it hard to use Fortran 95 in an R package?

Calling a native Fortran 95 procedure from R is fairly easy. So it should also be easy to write an R function that calls a native Fortran 95 procedure and include that function in an R package. Unfortunately, it is not. Using Fortran 95 in an R package is apparently impossible if you follow the officially sanctioned procedures for authoring R packages. The reason is that R is a free, open-source, cross-platform program built entirely with free, open-source, cross-platform development tools. But free, open-source, cross-platform tools for compiling Fortran 95 are not yet ready for prime time.

R packages are supposed to be written in a generic way that makes them available to users who are running R on Unix, MacOS, Windows, or any other operating system. If you want to create your own package, you are supposed to write it in a careful manner described in *Writing R Extensions*. You are supposed to bundle together all the R source code, native (Fortran or C) code, documentation, help files, test files, data files and code examples into something called a "zipped tarball," a compressed archive developed for Unix. This tarball is created by Rtools using a command `R CMD build`. You submit the tarball to CRAN. The CRAN managers test your tarball. If it works as it should, they place it in the archive. R users can download the tarball, build the package and install it regardless of what platform they are using. On a Windows computer, the package would have to be built with Rtools, which uses the same MinGW compilers that are used to build R itself.

The Free Software Foundation, which creates and distributes MinGW, works at its own pace and has been slow to implement versions of Fortran beyond Fortran 77. They have been making progress, and they now have a compiler called `gfortran` which is almost fully compliant with the Fortran 95 standard. This compiler seems to work well on Unix-like systems, but a thoroughly tested Windows version capable of producing DLL's has not yet arrived. Until it does, Fortran source code in user-contributed R packages needs to be written in the style of Fortran 77.

2.4 One solution: distributing an R package in binary form

The fact that the MinGW compilers for Windows cannot fully handle Fortran 95 is irrelevant to the vast majority of Windows R users, because very few of them actually need those compilers anyway. Most of these users download and install the precompiled, binary version of R for Windows. And if they need a package, they will download the precompiled binary version of the package from from CRAN and install it as described in Section 1.2.

If you want to create an R package that calls Fortran 95, you can compile and link the Fortran source using Intel Fortran or any other commercial compiler capable of producing

a DLL. You can then use Rtools to build a precompiled, binary version of your package for Windows that includes the DLL but not the Fortran source. The procedure for doing so is not described in *Writing R Extensions*. And packages created in this manner will not be accepted by CRAN, because they cannot be tested in the usual manner or rebuilt for non-Windows platforms. But the Windows (`.zip`) archive produced in this manner looks just like any precompiled binary R package for Windows available on CRAN. A user who is given the `.zip` file can install the package from the Windows GUI version of R by selecting "Install package(s) from local zip files..." from the "Packages" menu, just as he would if he had downloaded the `.zip` file from CRAN.

My procedure for doing this, which I describe in the next two sections, was devised by trial and error. It works on my computer (Windows Vista Business, 64-bit, Service Pack 1) with the current version of R (Version 2.7.0) and the most recent version of Rtools (`Rtools28.exe`). I see no reason why this procedure should not work with Windows XP or with future versions of R and Rtools. The procedure may actually be easier with Windows XP, because you won't have to work around the annoying security restrictions imposed by Vista.

3 Preparing your Fortran 95 and R source code

3.1 Motivating example: NORM

A nontrivial example of an R package is Version 3 of Norm. Norm allows a user to analyze multivariate data with missing values, and its name is derived from the model (multivariate normal) that underlies its procedures. The package contains an EM algorithm for parameter estimation, an MCMC (data augmentation) procedure for posterior simulation, routines for predicting and imputing missing values, and a procedure for combining the results from analyses after multiple imputation using Rubin's (1987) rules for scalar estimands. Detailed information about this package is described in the accompanying user's manual.

3.2 Fortran source

Virtually all of the computations in Norm are carried out in Fortran 95, using the programming style described by Lemmon and Schafer (2005). The Fortran routines are encapsulated in the dynamic-link library `norm.dll`. The source code for producing `norm.dll` is found in these files:

```
error_handler.f90
```

```
constants.f90
dynalloc.f90
matrix.f90
quick_sort.f90
randgen.f90
tabulate.f90
norm_engine.f90
norm.f90
```

Of these nine files, the first seven are standard code libraries used in many other applications developed at The Methodology Center. The file `norm_engine.f90` defines a module of computational routines for EM, MCMC, etc. The functions declared to be `public` in the `norm_engine` module perform extensive argument checking and error messaging; they were designed to be efficient, robust and virtually crash-proof, and we made them easy to call from other Fortran code by following the principles in Lemmon and Schafer's (2005) Chapter 4. The last file, `norm.f90`, contains the wrapper functions for the procedures exported by the DLL, written in the style described in Chapter 6.

