Adding Functions Written in C to R

P.Rossi 10/29/06

There may come a time when a function written in R runs so slowly that you must convert part of the function to some lower-level language such as C or FORTRAN. Typically, you will profile your code in R and convert only those parts of it which are a bottleneck. In the case of MCMC methods, this will often be loops over all n observations in a dataset. Of course, if these loops can be "vectorized" or written entirely in matrix operations, then there is no need to convert to the lower level language. In fact, most matrix operations will be faster in R than if re-written in C because R uses an optimized version of the BLAS to do matrix calculations.

If you are working in the LINUX environment, you can simply write C functions and compile them using the `R CMD SHLIB` command. This command uses the standard C compilers, PERL and various UNIX utilities to build what is called a dynamic linked library. A DLL can be loaded into R at anytime using the `dyn.load()` function.

If you are working under Windows, you will need to assemble the various compilers and tools needed. You will need three things:

1. A set of compatible tools which are various UNIX utilities ported to Windows .
2. A version of PERL for Windows.
3. The "minimal" version of GNU for windows (MinGW)

These can be found using the link

http://www.murdoch-sutherland.com/Rtools

MinGW includes the GNU C, C++, and F77 compilers. R is compiled using the GNU C compiler so you can be assured of compatibility. In addition, the GNU C compiler port for Windows has a good reputation for producing fast code (much faster than the Microsoft C compiler).

Download the tools.zip file and unzip into c:\Rtools.

Allow the MinGW and PERL .exe files to install these sets of programs in their default places (c:\Perl and c:\MinGW).

You must modify the path under windows so that R CMD SHLIB can find everything that it needs. To modify the path, right click on the "my computer" icon on the desktop and select "properties." Go to the "advanced tab" and click on environmental variables. Select the system variable path and make sure that it BEGINS with:

```
c:\rtools;c:\mingw\bin;c:\Perl\bin\;c:\PROGRA~1\R\rw2001\bin;
```

(assuming that you have R version 2.0.1 installed). To edit the path, select "edit" and use the arrow keys to advance to the beginning of the path and insert the above text.

Now you are ready to start writing code and creating DLLs.

A Simple Example

Let's write a very simple function in C, compile it, load into R and write an R interface to it.

One simple (but unnecessary task) would be to write a Chi-squared simulator using the "brute force" method of summing up the squares of independent standard normal random variables. Obviously, we don't need this as R has a built-in Chi-squared generator. Moreover, generating normals and adding them up is very inefficient. However, this example will demonstrate many aspects of C and the R interface. In particular, the example includes:

1. use of R internal C functions (see pages 55-62 of "Writing R extensions" for a complete list)
2. passing in pointers to both scalars and vectors
3. looping in C
4. rudimentary "debugging" via print statements

The code below is one implementation of the required task.

```
#include <R.h>
```

```
#include <Rmath.h>
#include <math.h>
/* include standard C math library and R internal function declarations    */

void mychisq(double *vec, double *chisq, int *nu)
                                        /* void means return nothing  */
{
      int i,iter;                       /* declare local vars */
                                        /* all statements end in;     */
      GetRNGstate();                    /* set random number seed     */
      for (i=0 ; i < *nu; ++i)
                                        /* loop over elements of vec */
                                        /*nu "dereferences" the pointer */
      {                                 /* vectors start at 0 location!*/
            vec[i] = rnorm(0.0,1.0);    /*use R function to draw normals */
            Rprintf("%ith normal draw= %lf \n",(i+1),vec[i]);
                              /* print out results for "debugging"      */
      }
      *chisq=0.0;
      iter=0;
      while(iter < *nu)                 /* "while" version of a loop  */
      {
            if( iter == iter)
                  {*chisq=*chisq + vec[iter]*vec[iter];}
                                        /* redundant if stmnt */
            iter=iter+1;                /* note: can't use ** */
            /* if you want to be "cool" use iter += 1  */
      }
      PutRNGstate();                    /* write back ran number seed */
}
```

Several notes about the differences between R and C are in order:

1. C allows the passing of pointers to functions so that copies are not made. In the example, vec, chisq, and int are pointers to memory locations holding doubles and integers. vec is a pointer to a contiguous block of memory holding the contents of a vector.
2. All variables must have a declared type. Here we are using pointer "types" as well as numeric types. C differentiates between integers and floating point numbers which R does not unless you tell it to.
3. To reference the contents of a memory location pointed to by a pointer, use the * before the pointer variable in an assignment statement.
4. Vectors start at location 0. vec[0] is the same as *(vec+0).
5. Loops are efficient unlike R.

To compile this function and create a DLL, simply issue the command

```
RCMD SHLIB <filename>
```

This will create a DLL with the same name as the file. A single DLL can contain many functions, each referenced by particular symbol. In this example, we have only one symbol we are interested in – the symbol mychisq.

To use this function, we must load the DLL into R by using the `dyn.load()` command and then write an interface function in R. The interface function will call the R system function `.C()` to pass arguments to our DLL.

```
call_mychisq=function(nu)
{
vector=double(nu); chisq=1
.C("mychisq",as.double(vector),res=as.double(chisq),as.integer(nu))$res
}
```

`call_mychisq` is nothing more than a wrapper to the `.C()` call. The first argument in the `.C()` call is the symbol name to be found in the current R symbol table from all loaded DLLs. The rest of the arguments are the arguments to the mychisq C function. Note that we coerce all of the R objects into the correct type needed by the C function. We name the coerced version of "chisq" "res" so that we can reference this. That is, the `.C()` call will return a list. The component of the list named "res" will be the "result" of mychisq. This is a copy of the argument to the C function which has been modified by the function.

To use the function, `dyn.load` it first and then call `call_mychisq` as in

```
dyn.load("<filename>.dll")
result=call_mychisq(10)
```

Note: if you make some changes to your program and then try to recompile and recreate the DLL while R has it dyn.loaded you will get a linker error. You must `dyn.unload()` it first!