How to link R with C

15th February 2005

Why link R with C

- Access to compiled routines already written in C (or Fortran). No need to reinvent the wheel!
- Speed
  - For loops in R can slow down your program dramatically
  - “Apply-type” functions in R (e.g. apply, lapply, tapply…) are possible ways to circumvent using For loops. However, not always possible to avoid them.
  - Writing the slow parts of an R program in C (even inefficiently) can significantly improve the speed / performance of your program
- I have used this feature for writing EM type programs

Standard Example

- Convolution of two finite sequences:

$$c_i = \sum_{j,k\ge 0:\, j+k=i} a_j b_k, \qquad\qquad i = 0,\ldots,n_a + n_b$$

- In R,

```
convolveR <- function(a,b){
  na <- length(a); nb <- length(b)
  ab <- rep(0,na+nb-1)
  for (i in 1:na){
    for(j in 1:nb){
       ab[i+j-1] <- ab[i+j-1] + a[i]*b[j]
     }
  }
  return(ab)
}
```

- In C,

```
void convolve(double *a, int *na, double *b, int *nb, double *ab)
{
 int i, j, nab = *na +*nb – 1;

 for(i=0; i < nab; i++)      /* Note that the indexing starts at 0 */
   ab[i] = 0.0;
   for(i=0; i < *na; i++)
     for(j=0; j < *nb; j++)
       ab[i+j] += a[i] *  b[j];
}
```

- We can then call this C function within R, using **.C**

```
convolveRC <- function(a,b)
   .C(“convolve”, as.double(a), as.integer(length(a)), as.double(b),
       as.integer(length(b)), ab = double(length(a) + length(b) - 1))$ab
```

- To illustrate, let

```
u <- seq(1,10000,length=1000)
v <- seq(2,20000,length=2500)
```

- **system.time(convolveR(u,v))** - 1.5mins
- **system.time(convolveRC(u,v))** - 0.04 sec
- Note that the C program needs to be compiled and then loaded before it can be called by R

The R interface function .C

- Provides a standard interface to compiled code that has been linked into R either at build time or via the R function **dyn.load**

- The first argument to **.C** is a character string giving the symbol name/routine name (e.g. “convolve”)

- The next set of arguments are the R data types (usually vectors) that need to be passed to the compiled C code

- The storage mode of these R data must match up directly with the C function arguments types and have the correct length

R storage mode		C type
logical	→	**int ***
integer	→	**int ***
double	→	**double ***
complex	→	**Rcomplex ****
character	→	**char ****

- The arguments passed to the compiled routine may be given “name” fields. These do not match anything in the C routine itself, but will be retained as the name fields in the results
- The arguments for the C program must be pointers.
- A pointer is a variable whose value is the address of an object in memory

- None of the supplied R data types to the C program can have NAs, unless a further argument **NAOK=TRUE** is supplied to the **.C** interface function

- Additional arguments to **.C** that may be used, but must come after those arguments that match the compiled routine, are **NAOK**, **DUP** and **PACKAGE**

- Note that the compiled C code should not return anything except through its arguments. C functions should therefore be of type void

Dynamically loading the compiled code

- Compiled code to be used with R is loaded as a shared object (in UNIX) or a DLL (in Windows)

- The shared object/DLL is loaded (outside of an R package) with **dyn.load( )** (and unloaded using **dyn.unload( )**). For example,

dyn.load("convolveC.so")

- The first argument of **dyn.load( )** is a character string giving the pathname to a shared library or DLL

- Programmers should probably avoid assuming a specific file extension for the object/DLL (such as “.so”) but use instead a construction like

```
file.path(paste(“convolveC”,.Platform$dynlib.ext,sep =“”))
```

for platform independence

- Loading is often done within an R package via a call to **library.dynam** in the **.First.lib** function, found in a file placed within the R subdirectory of the package. For example,

```
.First.lib <- function(lib, pkg) library.dynam(“libname”, pkg, lib)
```

where libname is the object/DLL name with the *extension omitted*

- Use **PACKAGE=“libname”** at the end of the **.C(…)** call to confine the search for the symbol/routine name to a specific shared object/DLL

- The shared library is loaded when **library(libname)** is executed

Compiling

- If creating an R package, then the code is compiled (and the shared library is built) when the package is installed using

```
R CMD INSTALL
```

- If working from outside a package then we can create a shared library by using **R CMD SHLIB** . For example, type at the UNIX prompt

```
R CMD SHLIB convolveC.c
```

to create **convolveC.so**

- **R CMD SHLIB** is just a way of calling **gcc** with appropriate options

Access to R's C libraries

- A number of the inbuilt R functions (that come with the base package) use C routines. For example, functions for random number generation, routines to calculate densities, cumulative distribution functions and quantile functions for the standard statistical distributions and optimization routines

- These C routines can also be used within your C program

- See the header files (e.g. **Rmath.h**) found in **/usr/local/lib/R/include**

- To access these routines put, for example, an include statement at the beginning of your C program

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

- Then compile using **R CMD SHLIB** to link these libraries

Suggestions

- Read the manual Writing R Extensions for more details

- For example, if you want to find out more about creating your own R package

- Or if you want to learn how to handle R objects directly in C. That is, if you want to call R functions from within your C code. (See the **.Call** or **.External** functions)

- Get your program working fully and as efficiently as possible in R before deciding to write some of it in C

- When writing your C program(s), I would suggest thinking about creating wrapper and header files

- Finally read other people's source code to learn more.

References

- Writing R Extensions Manual - R Development Core Team
- S Programming - Venables and Ripley (2000). Springer
- The C Programming Language (2nd Edition) - Kernighan and Ritchie (1988). Prentice Hall