All nine Fortran source files conform to the Fortran 95 standard. They do not use any extensions or features specific to the compiler we are currently using (Intel), except for the directive comment lines in `norm.f90` that define the symbols to be exported in the DLL.

3.3 R source

Users of R do not want to call Fortran directly. The Norm package has a collection of R functions that accept data in the form of standard R data objects (matrices, data frames, etc.), convert the data into forms recognizable by Fortran, send the data to Fortran, retrieve the results from Fortran, and return the results to the R user in the form of R objects.

The R source code that defines these functions is found in two files:

```
norm.R
miInference.R
```

These files are designed to be read into R using `source`, as in `source("norm.R")` and `source("miInference.R")`. Under ordinary circumstances, R code for a package would placed in a single file named *package-name*`.R`. In this case, a couple of R functions have been placed in `miInference.f90`, because these functions are not directly tied to Norm, but are useful for any application involving multiple imputation (MI). In the future, we may use the functions in `miInference.R` in other packages. For example, we might want to create another package of tools for handling multiply imputed data, regardless of what models were

used to create the imputations, and then rebuild Norm to require that package. But for now, we have simply placed the general MI-related functions in `miInference.R` and kept everything else in `norm.R`.

When writing R source code for a package, you should avoid the common practice of using the equal sign `"="` as an alternative the assignment operator `"<-"`. In an ordinary R session, the two commands

```
> x <- 5
> x = 5
```

will do the same thing, but the first way is preferred.

You should also avoid using `"T"` and `"F"` as abbreviations for `"TRUE"` and `"FALSE"`. In R, `"TRUE"` and `"FALSE"` are literal logical symbols, whereas `"T"` and `"F"` are system variables whose values have been initialized to `"TRUE"` and `"FALSE"`. If you make a mistake in your code, you may accidentally change the value of `T` or `F`, as in this example:

```
> T <- F
> T
[1] FALSE
```

Finally, when you create your functions, you should avoid the common practice of placing the name of the object to be returned by itself in last line of your function code, and use the formal syntax of `return`. Instead of doing this,

```
myFunc <- function( arg1, arg2, ... ){
    # do something
    result <- list( comp1, comp2, ... )
    result }
```

you should do this:

```
myFunc <- function (arg1, arg2, ... ){
    # do something
    result <- list( comp1, comp2, ... )
    return(result) }
```

3.4 Classes and generic methods

R is an object-oriented language built on classes. Every object in R has an attribute called `class`, which is a character string that tells you what kind of object it is. There is also a function called `class` which allows you to query an object to discover its class.

```
> # define some simple objects
> ivec <- 1:10         # a vector of integers
> rmat <- matrix( rnorm(10), 5, 2 ) # a matrix of random numbers
> converged <- F       # a logical value
> mylist <- list( ivec, rmat, converged) # a list with 4 components
>
> # see their classes
> class(ivec)
[1] "integer"
> class(rmat)
[1] "matrix"
> class(converged)
[1] "logical"
> class(mylist)
[1] "list"
```

Classes are important because they allow us to create generic methods. A generic method in R is a function that does different things depending on what kind of information you supply to it. (Throughout this document, the terms "method" and "function" will be used interchangeably.) The most widely used generic method in R is `print`. The `print` function displays the contents of an object on the screen, but the format in which the information is displayed varies depending on the object's class; vectors, matrices and lists are all displayed differently. Another well known example is `summary`, which summarizes the results of statistical modeling procedure. The function `summary` can be applied to the results from a linear model fit by `lm`, or to the results from a logistic model fit by `glm`, and the results will be different.

