Calling C functions from R using .C and .Call

Søren Højsgaard

April 14, 2008

This note describes how to call C functions from R. A more thorough description is provided in the "Writing R extensions"–manual. It is all based on working on a Windows platform. If you find errors in the description, please do report.

1 The compiler

The MinGW compiler is required and can be obtained from `http://www.mingw.org/`. However, it is probably more convenient to install `Rtools` (which includes the compiler) from `http://www.murdoch-sutherland.com/Rtools/`. (See the file `Rtools.txt` for installation details. It is quite easy!).

Note: The compiler must be on your computers path. If you are using `cygwin` there is a potential source of conflict, because R does not seem to work well with the `cygwin` compiler. In that case, the path to the MinGW compiler must appear before that to `cygwin` in your search path. All of this will be taken care of if you use `Rtools`.

2 .C and .Call

There are two ways of calling C functions from R: Using .C() or using .Call(). The .C() function is the basic way of doing it whereas .Call() is a little more difficult but offers several advantages: With .C() all arguments must be "primitive" (vectors of doubles or integers) whereas .Call() accepts R data structures such as matrices and lists. Likewise, .C() returns only primitive values whereas .Call() returns R data structures. In fact, .C() does not return anything at all, so any result from the C program must go into one of the arguments of the C function. There are certain additional advantages of using .Call() which will not be discussed here. The specific form of the C functions to be called will depend on whether .C() or .Call() is being used. I am not aware of any difference in performance of the two approaches. Hence it is largely a matter of taste which approach to choose. However, C code which can be called using .C() is more portable (outside of the R system) so that might be one issue to consider.

To be specific about the differences: If we want to call a C function using .C() with a matrix argument, then the matrix must be turned into and supplied as a numeric (or integer) vector along with the number of rows and columns. On the other hand, if using .Call() the matrix can be supplied as it is, but then there is some extra work to be done in the C program.

We illustrate the differences by considering matrix multiplication:

```
> A <- matrix(1:9, nr = 3)
> B <- matrix(1:6, nr = 3)
> A %*% B
```

```
     [,1] [,2]
[1,]   30   66
[2,]   36   81
[3,]   42   96
```

2.1 Using .C()

A C function for matrix multiplication to be called with .C() is:

```c
/* File: matprod1.c */

void matprod1(double *x, int *nrx, int *ncx,
              double *y, int *nry, int *ncy, double *ans)
{
  double sum;
  int ii, jj, kk;

  for (ii=0; ii<*nrx; ii++){
    for (jj=0; jj<*ncy; jj++){
      sum = 0;
      for (kk=0; kk<*ncx; kk++){
        sum = sum + x[ii+*nrx*kk]*y[kk+*nry*jj];
      }
      ans[ii+*nrx*jj] = sum;
    }
  }

}
```

To compile the C function, issue the following command in a command window:

```
Rcmd SHLIB matprod1.c
```

This creates a .dll file on Windows (and a .so file on Unix/Linux as far as I understand it). (If you have more .c files these are just listed and the output will still be one .dll file).

The function is called with:

```
> dyn.load("matprod1.dll")
> ans <- .C("matprod1", as.numeric(A), nrow(A), ncol(A), as.numeric(B),
     nrow(B), ncol(B), ans = numeric(nrow(A) * ncol(B)))$ans
> dyn.unload("matprod1.dll")
> dim(ans) <- c(nrow(A), ncol(B))
> ans
```

```
     [,1] [,2]
[1,]   30   66
[2,]   36   81
[3,]   42   96
```

Note that the arguments are explicitely converted to have the right types. This may not be necessary, but it is safe to do so.

