# An Introduction to the `.C` Interface to R

Roger D. Peng    Jan de Leeuw
*UCLA Department of Statistics*

August 28, 2002

## 1 Introduction

It is easy to extend R with R code. You just write functions in R, save them in text files, and if you are very motivated you write documentation and organize them as R packages.

In this note we discuss one way to extend R with compiled C code. The code discussed in this note should work on any Unix installation of R and for the Darwin/X11 version, running on Mac OS X 10.1 or better. We do not discuss (yet) how to add C code to the Carbon version of R. This is more complicated, and certainly requires a different set of tools. Similarly, one can incorporate C code in Windows using MinGW or Cygwin, but it is somewhat more complicated than in Unix and is not discussed here.

The PDF manual *Writing R Extensions* provides many more details on incorporating C code into R. The manual is available from the CRAN website. It also covers the `.Call` interface which is somewhat more complicated to use but much more powerful. Here we only provide a brief introduction to `.C`.

## 2 Tools

We assume you have the `cc` or `gcc` compiler installed. On OS X `cc` comes with the Apple Developer Tools, which you probably have if you bought OS X 10.1. If you don't have them, you can go to the Apple Developer site, become a member (for free), and download the developer tools. If you are using a Solaris machine or another Unix system (such as GNU/Linux), then either `cc` or `gcc` should already be installed.

Of course, you should also have a current version of R. The current version as of this writing is 1.5.1.

## 3 Writing the C Code

If we want to interface C code with R, the functions we write in C need to have a few important properties:

1. C functions called by R must all return `void`, which means they need to return the results of the computation in their arguments.

2. All arguments passed to the C function are passed by reference, which means we pass a pointer to a number or array. One must be careful to correctly dereference the pointers in the C code. Sloppy handling of pointers can be a source of many nasty (and hard to trace) bugs.

3. Each file containing C code to be called by R should include the `R.h` header file. The top of the file should contain the line

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

   If you are using special functions (e.g. distribution functions), you need to include the `Rmath.h` header file.

When compiling your C code, you use R to do the compilation rather than call the C compiler directly. This makes life much easier since R already knows where all of the necessary header files and libraries are located. If you have written C code in a file called `foo.c`, then you can compile that code on the command line of your Terminal window with the command

```
R CMD SHLIB foo.c
```

This command produces a file called `foo.so`, which can be dynamically loaded into R (more on that later). If you do not want to name your library `foo.so`, then you can do the following:

```
R CMD SHLIB -o newname.so foo.c
```

The file extension `.so` is not necessary but it is something of a Unix tradition.

Once you have compiled your C code you need to launch R and load the library. In R, loading external C code is done with the `dyn.load` function. If I have compiled C code from a file `foo.c` into a file `foo.so` then I can load that code using

```
> dyn.load(‘‘foo.so’’)
```

Now all of the functions that you wrote in the file `foo.c` are available for R to call.

## 4 Your First Program

The first program everyone writes is the “Hello, world!” program. This might well be the only program that has been written in every computer language on the planet. The program, when executed, simply prints out the string “Hello, world!” to the screen. Since that’s a little boring, we will modify the standard version slightly so that it takes an argument `n` indicating the number of times to print the string “Hello, world!”. This program is only slightly more annoying. The pure R version of this program is:

```
hello1 <- function(n) {
  for(i in 1:n) {
    cat("Hello, world!\n")
  }
}
```

Simple. Now what if we wanted to run the `for` loop in C code rather than use pure R? We can do this by writing the relevant C code and calling it from R using `.C`.

First, one must create a separate file containing the C code which will do the work. We will call that file `hello.c`. The file `hello.c` might look something like this:

```
#include <R.h>

void hello(int *n)
{
  int i;

  for(i=0; i < *n; i++) {
    Rprintf("Hello, world!\n");
  }
}
```

As you can see, we have simply taken the loop in R and translated it into C. Notice that there is no `main` function. Since we are not really writing a C *program*, just a C *function*, there is no need for a separate `main` function. The function `Rprintf` is exactly like the standard `printf` function in C except that `Rprintf` sends its output to the R console so that you can see it when running R.

