5 Extending R: High performance with compiled code

Sometimes the only way to make your code run faster is to rewrite it in a lower-level language. In fact many high-level languages, like R, have modules written in C or Fortran for greater speed (e.g., R's vectorised functions). Programs written in lower-level languages sometimes require more development time and tend to be more complex to code and debug but when performance bottlenecks are caused by language limitations (Fig. 1.3), refactoring may be the only option. In the final sections of this document, we will show you how to make C do the heavy lifting by extending R with C. This can be a complex operation for those not too familiar with programming. However, even those with little programming experience in C should be able to follow the simple examples outlined here. We do however advise to exapand your experience with C a bit before trying this on your own examples.

In this section we provide what is hopefully a user-friendly guide to writing R-extensions in C [7]. Before we start with another ecological example, we will run through the setup of the system and extend the simple example we used in section 1.3 to C. Readers should take care because calling compiled code from R can be rough and small errors in your C code may cause R to crash (e.g. segfaults - or memory access violations - are common). So, do this at your own risk, and make sure to save crucial analyses and data before you start. We refer to the "Writing R Extensions" manual for detailed information [8]. All the examples given below were tested on Windows, Mac and Linux machines.

5.1 Getting started with extending R

5.1.1 Getting started under windows

Unfortunately windows is not the most friendly environment for extending your R code with C, and you will have to take a few extra steps to get things done compared with your *UNIX based brethren. Luckily it is not difficult, as the good people developing R have bundled all the tools you need in one zip-file "Rtools", which you will need to download [9] and install [10]. This installation contains all the tools you need to build packages and compile C code for use in R. Once you have downloaded and installed the *Rtools*, you will need to take the following steps to change your environment variables. More detailed help

[7] At the end of this tutorial some readers may wish to try Rccp (Eddelbuettel & François, 2011), which is an R-package that attempts to ease the integration of R and C. As we show, *Rccp* is in no way required to write your own extensions, but for those who feel uneasy with writing C extensions or require easy transfer of data between R and C, the *Rccp* package will provide help.

[8] `http://stuff.mit.edu/afs/sipb/project/r-project/arch/i386_rhel3/lib/R/doc/manual/R-exts.html`

[9] `http://cran.r-project.org/bin/windows/Rtools/`

[10] the documentation for these tools can be found on this webpage `http://cran.r-project.org/doc/manuals/R-admin.html#The-Windows-toolset`

can be found in the R FAQ [11].

1. Change your PATH variables properly. To do this in the newer versions of Windows (e.g. Vista, Windows 7) go to the 'Control Panel', then click on 'User Accounts' and select 'Change my environmental variables' in the left panel [12].

2. Now we create a new environment variable with 'Variable name' PATH. And set as 'Variable value' the following paths '...\Rtools\bin;...\Rtools\gcc-4.6.3\bin;...\R-3.1.0\bin', where ... refers to the directories on your machine where *Rtools* and *R* is located, respectively. These paths correspond to the versions of *Rtools* (31) and *R* (3.1.0) on our Windows test machine.

3. Restart your computer.

5.2 Writing C code

Providing a comprehensive guide on coding in C is far beyond the scope of this tutorial. Luckily there are many online resources on C and C++ freely available. If you are a beginner at C, these guides will help you write simple C and C++ code within only a few hours of self study. This is fortunate, because, as we will show below, often only very simple programs are needed to speed-up code considerably. In this tutorial, we will give some basic pointers that should help to get you started with C code meant for use in R. We will start, step by step, by rewriting a simplified version of the R code in section 1.3. We use this code as a means to illustrate the steps you need to take to "load a dynamic library" to call C functions from R. It is not an example of typical program that could benefit from calling C from R (in constrast to the program in section 6).

The code in R looks like this:

R Example 5.1.

```
## OnlineRcode57.R

SimpleR<-function(N){
  answer<-numeric(N)
  for (i in 1:N) { answer[i]<-(N/(1+N))}
  return(answer)
}
```

[11] http://cran.r-project.org/bin/windows/base/rw-FAQ.html#How-do-I-set-environment-variables_003f

[12] For other versions of Windows see http://cran.r-project.org/bin/windows/base/rw-FAQ.html#How-do-I-set-environment-variables_003f

The same function in C looks like this:

R Example 5.2.

```
## OnlineRcode58.R

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

/* Rewrite of  InterpreterQuirks.R */
void SimpleC(int *nc, double *dnc, double *answerc) {

  int n = nc[0];
  double dn = dnc[0];
  int i;

  for (i=0; i<n; i++){
    answerc[i] = (dn/(1+dn));
  }
}
```

Before we can time the execution of the program in C we need to compile it, load it into R and create a wrapper function. However, before we do so, perhaps we should provide a very quick run through of the meaning of the different code elements in C, compared to how we code in R. Those familiar with C, and don't feel the need for a short refresher, can skip the next part.