A generic function in R is actually a group of functions called by the same name but distinguished by the class of the first argument. Suppose that you fit a linear model using the function `lm`.

```
> myModelFit <- lm( y ~ x1 + x2, data=myDataFrame )
```

The result, which we have called `myModelFit`, is an object of class `"lm"`. It was created as a list, but its class attribute was changed from `"list"` to `"lm"` to let other R functions know exactly what kind of object it is. Now suppose you send this object to the generic function `summary`.

```
> summary( myModelFit )
```

When you do this, the R language perceives that the first (and, in this example, the only) argument is an object of class `"lm"`, so it takes that object and passes it to another function called `summary.lm`. If you look at the R help page for `summary`, this is what you will see.

```
summary                 package:base                 R Documentation

Object Summaries

Description:

     'summary' is a generic function used to produce result summaries
     of the results of various model fitting functions.  The function
     invokes particular 'methods' which depend on the 'class' of the
     first argument.

Usage:

     summary(object, ...)

     ## Default S3 method:
     summary(object, ..., digits = max(3, getOption("digits")-3))
     ## S3 method for class 'data.frame':
     summary(object, maxsum = 7,
            digits = max(3, getOption("digits")-3), ...)

     ## S3 method for class 'factor':
     summary(object, maxsum = 100, ...)

     ## S3 method for class 'matrix':
     summary(object, ...)
```

In the first part of the `Usage` section,

```
summary(object, ...)
```

the ellipsis `"..."` indicates that if you had called `summary` with additional arguments after `myModelFit`, those arguments would also be passed along to `summary.lm`. To see what additional arguments would be appropriate, you can look at the help page for `summary.lm`:

```
summary.lm              package:stats              R Documentation

Summarizing Linear Model Fits

Description:

     'summary' method for class '"lm"'.

Usage:

     ## S3 method for class 'lm':
     summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)
```

```
      ## S3 method for class 'summary.lm':
      print(x, digits = max(3, getOption("digits") - 3),
            symbolic.cor = x$symbolic.cor,
            signif.stars = getOption("show.signif.stars"), ...)

Arguments:

  object: an object of class '"lm"', usually, a result of a call to
          'lm'.

       x: an object of class '"summary.lm"', usually, a result of a
          call to 'summary.lm'.

        < lines omitted >
```

From the first part of the `Usage` section, wee see that the additional arguments `correlation` and `symbolic.cor` might be useful when we are applying `summary` to an object of class `"lm"`. Notice also that there is another part to the `Usage` section:

```
      ## S3 method for class 'summary.lm':
      print(x, digits = max(3, getOption("digits") - 3),
            symbolic.cor = x$symbolic.cor,
            signif.stars = getOption("show.signif.stars"), ...)
```

This part tells us what happens when the result from `summary.lm` is displayed with the generic method `print`, whether you do it by explicitly, like this,

```
> print( summary( myModelFit ) )
```

or implicitly in an interactive session, like this:

```
> summary( myModelFit )   # this implicitly calls "print"
```

The result from the function `summary.lm` is an object of class `"summary.lm"`, so when you send it to the generic method `print`, it immediately gets passed along to another function called `print.summary.lm`. These two special functions, `summary.lm` and `print.summary.lm`, are closely related, and so the R Development Team has chosen to document them together on the same help page. You can call these special functions directly if you like, as in

```
> print.summary.lm( summary.lm( myModelFit ) )
```

but that would get tedious; it's easier to just use the generic names `summary` and `print`.

3.5 Defining your own classes

Defining new classes is simple. You just change the class attribute of an object to something new. If you say

```
> x <- rnorm(10)
> class(x) <- "boogeyman"
```

then you have just created an object of class `"boogeyman"`. If you then create a new function called `summary.boogeyman`, that function will be invoked automatically when you call `summary` with an object of class `"boogeyman"` as its first argument. What happens if there is no function called `summary.boogeyman`? If R cannot find it, then it tries to do something sensible. As a last resort, it sends your object to a function called `summary.default`. Every generic method in R is supposed to have a default method to deal with objects that are not handled by other class-specific functions.

When developing a package, you may want to create new classes, but this should be done thoughtfully and for good reason. In the Norm package, we have defined a new class called `"norm"`. Objects of class `"norm"` are created by the function `emNorm`, which runs the EM algorithm, and also by `mcmcNorm`, which runs a data augmentation procedure. We could have created two different object classes, one for the result from EM and another for the result from MCMC, but then we would have had to write twice as many functions to deal with them. In the file `norm.R`, the last part of the R code that defines the function `emNorm` looks something like this:

```
result <- list( y = y, x = x, method = "EM",
   # lines here have been omitted
   msg = msg)
# set class and return
class(result) <- "norm"
return(result)}
```

