Interfacing C code from R

Søren Højsgaard
Aalborg University, Denmark

November 9, 2012

Contents

1 Introduction

In the following we describe how to call C or C++ functions from R.

- Calling `C` (or `C++`) functions from `R` is usually done to gain speed.
- Knowledge of `C` / `C++` is not a prerequisite for the course. We will show some examples for inspiration and refer you to studying the languages on your own.
- A typical workflow is to first make a pure `R` implementation of your method; then identify possible bottlenecks wrt. computing time. These would then be a candidate for a `C` implementation.
- However, some of the `C` and `Fortran` code underlying `R` is very efficient so there is not necessarily much (if anything) to be gained from a `C` implementation
- The exposition is based on working on a Windows platform.

2 Preparation

For Linux users:

- Things seem to be working out of the box (on the most recent Ubuntu)

For Windows users:

- Download the `Rtools` from a CRAN mirror near you.
- Install `Rtools` and allow for the suggested change of the computers PATH variable.
- It is a good idea to read the file `Rtools.txt` in the `Rtools` installation directory; especially if you have `cygwin` installed on your computer.

For Mac users:

- Unknown – sorry

3 Programming in `C`

3.1 Example: The exponential function

Recall our implementation of the exponential function in `R`:

```
R> expf <- function(x, eps=.Machine$double.eps){
+    if (x<0) {
+      1/expf(-x)
+    } else {
+      n <- 0; term <- 1; ans <- 1
+      while(abs(term)> eps) {
+        n = n + 1
+        term = term * x / n
+        ans <- ans + term
+      }
+      ans
+    }
+  }
```

Various C–implementations are shown below.

```c
/* Compile with gcc -o expfun1 expfun1.c */
#include <stdio.h>
#include <math.h>

int main (void){
  char str[8];
  double x, ans=1.0;
  printf("---␣expfun1␣---\n");
  printf("enter␣x:␣");
  fgets(str, sizeof(str), stdin);
  sscanf(str, "%lf", &x);

  double term=1.0, eps=1e-16;
  int n=0;
  while (fabs(term)>eps){
    n++;
    term = term * x / n;
    ans = ans + term;
  }

  double expx = exp(x);
  double rerr = (ans-expx)/expx;
  printf("input=%f␣ans=%f␣expx=%f␣rerr=%f\n", x, ans, expx, rerr);
}
```

```c
/* Compile with gcc -o expfun2 expfun2.c */
#include <stdio.h>
#include <math.h>

double C_expfun2 (double x){
  double ans=1.0, term=1.0, eps=1e-16;
  int n=0;
  while (fabs(term)>eps){
    n++;
    term = term * x / n;
    ans = ans + term;
  }
  return(ans);
}

int main (void){
  char str[8];
  double x, ans;
  printf("---␣expfun2␣---\n");
  printf("enter␣x:␣");
  fgets(str, sizeof(str), stdin);
  sscanf(str, "%lf", &x);

```

```
  if (x<0)
    ans = 1/C_expfun2(-x);
  else
    ans = C_expfun2(x);

  double expx = exp(x);
  double rerr = (ans-expx)/expx;
  printf("input=%f␣ans=%f␣expx=%f␣rerr=%f\n", x, ans, expx, rerr);
}
```

```
/* Compile with gcc -o expfun3 expfun3.c */
#include <stdio.h>
#include <math.h>

void C_expfun3 (double x, double *ans){
  double term=1.0, eps=1e-16;
  int n=0;
  while (fabs(term)>eps){
    n++;
    term = term * x / n;
    *ans = *ans + term;
  }
}

int main (void){
  char str[8];
  double x, ans=1;
  printf("---␣expfun2␣---\n");
  printf("enter␣x:␣");
  fgets(str, sizeof(str), stdin);
  sscanf(str, "%lf", &x);

  if (x<0){
    C_expfun3(-x, &ans);
    ans = 1/ans;
  } else {
    C_expfun3(x, &ans);
  }

  double expx = exp(x);
  double rerr = (ans-expx)/expx;
  printf("input=%f␣ans=%f␣expx=%f␣rerr=%f\n", x, ans, expx, rerr);
}
```

The workhorse for calculating the exponential is simple; everything else in the programs is about getting data in and out.

```

void C_expfun3 (double x, double *ans){
  double term=1.0, eps=1e-16;
  int n=0;
  while (fabs(term)>eps){
    n++;
    term = term * x / n;
    *ans = *ans + term;
  }
}
```

This simple workhorse can be called directly from R. More about this later.

