

Calling C Functions in R and Matlab

Phil Spector (spector@stat.berkeley.edu)
Statistical Computing Facility
Department of Statistics
University of California, Berkeley

1 Introduction

Suppose we have a C function called `colmeans`, which calculates the mean of each column of a matrix. (Both R and matlab have very efficient ways of doing this, but it serves as a good example.) Here's the source code of the function, stored in the file `colmeans.c`:

```
void colmeans(double *x,int ncol,int nrow,double *ans)
{
    long i,j;
    double *tmp;

    tmp = x;
    for(i=0;i<ncol;i++){
        ans[i] = 0.;
        for(j=0;j<nrow;j++)ans[i] += *tmp++;
        ans[i] /= (double)nrow;
    }
}
```

Note that the function is written under the assumption that matrices are stored as vectors, with the columns stacked on each other. In other words, the first `nrow` elements of `x` represent the first column of the matrix, the next `nrow` elements represent the second column, and so on. This is the way that Fortran, R, and matlab all store matrices, and any matrix function which will be called from R or matlab must follow this convention. Note that no special care was taken when writing the function to make sure it will be callable from R or matlab. In general, it's not necessary to make any changes to your function simply because you're going to call it from R or matlab, so almost any C functions you've written will be usable in those languages.

2 R

There are two methods of calling C functions from R. The first, based on an R function called `.C` requires that you map the arguments of your C function one-to-one with arguments you pass from R. So, for example, when working with matrices, you'd need to pass not only a pointer to the matrix, but the number of rows and columns. Furthermore, when using `.C`, all the arguments to your C function must be pointers to a suitable location in memory that either contains the input arguments or provides enough space to contain an output argument, and your C function can not return a value.

The second method, based on an R function called `.Call` allows you to pass R objects into the C environment and provides macros and functions to extract the information you need from these objects. When using this mechanism, your function can return an R object directly. We'll look at each method in the next two subsections.

2.1 .C

Since `.C` demands that all of its arguments are pointers, and some of the arguments to `colmeans` are not pointers, we would write a wrapper function as follows, in a file called `colmeansr.c`:

```
void colmeans(double*,int,int,double *);

void colmeansr(double *x, int *ncol,int *nrow,double *ans)
{
    colmeans(x,*ncol,*nrow,ans);
}
```

To create the shared object necessary to access this function, execute the following command:

```
R CMD SHLIB colmeansr.c colmeans.c
```

This will generate a shared object file called `colmeansr.so` on Linux and Unix systems, and a dynamically loaded file called `colmeansr.dll` on Windows systems.. Remember that if you want to test your function on a different architecture, you'll need to delete any object files (`.o` and `.so`) before recompiling.

In order to access a C function in R, the shared object needs to be dynamically loaded into the running R process. The basic idea of dynamic loading is that new symbols are entered into the symbol table of an executing program so that it has access to new functions without the need of recompiling the executing program. In R, the `dyn.load` function takes care of this task. Now that we've created a shared object, we can dynamically load it into a running R session with the R command:

```
dyn.load('./colmeansr.so')
```

It's advisable to pass the complete path of your shared object to `dyn.load` – in this case, we're telling R that `colmeansr.so` is located in the current directory. Having loaded the shared object, we access it from R using the `.C` function. The first argument to this function is the name of the C function that you will be executing, in this case `colmeansr`. The remaining arguments should have a one-to-one correspondence with the arguments to your function, so in this case there should be four arguments after the function name. To insure that the correct type of object gets passed from C to R, you can use functions like `as.double` or `as.integer` to insure that R will pass pointers to the appropriate data types to your function. The following table shows how to declare variables in a C or Fortran program, and their corresponding types in R.

R	C	FORTRAN
"single"	float*	real
"double"	double*	double precision
"integer"	int*	integer
"character"	char**	character
"complex"	struct { double re,im; }*	double complex
"list"	char**	-

Unfortunately, using these coercion functions causes R to forget about the matrix nature of any of its arguments. For matrices, it's easier to set the storage mode to one of the types in the first column of the table, as shown in the example below.

The return value from `.C` is a list with elements corresponding to each of the arguments passed to the C function. Thus elements can be extracted using subscripting, or by naming the argument when it's passed to `.C` and extracting by name.