The result from a call to `emNorm` is actually a list, but just before we return the result, we change its class attribute to `"norm"`. Notice that one component of the object is named `method`, which in this case is set to `"EM"`. The last part of the R code that defines the function `mcmcNorm` looks similar to this, except that the component `method` is set to `"MCMC"`. We did this so that our function `summary.norm`, which is also defined in `norm.R`, will be able to distinguish between `"norm"` objects created by the two functions. Here is an example of how a user would invoke these functions to run EM and summarize the results.

```
> emResult <- emNorm(y)  # y is a data matrix containing NA's
> summary( emResult, show.var=F, show.patt=F, show.param=F )
```

```
Method:                          EM
Prior:                           "uniform"
Convergence criterion:           1e-05
Iterations:                      17
Converged:                       TRUE
Max. rel. difference:            6.00213e-06
-2 Loglikelihood:                7310.107
-2 Log-posterior density:        7310.107
Estimated rate of convergence: 0.46362
```

3.6 Creating your own generics and methods

When you create new object classes, some of the functions that you will write to operate on these classes will be extensions to generic methods that already exist (e.g., `summary` and `print.summary`). You may also want to create your own generics. Once again, this should be done thoughtfully and for good reason.

In the Norm package, we made the functions `emNorm` and `mcmcNorm` generic, because we wanted to allow users to call them by two different techniques. In the first technique, the user provides a data matrix or data frame containing missing values, so that the EM or MCMC algorithm will be applied to the raw data. Through additional optional arguments, the user could specify starting values, prior distributions, and other options that determine how long the algorithms should run and when they should stop. In the second technique, the user calls these functions with the first argument being an object of class `"norm"` resulting from a previous call to `emNorm` or `mcmcNorm`. In a typical analysis, the user will probably want to run EM first, and then use the parameter estimates from EM as starting values for MCMC. If you take the result from `emNorm` and use it as an argument to `mcmcNorm`, the data, the estimated parameters, and the specification of the prior distribution (all of which are stored in the `"norm"` object) are carried over automatically. Because of the iterative nature of EM and MCMC, there are situations where it makes sense to use a result from either function as an argument to either function. Here is an example of how a user might take a result from `emNorm` and use it as an argument in another call to `emNorm`.

```
> # Run EM on a data matrix containing NA's
> emResult <- emNorm(y)
Warning message:
In emNorm.default(y) : Algorithm did not converge by iteration 1000
>
> # Run EM some more
> emResult <- emNorm( emResult )
Warning message:
In emNorm.default(y = obj$y, x = obj$x, intercept = FALSE, iter.max = iter.max,  :
  Algorithm did not converge by iteration 1000
```

Remember: The rationale for creating new classes and generic methods is to make your functions easy to remember and easy to use. The average user should not need to delve into the fine details of what is contained inside an object, or what happens inside your functions.

Creating a generic method takes only a couple of lines of code. The generic functions `emNorm` is defined this way.

```
emNorm <- function(obj, ...){
   # S3 generic function
   UseMethod("emNorm")}
```

The argument `"..."` is a placeholder for additional unspecified arguments to be passed along. After defining the generic method, we defined two specific methods to be called by it. One is called `emNorm.default`; it accepts raw data in the form of a data frame, data matrix or vector and applies the EM algorithm.

```
emNorm.default <- function(obj, x=NULL, intercept=TRUE,
   iter.max=1000, criterion=NULL, prior="uniform", prior.df=NULL,
   prior.sscp=NULL, starting.values=NULL, ...){
      # many lines of code omitted #

   # call Fortran
   tmp <- .Fortran("norm_em",
      n = nrow(y),
      r = ncol(y),
      p = ncol(x),
      x = x,

      # many lines of code omitted #

return(result)}
```

The other specific method is called `emNorm.norm`; it an object of class `"norm"`, extracts the raw data and other information from it, and passes it along to `"emNorm.default"`. All the hard work of checking the user-supplied arguments, preparing the data to send to Fortran, calling Fortran, etc. is done within `"emNorm.default"`, and `"emNorm.norm"` is just a few lines of R code that calls `emNorm.default`.

```
emNorm.norm <- function( obj, iter.max = 1000,
   criterion = obj$criterion,
   prior = obj$prior, prior.df = obj$prior.df,
   prior.sscp = obj$prior.sscp,
   starting.values = obj$param, ...){
   ###############################################
   # S3 method for class "norm"
   ###############################################
   result <- emNorm.default( obj$y, x = obj$x,
```

```
    intercept = FALSE, iter.max = iter.max, criterion = criterion,
    starting.values = starting.values,
    prior = prior, prior.df = prior.df, prior.sscp = prior.sscp)
  return(result)}
```

We adopted this strategy to avoid unnecessary duplication of code, which makes the package easier to debug and maintain. If you find yourself writing lots of specific methods with large sections of code being duplicated, then it would be wise to stop and re-think your programming strategy.