The corresponding R program would be as follows:

```
hello2 <- function(n) {
  .C("hello", as.integer(n))
}
```

The first argument to `.C` is a quoted string containing the name of the C function to be called. Recall that in the file `hello.c` we named our C function `hello`. Therefore, the first argument to `.C` in this case should be `‘‘hello’’`. The rest of the arguments to `.C` are arguments that need to be passed to the C function. The C function only has one argument, the number of times to print “Hello, world!”, so we pass in the variable `n` (after coercing it to an integer). The type to which you coerce the variables passed to the C function and the types in the prototype of your C function should match (as well as the order of the variables passed). Notice that we coerce `n` to integer type using `as.integer` and in the C function we have set `n` to be of type `int *` (remember that variables are always passed as pointers when using `.C`).

### 4.1 Running the `hello2` Program

The first thing you must do is write the code! We put the R code in a file called `hello.R` and the C code in a file called `hello.c`. Having done that, we must then compile the C code. At the command line (in your Terminal window), we can type

```
R CMD SHLIB hello.c
```

Running this command will produce a file called `hello.so`. Now, startup R. In R we type

```
> source(‘‘hello.R’’)
> dyn.load(‘‘hello.so’’)
> hello2(5)
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
[[1]]
[1] 5
```

Wait! What are those numbers and brackets at the bottom? They are the return values for `.C`. `.C` returns a list containing the (possibly modified) arguments which were passed into your C function. In this `.C` call, we passed in an integer with the value 5. Therefore, `.C` returns a list containing the number 5. If we had passed in more arguments, then `.C` would return a longer list. For this program we don’t care about the return value so we ignore it. But in most practical programs, we will need some elements of the return value.

## 5 More Examples

In this section we present more (statistically relevant) examples of calling C code from R.

## 5.1 Kernel Density Estimator

We will implement a simple kernel density estimator in this section. Given iid data $x_1, \ldots, x_n$ from an unknown density, we estimate the density at a point $x$ with

$$\hat{f}(x) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

where $K$ is a kernel function which is symmetric, positive, and integrates to 1. For this example we will use

$$K(z) = \frac{1}{\sqrt{2\pi}} e^{-z/2},$$

i.e. the normal density.

The naive way to implement this in pure R code would be something like the following:

```
ksmooth1 <- function(x, xpts, h) {
  dens <- double(length(xpts))
  n <- length(x)

  for(i in 1:length(xpts)) {
    ksum <- 0

    for(j in 1:length(x)) {
      d <- xpts[i] - x[j]
      ksum <- ksum + dnorm(d / h)
    }
    dens[i] <- ksum / (n * h)
  }
  dens
}
```

Here, `x` is the original data, `xpts` is a vector of points at which we want to evaluate the density, and `h` is the bandwidth parameter for the kernel function. While this function does work, it can be made much more compact with the following:

```
ksmooth2 <- function(x, xpts, h) {
  n <- length(x)
  D <- outer(x, xpts, "-")
  K <- dnorm(D / h)
  dens <- colSums(K) / (h * n)
}
```

Unfortunately, `ksmooth2` is not really much faster than `ksmooth1` because the `outer` function also has quite a bit of overhead.

What we would like to do is implement something like `ksmooth1` but with the double `for` loop implemented in C rather than in R. In R we would like to have something like

```
ksmooth3 <- function(x, xpts, h) {
    n <- length(x)
    nxpts <- length(xpts)
    dens <- .C("kernel_smooth", as.double(x), as.integer(n),
                as.double(xpts), as.integer(nxpts), as.double(h),
                result = double(length(xpts)))
    dens[[‘‘result’’]]
}
```

The `.C` function calls a C function called `kernel_smooth` which essentially implements the double `for` loop. Before we get to the C code, notice that here we store the return value of `.C`, which is a list containing all of the arguments we passed to `kernel_smooth`. When calling `.C` the last argument was given a name — the name here is `result`. The reason this argument was given a name is because it represents the vector of values of the estimated density function, i.e. the numbers that we eventually want to output. We will have to extract this vector after `.C` is finished so we give it a name. Since `dens` is just a list, we can extract the `result` element by subsetting the list using the given name. That is what happens in the last line of the function.