3.2 Example: Roots of 2nd degree polynomial

```c
/* Compile into findroots2 with gcc -o findroots2 findroots2.c */
#include <stdio.h>
#include <math.h>
# define nroot 2
FILE *output;

void getroots (double parm[3], double roots[nroot]) {
  roots[0] = (-parm[1] + sqrt(pow(parm[1],2) - 4*parm[0]*parm[2]))/(2*parm[0]);
  roots[1] = (-parm[1] - sqrt(pow(parm[1],2) - 4*parm[0]*parm[2]))/(2*parm[0]);
  return;
}

int main(void){
  int i;
  double coef[3], roots[nroot];
  char line[256];

  printf("Solving␣a␣x^2␣+␣b␣x␣+␣c␣=␣0\n");
  printf("set␣a:␣");  fgets(line, sizeof(line), stdin);
  sscanf(line, "%lf", &coef[0]);
  printf("set␣b:␣");  fgets(line, sizeof(line), stdin);
  sscanf(line, "%lf", &coef[1]);
  printf("set␣c:␣");  fgets(line, sizeof(line), stdin);
  sscanf(line, "%lf", &coef[2]);

  sprintf(line, "a=%f␣b=%f,␣c=%f\n", coef[0], coef[1], coef[2]);
  printf(line);

  getroots(coef,roots);
  printf("Solution:␣%f␣%f", roots[0], roots[1]);

  output = fopen("getroots.output.csv","w");
  for(i=0;i<nroot;i++) {
    fprintf(output,"%f,",roots[i]);
  }
  fprintf(output,"\n");
  fclose(output);
  return;
}
```

4 Further reading

- In general: The manual "Writing R Extensions"
- `C` programming: *The C Programming Language* by Kernighan and Ritchie. A classic.
- `C` programming: `http://www.cprogramming.com/`

5 Interfacing `C`–code with `.C()` and `.Call()`

`C` functions can be called from `R` using `.C()` and `.Call()`.

- The `.C()` function is the basic way of calling `C` functions from `R`.

- With `.C()` all arguments must be "primitive" (essentially vectors of doubles or integers).
- `.C()` "returns" only primitive values.
- The `.Call()` interface is more difficult but offers several advantages
- `.Call()` accepts `R` data structures such as matrices and lists.
- `.Call()` returns `R` data structures.
- (`Fortran` code can be called in a way similar to `.C()` using `.Fortran()`. We shall not discuss calling `Fortran` in detail.)

We illustrate the mechanisms by considering matrix multiplication (which is something you do not want to write your own `C` functions for; those in `R` are very good).

```
R> A <- matrix(1:12, nc=6)
R> B <- matrix(1:18, nr=6)
R> A %*% B

     [,1] [,2] [,3]
[1,]  161  377  593
[2,]  182  434  686
```

5.1 Using .C()

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

```
1  /* File: matprod1.c: Calculate the product of matrices X and Y */
2  void matprod1(double *X, int *dimX, double *Y, int *dimY, double *ans){
3    double sum;
4    int ii, jj, kk;
5    int nrX=dimX[0], ncX=dimX[1], nrY=dimY[0], ncY=dimY[1];
6
7    for (ii=0; ii<nrX; ii++){
8      for (jj=0; jj<ncY; jj++){
9        sum = 0;
10       for (kk=0; kk<ncX; kk++){
11         sum = sum + X[ii+nrX*kk]*Y[kk+nrY*jj];
12       }
13       ans[ii+nrX*jj] = sum;
14     }
15   }
16 }
```

To compile the `C` function, issue the following command at the command prompt/in a shell:

```
R --vanilla CMD SHLIB src/matprod1.c
```

On Windows this creates `matprod1.dll`; on Linux this creates `matprod1.so`.

The function is called with:

```
R> mprod1_C_C <- function(A,B){ # Using .C and own C code
+    ans1 <- .C("matprod1", as.numeric(A), dim(A), as.numeric(B), dim(B),
+               result=numeric(nrow(A)*ncol(B)))$result
+    dim(ans1) <- c(nrow(A), ncol(B))
+    ans1
+  }
R> dyn.load("src/matprod1.dll")
R> mprod1_C_C(A, B)

     [,1] [,2] [,3]
[1,]  161  377  593
[2,]  182  434  686

R> dyn.unload("src/matprod1.dll")
```

Notice the following:

- Matrices are converted to vectors (of the right type) and dimensions of the matrices are supplied explicitly.
- A vector designated to hold the answer is created and supplied as well.

5.2 Using `.Call()`

It would be much more convenient to be able to simply pass the matrices A and B to the matrix multiplication function without having to pass the dimensions explicitly etc.

This can be done using SEXPs (S expressions).