When you create specific methods associated with a generic method, there are some rules that you need to follow.

- The first dummy argument to each specific method must have the same name as the first argument to the generic. In the case of `emNorm`, the dummy argument is named `"obj"`.
- Every other dummy argument in the generic method must also appear in every specific method, with the same name. In particular, the last argument to the generic method, `"..."`, should also appear as the last argument to every specific method.

3.7 What should I do in R, and what should I do in Fortran?

The community of R developers and sophisticated R users will probably tell you that you should write as much of your code as possible in R, and reserve only the most computationally intensive parts for Fortran. That's fine if your only goal is to create an R package. But if the procedures that you develop are truly useful, you may eventually want to disseminate them to researchers who do not use R.

In the long run, we believe that it's a good idea to do as much of the input checking, data manipulation and computation as possible within your Fortran code, not within R. Good Fortran code has a long shelf life and can be used in many different environments. A Fortran DLL that you create for R can also, with little or no revision, be called from Fortran, SAS, MATLAB, C, and Visual Basic. Doing as much as you can within Fortran helps to maximize the usefulness and portability of your routines.

Within your R code, you will need to do some checking and manipulation of inputs that users supply to your functions. But we believe that those checks and manipulations should be minimal. Argument checking and manipulation within R should be limited to issues of data type, dimensions, and so on, just to make sure that they can be accepted by Fortran, so that R does not crash when you call your DLL.

In the Norm package, for example, a user may supply starting values for parameters to the EM algorithm. The function `emNorm` makes sure that those starting values have the correct size and shape and then passes them to Fortran. More sophisticated tests, such as making sure that the user-supplied covariance matrix is symmetric and positive definite, are performed within Fortran. If a test fails, Fortran supplies an informative error message to R as a character string of length 255, and R prints out that message for the user.

4 Getting ready to build your package

4.1 Status of the project

At this point, we assume that you have successfully created a set of R functions and Fortran subroutines that work well together. We assume that your R functions are defined in a file called *package-name*`.R` (although you may have multiple files of R code). We also assume that your Fortran subroutines have been encapsulated in a dynamic-link library called *package-name*`.dll`.

4.2 Checking for library dependencies

Depending on what Fortran compiler you are using, the routines that you have encapsulated in *package-name*`.dll` may require some compiler-specific runtime libraries. Applications developed with older versions of Intel Fortran often depended on procedures contained in a file called `dforrt.dll`. Applications developed with Salford Fortran often depended on `salflibc.dll`. Before proceeding, it's a good idea to learn if such dependencies exist. One easy way to do this is to copy *package-name*`.dll` onto another Windows machine that has a recent version of R but does not have your compiler or Microsoft Visual Studio. Start an R session from inside the directory where you placed you DLL, and then issue this command:

```
> dyn.load("package-name.dll")
```

If your DLL depends on another DLL that is not present, R will print an error message with the name of the missing DLL. If that happens, you need to go back to the computer that you have been using for development, find the missing DLL, copy it to the new computer into the same directory as *package-name*`.dll`, and try the dynamic loading again. If your DLL depends on another DLL, you will need to include a copy of that other DLL when you build and distribute your package.

4.3 Creating a startup banner

Some packages print a message when the user loads them. With Norm, this is what you see.

```
> library(norm)
NORM library for R, Version 3.0.0
Copyright (C) 2008 Joseph L. Schafer
The Methodology Center, Penn State University
This software is distributed without warranty;
see LICENSE file for details.
```

A startup banner lends a professional touch and lets the user know that the package loaded correctly. A startup banner may be created within an optional function called `.onAttach`, which you define in your file *package-name*`.R`, like this.

```
.onAttach <- function(lib, pkg)
  cat( c( "NORM library for R, Version 3.0.0",
          "Copyright (C) 2008 Joseph L. Schafer",
          "The Methodology Center, Penn State University",
          "This software is distributed without warranty;",
          "see LICENSE file for details."), sep="\n")
```

4.4 Writing a user's manual

When you create an R package, you provide documentation (help) files that give the details of each function. Documentation files are useful to someone who already knows what the package does and has some idea of how to use it. You ought to consider writing another document that gives an overview of the package for new users and shows them how to get started with real data examples. This document can be in any form, but we recommend making it a `.pdf` file. The document will eventually be placed in a directory (folder) called `doc`, a subdirectory of the directory where your package is installed.