2.2 Using .Call()

A C function for matrix multiplication to be called with .Call() is:

```
/* File: matrprod2.c */

#include <Rinternals.h>
SEXP matprod2(SEXP x, SEXP y)
{
  int nrx, ncx, nry, ncy;
  int *xdims, *ydims;
  double *ansptr, *xptr, *yptr;
  SEXP ans;

  xdims = INTEGER(coerceVector(getAttrib(x, R_DimSymbol), INTSXP));
  ydims = INTEGER(coerceVector(getAttrib(y, R_DimSymbol), INTSXP));
  nrx = xdims[0];     ncx = xdims[1];
  nry = ydims[0];     ncy = ydims[1];

  PROTECT(x = coerceVector(x, REALSXP));
  PROTECT(y = coerceVector(y, REALSXP));
  xptr = REAL(x);
  yptr = REAL(y);

  PROTECT(ans = allocMatrix(REALSXP, nrx, ncy));
  ansptr = REAL(ans);

  double sum;
  int ii, jj, kk;

  for (ii=0; ii<nrx; ii++){
    for (jj=0; jj<ncy; jj++){
      sum = 0;
      for (kk=0; kk<ncx; kk++){
        sum = sum + xptr[ii+nrx*kk]*yptr[kk+nry*jj];
      }
      ansptr[ii+nrx*jj] = sum;
    }
  }

  UNPROTECT(3);
  return(ans);
}
```

The function is called with:

```
> dyn.load("matprod2.dll")
> ans <- .Call("matprod2", A, B)
> dyn.unload("matprod2.dll")
> ans
```

```
     [,1] [,2]
[1,]   30   66
[2,]   36   81
[3,]   42   96
```

3 Using linear algebra routines

There are various libraries with linear algebra routines available on the internet, e.g. lapack (`http://www.netlib.org/lapack/`) or linpack (`http://www.netlib.org/linpack/`). These libraries use a library called blas. These routines are implemented in Fortran. Some of the routines in these are available in R.

The functions above in a version using `lapack` are given in the file `matprod34.c`:

```
/* File: matprod4.c */

#include <Rinternals.h>
#include <R_ext/Applic.h> /* for dgemm */

void matprod3(double *x, int *nrx, int *ncx,
              double *y, int *nry, int *ncy, double *ans)
{
  char *transa = "N", *transb = "N";
  double one = 1.0, zero = 0.0;
  F77_CALL(dgemm)(transa, transb, nrx, ncy, ncx, &one,
                  x, nrx, y, nry, &zero, ans, nrx);
}


#define getDims(A) INTEGER(coerceVector(getAttrib(A, R_DimSymbol), INTSXP))

SEXP matprod4(SEXP x, SEXP y)
{
  int nrx, ncx, nry, ncy;
  int *xdims, *ydims;
  double *ansptr, *xptr, *yptr;
  SEXP ans;

  xdims = getDims(x);   ydims = getDims(y);
  nrx = xdims[0];       ncx = xdims[1];
  nry = ydims[0];       ncy = ydims[1];

  PROTECT(x = coerceVector(x, REALSXP));
  PROTECT(y = coerceVector(y, REALSXP));
  xptr = REAL(x);   yptr = REAL(y);

  PROTECT(ans = allocMatrix(REALSXP, nrx, ncy));
  ansptr = REAL(ans);

  char *transa = "N", *transb = "N";
  double one = 1.0, zero = 0.0;
  F77_CALL(dgemm)(transa, transb, &nrx, &ncy, &ncx, &one,
                  xptr, &nrx, yptr, &nry, &zero, ansptr, &nrx);
  UNPROTECT(3);
  return(ans);
}
```

To be able to compile this file, first create a file named `Makevars` with the contents

```
PKG_LIBS=$(BLAS_LIBS)
```

and then compile as described above.

The calls are as before

```
> dyn.load("matprod34.dll")
> ans <- .C("matprod3", as.numeric(A), nrow(A), ncol(A), as.numeric(B),
     nrow(B), ncol(B), ans = numeric(nrow(A) * ncol(B)))$ans
> dyn.unload("matprod34.dll")
> dim(ans) <- c(nrow(A), ncol(B))
> dyn.load("matprod34.dll")
```

```
> ans <- .Call("matprod4", A, B)
> dyn.unload("matprod34.dll")
> ans

     [,1] [,2]
[1,]   30   66
[2,]   36   81
[3,]   42   96
```

4 Figuring out how R functions use compiled code

To figure out how matrix multiplication `%*%` is defined in R, do

```
> get("%*%")
.Primitive("%*%")
```

which tells that it is a primitive (i.e., implemented in C). Now look into the source code of R: The file `<R_HOME>/src/main/names.c` directs to the function `do_matprod` in the file `<R_HOME>/src/main/array.c`. From this file one might get inspiration to writing similar functions.