5.2.1 C CODING IN A NUTSHELL

C code is surely different from R, but even with no C experience one should be able to comprehend exactly what is going on. Therefore, let's have a look at the individual elements of the program *SimpleC* and highlight the differences with R. In the first line we find a "preprocessor" directive, the "#include" part. This tells the compiler that we want to use functions from the "header files" called R.h and Rmath.h in our program. Header files are libraries that contain functions. When coding in C many function are not available by default as in R, but we can gain access to many C functions by including correct header files in the first lines of our code. Think of it as loading a package in R. Note that we don't need these header files in this example (it will run perfectly well without it). We just added it as an example to show how you can include functions, even a bunch of functions you know and love from R, by loading pre-made libraries in your C functions.

Next we see sections of code between "/*" and "*/", we can use these as we use # in R code, to comment sections for our convenience. The next important line is where we start our C function, with "void". This line tells the compiler

that there is a function named *SimpleC*, with an integer input "*int *nc*" and double (i.e. floating point) inputs "*double *answerc*" and "*double dnc*", and that the function should not return anything ("void"). Usually in C we would start with something like a main function, for instance the command "int main()" would make a function called main that returns an integer. However, R requires that C functions start with *void*, as they should not return anything [13]. The operations work by sending an empty vector (e.g. full of zeroes) from R to *SimpleC* called $*answer$ which is modified by *SimpleC*. We then access it in R in its modified state. It is called "$*answer$" (not "answer") because the "*" specifies that it is a pointer, which specifies the location of an R object in the memory [14].

We also see that, just as in R, our functions are encapsulated by "curly braces" ({ and }) which signal the start and end of functions or "if" statements. Inside the "curly braces", we see that as in R, input and data are stored in variables, however, in C one must declare the specific type of each variable before it is used. As seen above for instance, we create two integers variables with "*intn*" and "*inti*", where we explicitly declare not only the name, but also the type of data that it will contain (integers, or whole numbers without decimals). Some basic types include *char*, *int*, $float$ and *double* (*char* for characters, *int* for integers and $float$ and *double* for numbers with decimals). We then continue to assign values from the pointers N and dN, and this is because we need real values for n and N for calculation. Remember that a pointer only tells us where an R object is in the memory, not its value (and we can't really calculate with memory locations of objects). Also, one should always double check the math in C code, so that we are sure that we are not dividing a double by an integer (which is why we created *double dnc* in addition to int *nc* in the above example).

One other thing you will notice is that the indexing of vectors starts at 0, so the first element of a vector V is not found by the command V[1] but with the command V[0]. In C and C++ vectors always start with 0, this is actually customary in computer science and many other programming languages (with R being the odd one in this sense). Next we see that for loops are initiated in a slightly different fashion with "$for(i = 0; i < n; i++)$". This specifies that the loop starts its iteration with i = 0 (corresponding to the first element of the vector), stops when i<n and is incremented at each iteration with 1 (i++). Finally, we end each line of code within a C program with a semi-colon ";".

5.3 Compiling C code and creating shared libraries

Now that we have some code written in C, we need to compile it and create a "shared library" from which it can be dynamically loaded into R. To do this

[13] This is the case when we use C and *.C* to call our function, as you will see later in a slightly more complex version: we can manipulate R objects and return them if we want to.

[14] Pointers literally "point" to locations in memory. So the vectors are first created in R, and then instead of copying every element of a potentially very large vector, we simply pass its location to the C function and get proceed with our calculations.

we will need the standard compilers and libraries for our operating system [15]. Windows users will have acquired them in section 5.1.1. Mac users should download the Xcode application followed by the Command Line Tools package. Linux users should acquire the correct packages for doing so. On Ubuntu the correct packages can be installed with:

```
sudo apt-get update
sudo apt-get install build-essential r-base-dev
```

In many cases these will be already installed on a *UNIX system, so this should work without extra effort. Once you are certain all tools are ready, we can start. One advantageous way to code in C for usage in R, is to do so "inline" within your R scripts. This way everything is in one place that needs to be. One way to code C in R is through using the *sink* and *cat* commands. In this way we use R to create the files.