- R objects are SEXPs.
- The SEXPs are passed to the C code where they are "digested".
- Advantage: calling the C-function from R is simpler and faster code is often obtained.
- Disadvantage: The C code gets more involved.

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

```
1  /* File: matprod2.c: Calculates the product of matrices X and Y */
2  #include <Rdefines.h>
3  #include "matprod1.h"
4
5  SEXP matprod2(SEXP X, SEXP Y) {
6    int nprot=0;
7    PROTECT(X = AS_NUMERIC(X)); nprot++;      /* Digest SEXPs from R */
8    PROTECT(Y = AS_NUMERIC(Y)); nprot++;
9    double *xptr; xptr = REAL(X);
10   double *yptr; yptr = REAL(Y);
11   int *dimX; dimX = INTEGER(GET_DIM(X));
12   int *dimY; dimY = INTEGER(GET_DIM(Y));
13   SEXP ans;                                 /* Create SEXP to hold result */
14   PROTECT(ans = allocMatrix(REALSXP, dimX[0], dimY[1])); nprot++;
15   double *ansptr; ansptr = REAL(ans);
16   matprod1(xptr, dimX, yptr, dimY, ansptr); /* Calculate product */
```

```
  UNPROTECT(nprot);                          /* Wrap up; */
  return(ans);                               /* Return the result to R */
}
```

Above we used the header file `matprod1.h`:

```
void matprod1(double *X, int *dimX, double *Y, int *dimY, double *ans);
```

To compile the file do:

```
R --vanilla CMD SHLIB src/matprod2.c src/matprod1.c
```

Load the file with:

```
R> mprod2_Call_C <- function(A,B){ # Using .Call and own C code
+    .Call("matprod2", A, B)
+  }
R> dyn.load("src/matprod2.dll")
R> mprod2_Call_C(A, B)

     [,1] [,2] [,3]
[1,]  161  377  593
[2,]  182  434  686

R> dyn.unload("src/matprod2.dll")
```

5.3 Using linear algebra libraries

There are various libraries with linear algebra routines available online, e.g.

- blas (`http://www.netlib.org/blas/`) or
- lapack (`http://www.netlib.org/lapack/`) or
- linpack (`http://www.netlib.org/linpack/`) or

These routines are implemented in `Fortran` and the libraries are available in `R`.

Matrix multiplication can be made using the routine `dgemm in blas` as:

```
/* File: matprod3.c */
#include <R.h>
#include <R_ext/BLAS.h>

void matprod3(double *X, int *dimX, double *Y, int *dimY, double *ans){

  int nrX=dimX[0], ncX=dimX[1], nrY=dimY[0], ncY=dimY[1];

  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);
}
```

We may alternatively use the `.Call()` interface:

```
/* File: matprod4.c */
#include <R.h>
#include <Rdefines.h>
#include <R_ext/BLAS.h>

SEXP matprod4(SEXP X, SEXP Y) {
  int nprot=0;
  PROTECT(X = AS_NUMERIC(X)); nprot++;
  PROTECT(Y = AS_NUMERIC(Y)); nprot++;
  double *xptr; xptr = REAL(X);
  double *yptr; yptr = REAL(Y);
  int *dimX; dimX = INTEGER(GET_DIM(X));
  int *dimY; dimY = INTEGER(GET_DIM(Y));

  SEXP ans; PROTECT(ans = allocMatrix(REALSXP, dimX[0], dimY[1])); nprot++;
  double *ansptr; ansptr = REAL(ans);

  char *transa = "N", *transb = "N";
  double one = 1.0, zero = 0.0;
  F77_CALL(dgemm)(transa, transb, &dimX[0], &dimY[1], &dimX[1], &one,
                  xptr, &dimX[0], yptr, &dimY[0], &zero, ansptr, &dimX[0]);
  UNPROTECT(nprot);
  return(ans);
}
```

To compile these files, first create a file named `Makevars` with the contents

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

and then compile as described above – or compile both using

```
R --vanilla CMD SHLIB src/matprod3.c src/matprod4.c
```

which creates `matprod3.dll`.