4.5 Including data examples

Package developers often distribute small datasets with their packages to test their functions and to use as examples in their documentation. This practice is highly recommended. The Norm package includes several small datasets. Once the package has been loaded, the user can see what datasets are included with Norm by typing:

```
> data(package="norm")
```

One of the datasets is called `marijuana`. If the user types

```
> data(marijuana)
```

then a copy of this dataset is loaded into the user's workspace as a data frame.

```
> marijuana
  Plac.15 Low.15 High.15 Plac.90 Low.90 High.90
1      16     20      16      20     -6      -4
2      12     24      12      -6      4      -8
3       8      8      26      -4      4       8
4      20      8      NA      NA     20      -4
5       8      4      -8      NA     22      -8
6      10     20      28     -20     -4      -4
7       4     28      24      12      8      18
8      -8     20      24      -3      8     -24
9      NA     20      24       8     12      NA
```

There are several different ways to include datasets with your package. We recommend arranging each one as an ordinary ASCII (text) file, calling *dataset-name*`.txt`, capable of being read into R using `read.table` with `header=T`. The file `marijuana.txt` looks like this.

```
Plac.15 Low.15 High.15 Plac.90 Low.90 High.90
 16  20  16  20  -6  -4
 12  24  12  -6   4  -8
  8   8  26  -4   4   8
 20   8  NA  NA  20  -4
  8   4  -8  NA  22  -8
 10  20  28 -20  -4  -4
  4  28  24  12   8  18
 -8  20  24  -3   8 -24
 NA  20  24   8  12  NA
```

Data files will be placed in a subdirectory called `data`.

4.6 Creating the package directory

Now you are ready to begin setting up the directories and files that Rtools will need to build your package. First, create a new directory with the same name as your package, all in lowercase, e.g. `norm`. In the discussion to follow, we will refer to this as the package directory.

Next, in the package directory, create three new, empty, plain text files with the following names (all uppercase):

```
DESCRIPTION
LICENSE
NAMESPACE
```

If you create these files by right-clicking the mouse within Windows Explorer, Windows will want to add the extension `.txt` to the file names. As you edit these files, you may want to keep the `.txt` extension so that they automatically open with your preferred text editor. However, you will need to remove `.txt` from the filenames before you check and build the package.

Next, in the package directory, create subdirectories with these names (all lowercase).

```
data
doc
libs
man
R
```

Next, start to populate these directories in the following way.

- Copy the data (*dataset-name*`.txt`) files for any datasets that you want to include with your package into `data`. If you have no datasets, you may delete `data`.
- Copy your user's manual, if you have one, into `doc`. If you don't have a user's manual, then you may delete `doc`.
- Copy your library *package-name*`.dll`, and any other DLL's that it may depend on, into `libs`.
- The directory `man` will hold the R documentation (`.Rd`) files for your functions and datasets, which we will describe later.
- Copy *package-name*`.R` and any other R source code files into `R`.

4.7 The DESCRIPTION file

This file is required for any package, and it needs to follow a specific format. The best way to create it is to follow someone's example. This is the `DESCRIPTION` file for the Norm package.

```
Package: norm
Type: Package
Title: Analysis of incomplete multivariate data under a normal model
Version: 3.0.0
Date: 2008-07-18
Author: Joseph L. Schafer <jls@stat.psu.edu>
Maintainer: Joseph L. Schafer <jls@stat.psu.edu>
Description: Functions for estimation and multiple imputation from
        incomplete multivariate data under a normal model
License: See LICENSE file
LazyLoad: yes
```

Later, when you build the package more information will be added to this file automatically.

4.8 The LICENSE file

This file is not required, but it is good to have one to protect your intellectual property rights. You can put whatever you want into this file. Here is the `LICENSE` file for Norm.

```
Citation
========

Schafer, J.L. (2008) NORM: Analysis of incomplete multivariate
data under a normal model, Version 3. Software package for R.
University Park, PA: The Methodology Center, The Pennsylvania
State University.


Copyright
=========

File norm/data/flas.txt contains data from:
     Schafer, J.L. (1997) Analysis of Incomplete Multivariate
     Data. London: Chapman & Hall/CRC Press.

File norm/data/cholesterol.txt contains data from:
     Ryan, B.F. and Joiner, B.L. (1994) Minitab Handbook
     (Third edition). Belmont, CA: Wadsworth.

File norm/data/marijuana.txt contains data from:
     Weil, A.T., Zinberg, N.E. and Nelson, J.M. (1968) Clinical
     and psychological effects of marihuana in man. Science,
     62, 1234-1242.

All other files: Copyright (C) 2008 by Joseph L. Schafer
```

```
License
=======

This software is provided in good faith to researchers free of charge.
It is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  It may be used by anyone if credit
is given.


Acknowledgement
===============

This work was supported by National Institute on Drug Abuse,
1-P50-DA-10075.
```

