Writing R extensions (.C/.Call)

R (brilliant though it is) has one flaw: `for` loops are E X T R E M E L Y slow.

Things can be sped up by outsourcing costly loops and leaving them to C, as C is really fast.

Excellent idea, the basic problem is just HOW to do this – especially when you are using some Windows PC.

Information on this is really hard to obtain and most information provided is not too user-friendly. While I was getting more and more confused by lots of wild and scary instructions, I thought it might be a good idea to write down the way to go, if I finally managed to write these extensions. Well, that's where I am now.

Thanks to all who provided any useful information on the internet on how to do this R extensions stuff and helped me along with it.

(If you have any comments, don't hesitate to contact me: `franziska.lindner@kit.edu`.)

Franziska Lindner 26/3/2011

CHAPTER 1

Getting windows ready for use

Getting any information on this issue is really hard, because everyone seems to either know how to do it or uses UNIX-based systems. The one and only helpful source I found is

`http://mcglinn.web.unc.edu/r-code/linking-c-with-r-in-windows/#setup`

In a nutshell:

The steps you have to take are

1. Download the file `Rtools.exe`

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

   which contains all the stuff you need to build packages and compile C code using the command line.

2. Change the path of environment variables.

   (German version of Windows 7)

   [Start] → [Systemsteuerung] → [Benutzerkonten] → [Eigene Umgebungsvariablen ändern]

   [Neu]
   ↓
   [Name der Variablen:]
   PATH

   [Wert der Variablen:]
   `c:\Rtools\bin;c:\Rtools\perl\bin;c:\Rtools\MinGW\bin;c:\Programme\R\R-2.12.1\bin`

   Make sure you use the right paths to Rtools and to R (and put the version of R you intend to use).

   > Note:
   > If you already have defined (might be deliberately or by accident) some path variable (this happens for example when installing other programs) with – let's say – the value `c:\Programme\...` (the dots just indicating some program's name), you might either erase the value and put

`c:\Rtools\bin;c:\Rtools\perl\bin;c:\Rtools\MinGW\bin;c:\Programme\R\R-2.12.1\bin`
instead or add the needed path value right in front of the existing one (separated by a semicolon):
`c:\Rtools\bin;c:\Rtools\perl\bin; c:\Rtools\MinGW\bin;`
`c:\Programme\R\R-2.12.1\bin;c:\Programme\...`

(Thank you, Dan, for this remark!)

3. RESTART your computer.

CHAPTER 2

Writing R Code

The exemplary function `foo` we are using looks like this:

```
foo<-function(N,a,b)
     {
         c<-numeric(N)

         for(i in 1:N)
         {
         for(j in 1:N)
            {
            c[i]=a[i]*b[j]
            }
         }
     return(c)
     }
```

We can now call the function (with exemplary parameters)

```
foo(N=4,a=c(1,2,3,4),b=c(5,6,7,8))
```