The calls are as before

```
R> mprod3_C_For <- function(A,B){ # Using .C and a Fortran routine
+    ans1 <- .C("matprod3", as.numeric(A), dim(A), as.numeric(B), dim(B),
+               result=numeric(nrow(A)*ncol(B)))$result
+    dim(ans1) <- c(nrow(A), ncol(B))
+    ans1
+  }
R> mprod4_Call_For <- function(A,B){ # Using .Call and a Fortran routine
+    .Call("matprod4", A, B)
+  }
R> dyn.load("src/matprod3.dll")
R> mprod3_C_For(A, B)

     [,1] [,2] [,3]
[1,]  161  377  593
[2,]  182  434  686

R> mprod4_Call_For(A, B)

     [,1] [,2] [,3]
[1,]  161  377  593
[2,]  182  434  686

R> dyn.unload("src/matprod3.dll")
```

5.4 Benchmarking - I

Compile everything:

```
R CMD SHLIB -o src/matprod.dll src/matprod4.c src/matprod3.c src/matprod2.c src/matprod1.c
```

Compare performances:

```
R> library(rbenchmark)
R> cols <- c("test", "replications", "elapsed", "relative")
```

```
R> A <- matrix(rnorm(100), nr=10); B<-A[,1:5]
R> dyn.load("src/matprod.dll")
R> N <- 50000
R> benchmark(A %*% B, mprod1_C_C(A,B), mprod2_Call_C(A,B), mprod3_C_For(A,B),
+            mprod4_Call_For(A,B), columns=cols, order="relative", replications=N)

                  test replications elapsed relative
3   mprod2_Call_C(A, B)        50000    0.56    1.000
1               A %*% B        50000    0.70    1.250
5 mprod4_Call_For(A, B)        50000    0.86    1.536
4    mprod3_C_For(A, B)        50000    1.61    2.875
2      mprod1_C_C(A, B)        50000    2.02    3.607

R> dyn.unload("src/matprod.dll")
```

Notice:

- Timewise, `.C()` performs poor relative to `.Call()`. There are two (at least) reasons:

- On the R side, `as.numeric()`, `numeric()` etc. takes time.
- The arguments are copied when calling the C code, and that takes time.

6 Calling R from C

It is possible to call R functions from within C. Consider the following (not very elegant) way of generating observations from a χ^2_f–distribution.

```c
#include <R.h>
#include <Rmath.h>
#include <math.h>

void myrchisq1(int *n, int *df, double *ans){
  double chisq, tmp;
  GetRNGstate();             /* set random number seed      */
  for (int jj=0; jj<*n; jj++){  //Rprintf("jj=%i\n", jj);
    chisq=0;
    for (int ii=0; ii<*df; ii++){
      tmp = rnorm(0.0,1.0); /* call R function */
      //Rprintf("%2ith draw= %8.4f \n",(ii+1),tmp);
      chisq=chisq + tmp*tmp;
    }
    ans[jj] = chisq;
  }
  PutRNGstate();             /* write back ran number seed */
}
```

An alternative interface to the function is provided as follows:

```c
#include <R.h>
#include <Rmath.h>
#include <math.h>
#include <Rdefines.h>

SEXP myrchisq2(SEXP n, SEXP df){
  int nprot=0;
  PROTECT(n); nprot++;
  PROTECT(df); nprot++;
  int _df = INTEGER_VALUE(df);
  int _n  = INTEGER_VALUE(n);

  SEXP ans;
  PROTECT(ans=NEW_NUMERIC(_n)); nprot++;
  double *ansptr; ansptr = REAL(ans);
  myrchisq1(&_n, &_df, ansptr);
  UNPROTECT(nprot);
  return ans;
}
```

```
R --vanilla CMD SHLIB -o src/callRfromC.dll src/callRfromC2.c src/callRfromC1.c
```

```
R> nn <- 10
R> nu <- 7
R> dyn.load("src/callRfromC.dll")
R> .C("myrchisq1", as.integer(nn), as.integer(nu), ans=numeric(nn))$ans

 [1] 14.438440  3.110139  4.501082  3.949214  5.574821 12.023367  7.682628
 [8] 11.469489  6.076906  9.359699

R> .Call("myrchisq2", nn, nu)

 [1]  1.922361  1.126529  6.489263  5.867347  5.965098  9.346552  4.489872
 [8]  3.928444 11.865221 14.109356

R> dyn.unload("src/callRfromC.dll")
```

7 Creating `C` code on the fly using **inline**

- The **inline** package allows to define `R` functions with in-lined `C` (or `C++`) code.
- The **inline** package is very good for prototyping: getting code up and running quickly.
- Both `.C()` and `.Call()` may be used as calling mechanisms.
- The `C` (or `C++`) code is given as a text string.