4.9 The NAMESPACE file

In the `NAMESPACE` file, you instruct R to load your Fortran DLL when the package is loaded. You also declare which functions in your source code will be accessible to package users and which ones will be kept hidden from them. (Functions may be kept hidden if they are meant only for internal use, i.e. they are only called by your own functions.) Making a function accessible to package users is called exporting, and every function that you export will need to be documented. Finally, the `NAMESPACE` file, you also declare the class-specific and default methods to be associated with each generic method. Here is the `NAMESPACE` file for the Norm package.

```
useDynLib( norm, .registration=TRUE )

export( emNorm )
export( emNorm.default )
export( emNorm.norm )

export( impNorm )
export( impNorm.default )
export( impNorm.norm )

export( loglikNorm )
export( loglikNorm.default )
export( loglikNorm.norm )

export( logpostNorm )
export( logpostNorm.default )
export( logpostNorm.norm )

export( mcmcNorm )
```

```
export( mcmcNorm.default )
export( mcmcNorm.norm )

export( miInference )
export( print.miInference )

export( summary.norm )
export( print.summary.norm )

S3method( emNorm, default )
S3method( emNorm, norm )
S3method( impNorm, default )
S3method( impNorm, norm )
S3method( loglikNorm, default )
S3method( loglikNorm, norm )
S3method( logpostNorm, default )
S3method( logpostNorm, norm )
S3method( mcmcNorm, default )
S3method( mcmcNorm, norm )
S3method( print, norm )
S3method( print, summary.norm )
S3method( summary, norm )
S3method( print, miInference )
```

4.10 Creating the R documentation files

The last major step in creating the package is to write help files. This step is tedious but necessary. R help files are written in a special markup language that resembles LaTeX. The source code for each page is kept in a text file with the extension `.Rd`. When the package is built, Rtools will convert the `.Rd` files into a variety of formats, including plain text, HTML (`.htm`), Windows compiled HTML for help (`.chm`) and LaTeX (`.tex`). The `.Rd` format is easy to learn, especially if you are familiar with LaTeX. It is described very well in the document *Writing R Extensions* by the R Development Core Team, and you will need to refer to that manual.

Each of the functions that you are exporting needs to be documented, but that doesn't mean that each one needs to have its own `.Rd` file. If several functions are closely related, you may group them together into a single help page. In particular, a generic method and the specific methods associated with it are usually documented on the same page. The file `emNorm.Rd` serves as documentation for `emNorm`, `emNorm.default` and `emNorm.norm`. The file begins like this.

```
\name{emNorm}

\alias{emNorm}
```

```
\alias{emNorm.default}
\alias{emNorm.norm}

\title{ EM algorithm for incomplete multivariate normal data}

\description{
Computes maximum likelihood estimates and posterior modes from
incomplete multivariate data under a normal model.
}

\usage{
% the generic function
emNorm(obj, \dots)

% the default method
\method{emNorm}{default}(obj, x = NULL, intercept = TRUE,
   iter.max = 1000, criterion = NULL, prior = "uniform",
   prior.df = NULL, prior.sscp = NULL, starting.values = NULL,
   \ldots)

% method for class norm
\method{emNorm}{norm}(obj, iter.max = 1000,
   criterion = obj$criterion, prior = obj$prior, prior.df = obj$prior.df,
   prior.sscp = obj$prior.sscp, starting.values = obj$param, \ldots)

}

\arguments{

  \item{obj}{an object used to select a method.  It may be
  a numeric matrix, vector or data frame of responses
  and \code{NA}'s to be  modeled as normal. If it
  is a data frame, any factors or ordered factors will be
  replaced by their internal codes, and a warning will be given.
  Alternatively, it may be an object of class \code{"norm"}
  resulting from a call to \code{emNorm} or
  \code{\link{mcmcNorm}}; see DETAILS.}

       < lines omitted >
```

The argument to `\name{}` is the name of the file, but without the `.Rd` extension. The three `\alias{}` commands tells R to display this page when a ueer asks for help on `emNorm`, `emNorm.default` or `emNorm.norm`. In the `\arguments{}` section, the arguments must exactly match those of the function as defined in your source code with respect to the argument names, their order and their default values. Rtools has a command-line utility for checking the package, `R CMD check`, which we will discuss later; it will check the `.R` and `.Rd` files for consistency.