The corresponding C function is

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

void kernel_smooth(double *x, int *n, double *xpts, int *nxpts,
                   double *h, double *result)
{
  int i, j;
  double d, ksum;

  for(i=0; i < *nxpts; i++) {
    ksum = 0;

    for(j=0; j < *n; j++) {
      d = xpts[i] - x[j];
      ksum += dnorm(d / *h, 0, 1, 0);
    }
    result[i] = ksum / ((*n) * (*h));
  }
}
```

Note that we had to include the `Rmath.h` header file so that we could use the `dnorm` function. The special functions which can be used in C code are all documented in the PDF manual *Writing R Extensions*.

## 5.2 Convolutions

The following C function computes the inner product of a vector $x$ with a lagged version of itself. Arguments are the lag, the vector, its length, and the result. Such a function is useful in time series analysis to compute convolutions and autocorrelations.

```
void cconv(int *l, double *x, int *n, double *s)
{
  double  *y = x + (*n - *l), *z = x + *l, *u = x;
  while ( u < y)
    *s += *u++ * *z++;
}
```

Suppose this function sits in a file `cconv.c`. We compile the C code in such a way that it can be loaded into R. The easiest way to do this is to use R. Just go to your terminal window, move to the directory which contains `cconv.c`, and say

```
R CMD SHLIB cconv.c
```

You will see

```
gcc -I/usr/local/lib/R/include  -I/sw/include   \
   -fPIC  -g -O2 -c cconv.c -o cconv.o
gcc -bundle -flat_namespace -undefined suppress \
   -L/sw/lib -o cconv.so cconv.o  -L/usr/local/lib
```

Of course you could also enter these commands yourself.

Now there are two things left to do. First you load the file into R. This is simply a matter of opening R, from the directory containing the newly created `cconv.so`, and saying

```
> dyn.load("cconv.so")
```

The second is to write a convenient wrapper function in R that calls the loaded code with the appropriate arguments. For this we use the `.C` function in R, which does most of the dirty work. Our R function `rconv` is

```
rconv <- function(lag,x) {
    .C("cconv",
       as.integer(lag),
       as.double(x),
       as.integer(length(x)),
       as.double(0.0))[[4]]
}
```

Observe that we convert all arguments to the approriate type (just to be sure). The `.C` function returns a list with all its arguments, we only need the last (fourth) argument. In this case we did not name the last argument but extracted it using a numeric index. Whether one uses named arguments (as in Section 5.1) or numeric indices is a matter of personal preference.

Thus, as an example,

```
> x <- rnorm(100)
> rconv(0, x)
[1] 108.0924
> rconv(1, x)
[1] 1.832316
```

With a little bit more trouble we can vectorize this, i.e. make it accept a vector of lags and return a vector of convolutions of the same length. The C code now becomes

```
void cconv (int *l, int *m, double *x, int *n, double *s)
{
  double *y, *z, *u;
  int i;
  for (i = 0; i < *m; i++)
    {
      y = x + (*n - l[i]);
      z = x + l[i];
      u = x;
      s[i] = 0.0;
      while (u < y)
        s[i] += *u++ * *z++;
    }
}
```

and the R wrapper function is written as

```
rconv <- function(lag,x) {
    .C("cconv",as.integer(lag),
       as.integer(length(lag)),
       as.double(x),
       as.integer(length(x)),
       as.double(vector("double",length(lag))))[[5]]
}
```

The example now is

```
> l <- c(0,1,2,10)
> x <- rnorm(100)
> rconv(1, x)
[1] 108.092417   1.832316   3.694085 -29.567533
> rconv(seq(10), x)
 [1]   1.832316   3.694085 -11.332405   3.703928  -1.572233   2.954725
 [7]   1.942543  -7.963158  -2.347786 -29.567533
```

## 6 A Note on Manipulating Matrices in C

Very often in statistics we deal with matrices. However, in C there is no matrix data type. Therefore, manipulating matrices in C can be cumbersome and confusing. One should generally try to deal with matrices using pure R code; R has many matrix manipulation routines that are highly optimized. Standard operations such as eigenvalue, singular value, QR, or Cholesky decompositions should *not* be done in user-written C code. However, there may still come a time when you need to pass a matrix to some specialized C code. In that case, it is important to remember that matrices are represented as just very long vectors (of length $nrows \times ncols$) in C. Extracting particular elements of the matrix will require careful handling of indices.