Here's how we'd call the `colmeansr` function:

```
mycolmeans = function(mat){
    nr = nrow(mat)
```

```

nc = ncol(mat)
result = numeric(nc); # or rep(0,nc);
if(!is.loaded('colmeansr')) dyn.load('./colmeansr.so')
storage.mode(mat) = 'double'
z = .C('colmeansr',mat,as.integer(nc),as.integer(nr),result = as.double(result))
return(z$result)
}

```

Note the use of the `is.loaded` function. When you're first developing functions to use with R, it will be necessary to reload the shared object each time you make a change, so you wouldn't use `is.loaded`. But once you'll actually be using the function in practice, you'd want to insert the `is.loaded` check so that the object isn't loaded each time you call the function.

It should be mentioned that although the `colmeans` C function accepts four arguments, we only need to pass the matrix itself to our R function, since we can extract or create the other arguments inside the function. When calling a C function from R, you should always insure that you are not passing any information to the R function that it can figure out for itself.

Once the shared object has been dynamically loaded, and the function definition is sourced or pasted into R, it can be accessed like any other function.

2.2 .Call

The major difference between `.C` and `.Call` is that, when using `.Call`, you can simply pass an R object into the C environment, without having to match up each argument of your C function. Instead, you can extract information as needed from the R object that you pass to C. In addition, your function can also return an R object directly into the R environment. Note that, unlike the `.C` mechanism, pointers to data are *not* extracted or copied before entering the C environment – all the objects passed into or out of the C environment must be R objects.

When writing a C function for use with `.Call`, the `SEXP` typedef allows you to receive and create R objects. Various macro variables identify R's basic types: `REALSXP` for doubles, `INTSXP` for integers, and `STRSXP` for strings. (There are also types for lists, symbols, functions and other R objects.) New R objects can be produced using the functions `allocVector` or `allocMatrix`, in each case returning `SEXP` objects. To get pointers to the data that these objects contain, the `REAL` or `INTEGER` macros can be used. Finally, attributes of the R objects can be extracted using the `getAttrib` function. All the necessary prototypes and macro definitions are found in the header files `R.h` and `Rinternals.h`, so those two files should be included whenever you'll be using the `.Call` interface, and can be inspected to learn more about how the `.Call` mechanism works. (There are also macro definitions in the `Rdefines.h` header file which provide compatibility with `Splus`, as well as simplifying some operations.) Here's a version of the `colmeansr` function, rewritten for use with the `.Call` interface.

```

#include <R.h>
#include <Rinternals.h>

void colmeans(double*,int,int,double *);

SEXP colmeansr(SEXP x)
{
    double *mat,*ans;
    int ncol,nrow;
    SEXP Rdim,answer;

    mat = REAL(x);
    Rdim = getAttrib(x,R_DimSymbol);

```

```

nrow = INTEGER(Rdim)[0];
ncol = INTEGER(Rdim)[1];

PROTECT(answer = allocVector(REALSXP, ncol));
ans = REAL(answer);

colmeans(mat, ncol, nrow, ans);
UNPROTECT(1);
return(answer);
}

```

Since memory obtained through `allocVector` is managed by R, there is a slight possibility that it would accidentally reclaim that memory before our function was done. In order to insure that this doesn't happen, we wrap the allocation in a call to the `PROTECT` macro, and then then call `UNPROTECT`, passing the number of `PROTECT` calls we wish to cancel, right before we return the object to the R environment.

One of the main benefits of using `.Call` is the simplicity with which we can access our C routine from R. After building the shared library using

```
R CMD SHLIB colmeansr1.c colmeans.c
```

this is the function we'd use to access the C routine:

```

mycolmeans = function(mat){
  if(!is.loaded('colmeansr')) dyn.load('./colmeansr1.so')
  return(.Call('colmeansr', mat))
}

```

For both the `.C` and `.Call` interfaces, any printing done in the C program loaded into R should use the `Rprintf` function for printing. `Rprintf` is called identically to `printf`, but guarantees that what you print will display correctly, even if a GUI interface is used. The `REprintf` function can be used to direct messages to standard error instead of standard output.