Near the end of each `.Rd` file that documents functions, you are expected to provide examples

of how each function is used. These examples may involve datasets distributed with your package, and the code examples should actually work. Under ordinary circumstances, the `R CMD check` utility will run all of these examples and make sure that they work. But because we are including code written in Fortran 95, `R CMD check` not be able to check the code examples. It will be your responsibility to make sure that the examples work.

In addition to documenting each exported function, you should document each dataset included in the package, and you should document the package itself. Each dataset should have its own file named *dataset-name*`.Rd` and the package should have a file called *package-name*`-package.Rd`.

As you create the `.Rd` files, you should frequently preview them to see how they will look. Rtools has command-line utilities for converting these files into different formats. To convert `emNorm.Rd` into HTML, open a command prompt window, go to the `man` directory, and type

```
R CMD Rdconv --type=html --output="" emNorm.Rd
```

which will create the file `emNorm.html`. The command

```
R CMD Rd2dvi emNorm.Rd
```

will convert it into LaTeX and then compile it to create `emNorm.dvi`. If your `.Rd` file contains syntax errors, these conversion routines will display not-so-helpful error messages. Errors within a section of the `.Rd` file may cause a whole section of the page (e.g., the Usage section) to simply disappear. So when you preview your help pages, make sure that each section is present. After previewing your files, you should delete the converted versions (`.dvi`, `.html`, etc.) from the `man` directory and keep only the `.Rd` files.

5 Checking and building the package

5.1 Checking it

If you have completed all the steps thus far, your package is ready to check and build. Checking the package is very important. The package-checking utility in Rtools does an excellent job of finding problems in your code and documentation.

The package-checking utility was also designed to run the code examples in each of your `.Rd` files. Unfortunately, it does not know how to do this when your package contains native routines written in Fortran 95, when you are distributing only the DLL and not the

Fortran source. Before checking your package, you will need to comment out the `useDynLib` command in the `NAMESPACE` file, like this.

```
#useDynLib( norm, .registration=TRUE )
```

To check the package, open a command prompt window and go to the directory that contains your package directory (one directory level above it). Then type this command:

```
> R CMD check --no-examples --no-tests package-name
```

Most likely, problems will be found in your `.R` and `.Rd` files which will need to be corrected. Make corrections until `R CMD check` issues no more warnings.

Note to users of Windows Vista. You may experience problems running `R CMD check` because the operating system may prevent certain directories and files from being created. The only solution to this problem that I have found is to uninstall R and reinstall it in your own personal disk space. For example, if your username is `jls`, your personal disk space is `C:\Users\jls`. Install R in a subdirectory of this, and move your package directory structure to another subdirectory of this. Then run `R CMD check` from one directory level above your package directory.

5.2 Building it

Now you are ready to build your package. Remove the comment symbol `#` from the `useDynLib` command in `NAMESPACE`. Then go to the command prompt and type this command:

```
> R CMD build --binary --use-zip package-name
```

This will create an archive named after your package, with the version number and a `.zip` extension. If you look inside this archive with Windows Explorer, you will see the basic package directory structure with subdirectories added to it. For example, there is now a subdirectory called `chtml` which contains your help files in Windows compiled HTML format.

But two crucial subdirectories will be missing from the `.zip` archive. One is `libs`, which contains your DLL file(s). Another is `doc`, which contains your user's manual if you have one. For the package to work properly, you will need to copy these two subdirectories and their contents into the `.zip` archive. Do this with Windows Explorer in the usual way.

5.3 Install it and test it

Try installing the package from the `.zip` file as described in Section 1.2. You should now be able to use the package functions and view the help files.

By default, the package is placed in a subdirectory of `library` where R has been installed. Look for your package there. One of the subdirectories of your package is called `R-ex`. It contains a set of R scripts (`.R` files) corresponding to the code examples in your `.Rd` files. Try running all of these examples to make sure that they work properly.

Now go forth and distribute your package.

6 References

Lemmon, D.R. and Schafer, J.L. (2005) *Developing Statistical Software in Fortrsn 95*. New York: Springer.