R Example 5.3.

```
## OnlineRcode59.R

## Open a connection to a file on the system and in
## the working directory
sink("Simple.c" )

## Send C code to this file with "cat"
cat("
/* Rewrite of  InterpreterQuirks.R */
void SimpleC(int *nc, double *dnc, double *answerc){

 int n = nc[0];
 double dn = dnc[0];
 int i;

    for (i=0; i<n; i++){
      answerc[i] = (dn/(1+dn));
    }
}
")

#stop writing Simple.c
sink(NULL)
```

[15] We need a compiler as it will translate the programs we write into an executable that the computer can "understand" and execute.

If you copy and paste the above, you should have a file in your working directory called "*Simple.c*" that has the above c-code as its contents. You can use the command *file.show*("*Simple.c*") in R to confirm this. Now that we have this stored as a file, let us compile it and create a shared object that we can load in R. Again we can do this from R if we want to (or you can use any command line tools you have [16]. If you are working from the command line, which may be the easiest approach if you are a Mac user, you can create a shared library for R by typing (in the directory you saved *Simple.c*) "R CMD SHLIB -o Simple.so Simple.c". This commands calls R to create shared objects (for loading into R). It accepts as arguments a list of files with extensions .o .c, .cpp, and .f (which are object files, C, C++, or FORTRAN sources, respectively). For more details see the R documentation for writing R extensions [17]. We can do the same from R by invoking the system command (the following code is for Unix and Mac users further in the document).

R Example 5.4.

```
## OnlineRcode60.R

system(R CMD SHLIB -o Simple.so Simple.c" )
```

When using Windows you should change the ".so" into ".dll" or you may receive a warning:

R Example 5.5.

```
## OnlineRcode61.R

system(R CMD SHLIB -o Simple.dll Simple.c" )
```

Now that we have compiled *Simple.c* into a shared object (*Simple.so*) or a "dynamic-link library" (Microsoft's implementation of shared objects: *Simple.dll*), which can be dynamically loaded in R, all we need to do before using it is to create a wrapper function in R. Crucial components of the wrapper function are 1) the R function that calls compiled C code *.C*() and 2) the *dyn.load*() function which loads shared objects. A typical wrapper function (in Unix/Mac) will look like this:

[16] These tools are included in the *Rtools* download for Windows see 5.1.1

[17] `http://stuff.mit.edu/afs/sipb/project/r-project/arch/i386_rhel3/lib/R/doc/manual/R-exts.html`

R Example 5.6.

```
## OnlineRcode62.R


## change to dyn.load("Simple.dll") when on windows
dyn.load("Simple.so" )

SimpleCWrapper <- function(N){

       out <- .C("SimpleC" ,
           nc     = as.integer(N),
           dnc    = as.double(N),
           answer = as.double(rep(0,N))
           )

       return(out$answer)
}
```

Within the function *.C* we call our C function *SimpleC*, followed by all the commands which need to be passed to it. However, Windows users will have created a *Simple.dll* file instead of a *Simple.so* file, so the wrapper function for Windows is:

R Example 5.7.

```
## OnlineRcode63.R

dyn.load("Simple.dll" )

SimpleCWrapper <- function(N){

       out <- .C("SimpleC" ,
           nc     = as.integer(N),
           dnc    = as.double(N),
           answer = as.double(rep(0,N))
           )

       return(out$answer)
}
```

One thing that is very important is that we ensure that the variables we send to the compiled code correspond exactly to what is expected (e.g. types

and names), or the code may crash. The final bit of the wrapper organizes the output of the function and returning what we are interested in. If you change your C code, always make sure to use *dyn.unload* to remove it. That's it! Now that we have a wrapper function, we are ready to test it:

R Example 5.8.

```
## OnlineRcode64.R

system.time(Canswers<-SimpleCWrapper(1e6))

##    user  system elapsed
##   0.000   0.004   0.007

system.time(Ranswers<-SimpleR(1e6))

##    user  system elapsed
##   1.332   0.000   1.335

identical(Canswers,Ranswers)

## [1] TRUE
```

We see that we have succesfully called a C function from R that gives us identical results to our R-version. Now that we have all the tools to extend R, we can start with a more serious example.