7.1 Example: The exponential function – using `.C()`

```
R> expfun.src <- '
+    double term=1.0, eps=1e-16, ans=1.0;
+    int n=0;
+    while (fabs(term)>eps){
+      n++;
+      term = term * *x / n;
+      ans = ans + term;
+    }
+    *res = ans;
+  '
R> library(inline)
R> expfun <- cfunction(signature(x="numeric", res="numeric"), expfun.src, convention=".C")
R> expfun

An object of class "CFunc"
function (x, res)
.Primitive(".C")(<pointer: 0x0000000067e814b0>, x = as.double(x),
    res = as.double(res))
<environment: 0x000000000aa85fe0>
Slot "code":
[1] "#include <R.h>\n\n\nextern \"C\" {\n  void filec34479062e4 ( double * x, double * res );\n}\n\nvoid filec34479062e4 ( dou
```

The function:

```
R> cat(expfun@code)

#include <R.h>


extern "C" {
  void filec34479062e4 ( double * x, double * res );
}

void filec34479062e4 ( double * x, double * res ) {

  double term=1.0, eps=1e-16, ans=1.0;
  int n=0;
  while (fabs(term)>eps){
    n++;
    term = term * *x / n;
    ans = ans + term;
  }
  *res = ans;

}
```

```
R> x <- 1
R> expfun(x, numeric(1))$res

[1] 2.718282

R> (expfun(x, numeric(1))$res - exp(x)) / exp(x)

[1] 1.633713e-16

R> x <- 20
R> expfun(x, numeric(1))$res

[1] 485165195

R> (expfun(x, numeric(1))$res - exp(x)) / exp(x)

[1] -1.228543e-16

R> x<- -20
R> expfun(x, numeric(1))$res

[1] 5.621884e-09

R> (expfun(x, numeric(1))$res - exp(x)) / exp(x)

[1] 1.727543
```

7.2 Example: Square the numbers $1, \ldots, n$ – using `.C()`

```
R> codeSq <- "
+    for (int i=0; i < *n; i++) {
+      x[i] = x[i]*x[i];
+    }"
R> library("inline")
R> sqfn <- cfunction(signature(n="integer", x="numeric"), codeSq, convention=".C")
R> x <- as.numeric(1:10)
R> n <- as.integer(10)
R> sqfn(n, x)$x

 [1]   1   4   9  16  25  36  49  64  81 100
```

7.3 Example: Multiplying matrices – using `.Call()`

```
R> codeMatProd <- '
+  /* Calculates the product of matrices X and Y */
+    int nprot=0;
+    PROTECT(X = AS_NUMERIC(X)); nprot++;
+    PROTECT(Y = AS_NUMERIC(Y)); nprot++;
+    double *xptr;    xptr = REAL(X);
+    double *yptr;    yptr = REAL(Y);
+    int nrX = INTEGER(GET_DIM(X))[0], ncX = INTEGER(GET_DIM(X))[1];
+    int nrY = INTEGER(GET_DIM(Y))[0], ncY = INTEGER(GET_DIM(Y))[1];
+    SEXP ans; PROTECT(ans = allocMatrix(REALSXP, nrX, ncY)); nprot++;
+    double *ansptr; ansptr = REAL(ans);
+
+    int ii, jj, kk;
+    for (ii=0; ii<nrX; ii++){
+      for (jj=0; jj<ncY; jj++){
+        double sum = 0;
+        for (kk=0; kk<ncX; kk++){
+          sum += xptr[ii+nrX*kk]*yptr[kk+nrY*jj];
+        }
+        ansptr[ii+nrX*jj] = sum;
+      }
+    }
+    UNPROTECT(nprot);
+    return ans; '
```

```
R> library(inline)
R> mprod5_inline_Call <- cfunction(signature(X = "matrix", Y="matrix"),
+                                  body=codeMatProd,
+                                  convention=".Call")
```

7.4 Benchmarking - II

```
R> A <- matrix(rnorm(100), nr=10); B<-A[,1:5]
R> dyn.load("src/matprod.dll")
R> N <- 2000
R> benchmark(A %*%B, mprod1_C_C(A,B),mprod2_Call_C(A,B),mprod3_C_For(A,B),
+            mprod4_Call_For(A,B),  mprod5_inline_Call(A,B),
+            columns=cols, order="relative", replications=N)


                     test replications elapsed relative
1                  A %*% B         2000    0.03    1.000
3      mprod2_Call_C(A, B)         2000    0.03    1.000
5    mprod4_Call_For(A, B)         2000    0.03    1.000
6 mprod5_inline_Call(A, B)         2000    0.03    1.000
4       mprod3_C_For(A, B)         2000    0.11    3.667
2         mprod1_C_C(A, B)         2000    0.13    4.333
```

8 EXERCISES: C programming

1. Write a C–program that will find the roots of a 2nd degree polynomial
2. Interface this C–program from R.
3. Can you beat Rs built–in function for matrix multiplication %*% in terms of computing time? In this quest you are allowed to use $A^\top$ as input when computing AB. You are also allowed to use parallel computing.