3 matlab

To access a C function in `matlab`, a small C function, similar to the program required by the `.Call` interface, needs to be written to extract arguments and set result values. For the current example, this C function would look like this, stored in a file called `colmeansm.c`:

```

#include "math.h"
#include "mex.h"

EXTERN_C void mexFunction(int nls, mxArray **plhs, int nrhs, const mxArray **prhs)
{
  long nrow, ncol;
  double *mat, *result;
  void colmeans(double*, int, int, double*);

  nrow = mxGetM(prhs[0]);
  ncol = mxGetN(prhs[0]);

  plhs[0] = mxCreateDoubleMatrix(ncol, 1, mxREAL);
  result = mxGetPr(plhs[0]);
}

```

```

mat = mxGetPr(prhs[0]);

colmeans(mat,ncol,nrow,result);

}

```

As can be seen, all C functions which will be accessed through matlab are named `mexFunction`; matlab will determine what function to use based on the file name of the output object. The arguments to the function correspond to the number of left-hand side arguments(output), an array of pointers to these objects, the number of right-hand side arguments(input), and a constant pointer to these values. (You can't change the values of arguments passed to matlab functions.) Any memory needed for results should be allocated using `mxCreateDoubleMatrix`; if it's necessary to free this memory, it should be done by calling `mxFree`. A call to `mxGetPr` must be used to extract the pointer to the actual data from the internal matlab objects.

To compile this function for use with matlab, the `mex` command should be used; you don't call the compiler directly. (You can safely ignore any warnings about the gcc version which will be generated by `mex`.) `mex` is a shell script, provided by matlab, which creates an output object that can be accessed as an ordinary matlab command. To account for multiple architectures, the output from `mex` has a different suffix representing the architecture under which it was built. For example, the command

```
mex colmeansm.c colmeans.c
```

will produce a file called `colmeansm.mexa64` on a 64-bit Linux system, while it will produce a file called `colmeansm.mexmaci` on an Intel Mac. These files will be recognized as matlab commands if they are located in the current directory, or in a `matlab` subdirectory of your home directory. Thus, since only one left-hand side argument and one-right hand side argument are being used, we would simply call the function as

```
result = colmeansm(mat)
```

4 Calling R Functions from C

Just as with calling C functions from R, there are two mechanisms for calling R functions from within a C function. The first, using a C function called `call_R`, allows you to construct a call to any R function from your C function. To do this, you need to create vectors containing information about the modes and lengths of the arguments you will be passing to the R function, as well as where you want the results. The second method, using the R function `.Call` constructs an actual call to the desired R function inside your C function, using R objects in much the same way that `.Call` allows us to use R objects inside a C function.

As an example of calling R functions from C, we'll use Simpson's rule to perform numerical integration with the ability to define the function we'll be integrating in R. First, here's a C function, stored in the file `simp.c` which implements Simpson's rule:

```

double simp(double (*func)(double),double start,double stop,long n)
{
    double mult,x,t,t1,inc;
    long i;

    inc = (stop - start) / (double)n;

    t = func(x = start);
    mult = 4.;
    for(i=1;i<n;i++){
        x += inc;
        t += mult * func(x);
    }
}

```

```

        mult = mult == 4. ? 2. : 4.;
    }
    t += func(stop);

    return(t * inc / 3.);
}

```

The next two subsections will show how to implement Simpson's rule in R, first using `call_R`, and then using `.Call`.

4.1 `call_R`

We'll start by writing an R function that will access our C function containing `call_R`. First, we need to pass the name of the function we wish to call as a list to our C program, along with the starting and stopping points for the integration, as well as the number of function evaluations we'll use. This R function will do that:

```

simpson = function(thefunc, start, stop, n=100) {
  if(!is.loaded('dosimp')) dyn.load('./dosimp.so')
  result = 0
  z = .C("dosimp", list(thefunc), as.double(start), as.double(stop),
        as.integer(n), result=as.double(result))
  z$result
}

```

Notice that the function we'll be using needs to be passed into C as a list. Inside the C program, a pointer will be extracted from this list that will let `call_R` identify the R function it needs to call. Here's the C program, stored in the file `dosimp.c`:

```

#include <R.h>

static char *sfunction;
double simp(double(*)(), double, double, long);

void dosimp(char **funclist, double *start, double *stop, long *n, double *answer)
{
    double sfunc(double);
    sfunction = funclist[0];
    *answer = simp(sfunc, *start, *stop, *n);
}

double sfunc(double x)
{
    char *modes[1];
    void *arguments[1];
    double *result;
    long lengths[1];

    lengths[0] = (long)1;
    arguments[0] = (void*)&x;
    modes[0] = "double";

    call_R(sfunction, (long)1, arguments, modes, lengths,

```

```

        NULL, (long)1, (char**)arguments);

    result = (double*)arguments[0];

    return(*result);
}

```

The function prototype for `call_R` is contained in the header file `R.h`, so that file should be included whenever you use `call_R`. To accomodate all possible arguments, the `arguments` vector is declared as a pointer to void, and it will be cast appropriately inside of `call_R`. The length of the `arguments` vector is one in this example, since we will only be passing one argument (the value of `x` to be evaluated) to the function we're integrating. The first argument to the `dosimp` C function is a list containing a pointer to the R function that we'll be integrating. Since the `simp` function requires a pointer to a function that accepts just one argument, we create a static variable (`sfunction`) to hold this pointer so that it can be referenced in the C function that will actually use `call_R` to evaluate the R function. All the `dosimp` function has to do is to make the pointer available to our `sfunc` function, and call `simp` with the appropriate arguments. To actually evaluate the R function (in the C function `sfunc`), we pass `call_R` a pointer to the R function we'll be integrating, followed by the number of input arguments, a vector of pointers to the arguments themselves (placed into a vector which has been declared as `void*` to accomodate any data type), and their lengths. This is followed by a vector of names to indicate what input arguments passed to `call_R` correspond to the named arguments of the R function being called. In this case, we can pass a `NULL` value, since we don't need the names. Finally, the number of output arguments and their location are passed to `call_R`. While this example didn't need to allocate any memory (since the input and output were both scalars, the same memory was used for both), the `R_alloc` and `R_free` functions are available for this purpose. When you use these functions, R can properly manage the memory you allocate, freeing it when your function has completed. Unlike `malloc`, `R_alloc` requires two arguments, the number of objects you need, and the size of each one.

To create the shared library which needs to be loaded by the `simpson` function, use the following command:

```
R CMD SHLIB dosimp.c simp.c
```

4.2 .Call

When using `.Call` to access an R function inside C, the function itself is passed into the C environment, where an R language object representing the function call is constructed and then evaluated. In order to do this, an environment needs to be specified for the evaluation; thus you'll need to pass an R environment to your C function. If you don't have a special need, passing `new.env()` will usually suffice. Since all the arguments which are passed to `.Call` are R objects, we need to pass any non-R arguments (like the number of repetitions and the number of random numbers to use in this case) to our function in a vector, and then extract them inside our C program. Here's a function that will use `.Call` to integrate an R function using Simpson's rule.

```

simpson = function(fun,start,stop,n=100){
  if(!is.loaded('dosimp'))dyn.load('./dosimp1.so')
  .Call("dosimp",c(start,stop,n),fun,new.env())
}

```

As in the previous example, using `.Call` results in a much simpler function.

Here's the C code for use with `.Call`, stored in a file called `dosimp1.c`:

```

#include <R.h>
#include <Rinternals.h>

```

```

static SEXP thefun,theenv;
double simp(double(*)(),double,double,long);

SEXP dosimp(SEXP args,SEXP function,SEXP env)
{
    double start,stop,nintvl,res;
    SEXP answer;
    double evalthefun(double);

    thefun = function;

    start = REAL(args)[0];
    stop  = REAL(args)[1];
    nintvl= REAL(args)[2];

    res = simp(evalthefun,start,stop,(long)nintvl);
    PROTECT(answer = allocVector(REALSXP,1));
    REAL(answer)[0] = res;
    UNPROTECT(1);
    return(answer);
}

double evalthefun(double x)
{
    SEXP rargs,Rcall,result,out;
    rargs = allocVector(REALSXP,1);
    REAL(rargs)[0] = x;

    PROTECT(Rcall = lang2(thefun,rargs));
    PROTECT(result = eval(Rcall,theenv));
    PROTECT(out = allocVector(REALSXP,1));

    out = result;
    UNPROTECT(3);
    return(REAL(result)[0]);
}

```

As in the `call_R` example, the pointer to the function to be integrated needs to be made static so that it can be accessed in `evalthefun`. The `evalthefun` function allocates space for the argument to the function being called, and then uses a two step process of creating an R language object with `lang2`, and then evaluating it with `eval`. Memory is allocated to hold the result, and all three occurrences are unprotected immediately before the result is returned to the R environment. The `dosimp` function simply extracts the input arguments and constructs the call to the `simp` function, allocates space to hold the answer, unprotected it and passes it back into the R environment. As always, the shared object is constructed using the following command

```
R CMD SHLIB dosimp1.c simp.c
```


5 Calling matlab functions from C

The `mexCallMATLAB` function makes it easy to call matlab functions from within a C program. To use it, you must first create appropriate arrays of matlab objects to hold the input arguments to the function, as well as a pointer to accept the results which the matlab function will generate. The matlab function you're calling takes care of allocating memory for its result, so all you need to do is pass the address of an appropriate pointer to `mexCallMATLAB`. (Notice that you must pass the address of the pointer, because the pointer's value is going to be changed inside of `mexCallMATLAB`.) Next, the number of input arguments and their values are passed to `mexCallMATLAB`, finally followed by a character string with the name of the matlab function you want to call. Here's an example of a C function that will implement Simpson's rule in matlab, stored in a file called `dosimp.c`:

```
#include <stdio.h>
#include "math.h"
#include "mex.h"

static char *thefunction;
double simp(double(*)(), double, double, long);

EXTERN_C void mexFunction(int nls, mxArray **plhs, int nrhs, const mxArray **prhs)
{
    double stop, start, res;
    char fun[100];
    long nintvls;
    mxArray *args[1];
    double evalthefun(double);

    if(nrhs < 3){
        mexErrMsgTxt("dosimp requires at least three arguments\n");
    }

    mxGetString(prhs[0], fun, 100);
    thefunction = fun;

    start = (double)mxGetScalar(prhs[1]);
    stop = (double)mxGetScalar(prhs[2]);
    if(nrhs == 4) nintvls = (long)mxGetScalar(prhs[3]);
    else nintvls = 100;

    plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);

    res = simp(evalthefun, start, stop, nintvls);
    *mxGetPr(plhs[0]) = res;
}

double evalthefun(double x){
    mxArray *mtmp, *args[1];

    args[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxGetPr(args[0])[0] = x;
```

```

    mexCallMATLAB(1, &mtmp, 1, args, thefunction);
    return(*mxGetPr(mtmp));
}

```

Since matlab functions can accept any number of arguments, it's a good idea to use the `nrhs` argument to see how many arguments were passed to the function. The `mexErrMsgTxt` function is used identically to `printf`, but returns from your `mexFunction` after writing the message and freeing any memory which was allocated by your function. Other printing functions available inside `mexFunctions` include `mexPrintf` and `mexWarnMsgTxt`. Also notice how default values can be set inside the function, again by checking the number of right hand side arguments. We create a static variable to hold the name of the function so that the `evalthefun` function can have access to it, while still having just one input argument. Inside `evalthefun` we allocate space for the single argument the function to be integrated needs, and use `mxGetPr` to extract the pointer to the value and set it to the input argument `x`. Finally, we call `mexCallMATLAB`. The first argument indicates in this example that the length of the output (left-hand side) is 1. Thus we can simply pass a pointer to a single `mxArray` pointer; if we were returning several objects, we would pass the address to an array of `mxArray` pointers. The next two arguments are the number of arguments to the matlab function being called, and the arguments themselves, constructed using `mxCreateDoubleMatrix` to allocate memory, and `mxGetPr` to set the arguments to appropriate values. The result (which in the `evalthefun` function needs to be a double) is extracted using `mxGetPr`.

The `mexFunction` for this example simply extracts the four input arguments (function name, starting value, stopping value, and number of function evaluations), allocates space to hold the (scalar) answer, and calls the `simp` function, setting the output (left-hand side) to the appropriate value. The necessary file to allow the use of `dosimp` in matlab can be created by the following command:

```
mex dosimp.c simp.c
```