which yields

```
[1]  8 16 24 32 .
```

CHAPTER 3

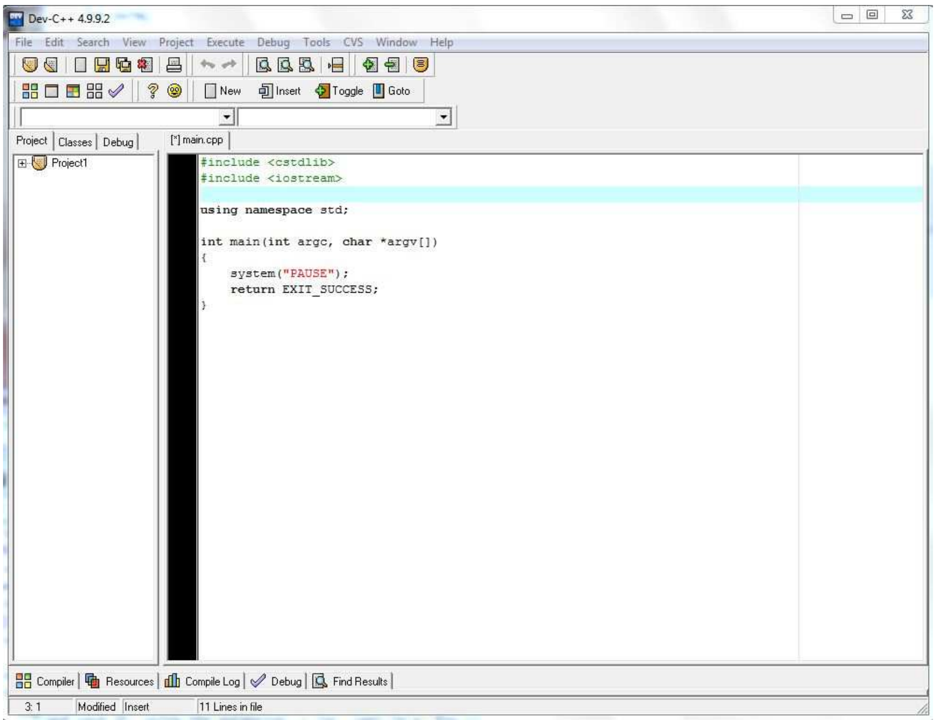
Writing C Code

1. Choose a C compiler (as you want to make sure that the C Code works, before you start loading it into R).

   Usually, windows users tend not to have a preinstalled C/C++ compiler.
   I recommend Dev C++ (which is freeware). You can obtain this from

   `http://www.bloodshed.net/index.html`

2. Start Dev C++. Starting a new project (choose 'Console Application'), yields the following:

   This source file (`main.cpp`) is where we insert our code.

3. Include the headers `<iostream>`, `<cstdlib>` (In our case, these two are already there!) and `<math.h>`. Those three headers should be sufficient for most (crude) computations.

   You include headers by typing

```
#include <iostream>
#include <cstdlib>
#include <math.h>
```

4. Now the most important part: function `foo`. We put the corresponding code in between `using namespace std;` and `int main(int argc, char *argv[])` .

   Important to notice:

   The function must be of type `void`, which means that it doesn't return anything. This is the reason why we have to add an additional argument. Our function in R returns some vector c. The correspnding function in C can't return anything, because it's of type `void`. Therefore, we have to pass an arbitrary vector c to our function. The function `foo` then modifies the vector and after having run the function, the components of c have changed and we can access the results of `foo` by calling vector c.

   Arguments passed to the C function have to be pointer types (that's just the way R passes the arguments down to the C function – if you call the function in R using a double variable, the C function is called using a double pointer.

```
void foo(int *N, double *A, double *B, double *C)
    {
    ...
    }
```

   As we don't want to use the address of the argument N in our function, but the actual value of N (as N is the length of the vectors a and b!), we have to call – when we are referring to the value of N – `*N` instead of just `N`.
   To save us from remembering this every time we want to use N, we rename `*N` and call it `n`.

   <u>Remember:</u> Each time you use a new variable in C, you have to say of which type it ought to be, which means that we now have to write `int n=*N;`, instead of `n=*N;`.

```
void foo(int *N, double *A, double *B, double *C)
    {
    int n=*N;
    ...
    }
```

   Next thing to add to our function would be the `for` loops. Loops in C start with 0 and we thus get

```
void foo(int *N, double *A, double *B, double *C)
    {
    int n=*N;
    for(int i=0;i<n;i++)
```

```
      {
      for(int j=0;j<n;j++)
         {
         C[i]=A[i]*B[j];
         }
      }
   }
```

5. Check the C Code for mistakes

   In order to do this, we need a tiny main programm, which calls our function and prints the results.

   If you check the compiler window, you are already seeing some code looking like this

```
int main(int argc, char *argv[])
  {
    system("PAUSE");
  return EXIT_SUCCESS;
  }
```

   That's the framework of the main program we are going to use.

   First thing we need to add is the declaration (i.e. specification of the type) of the pointers `N,a,b and c` (remember, we had to use pointers as arguments of the C function, as R passes down pointers to the C function...).

   We then define (i.e. specify the values of) `N,a and b` (`c` is arbitrary, thus we don't have to specify the exact values.)

```
int main(int argc, char *argv[])
  {
  int *N=new int[1];
  double *a=new double[4];
  double *b=new double[4];
  double *c=new double[4];

  //Remember that indexing in C starts with 0!!
  N[0]=4;

  a[0]=1;
  a[1]=2;
  a[2]=3;
  a[3]=4;

  b[0]=5;
  b[1]=6;
  b[2]=7;
  b[3]=8;

  foo(N,a,b,c);

//To look at the vector c and to check whether foo works the way it should,
```

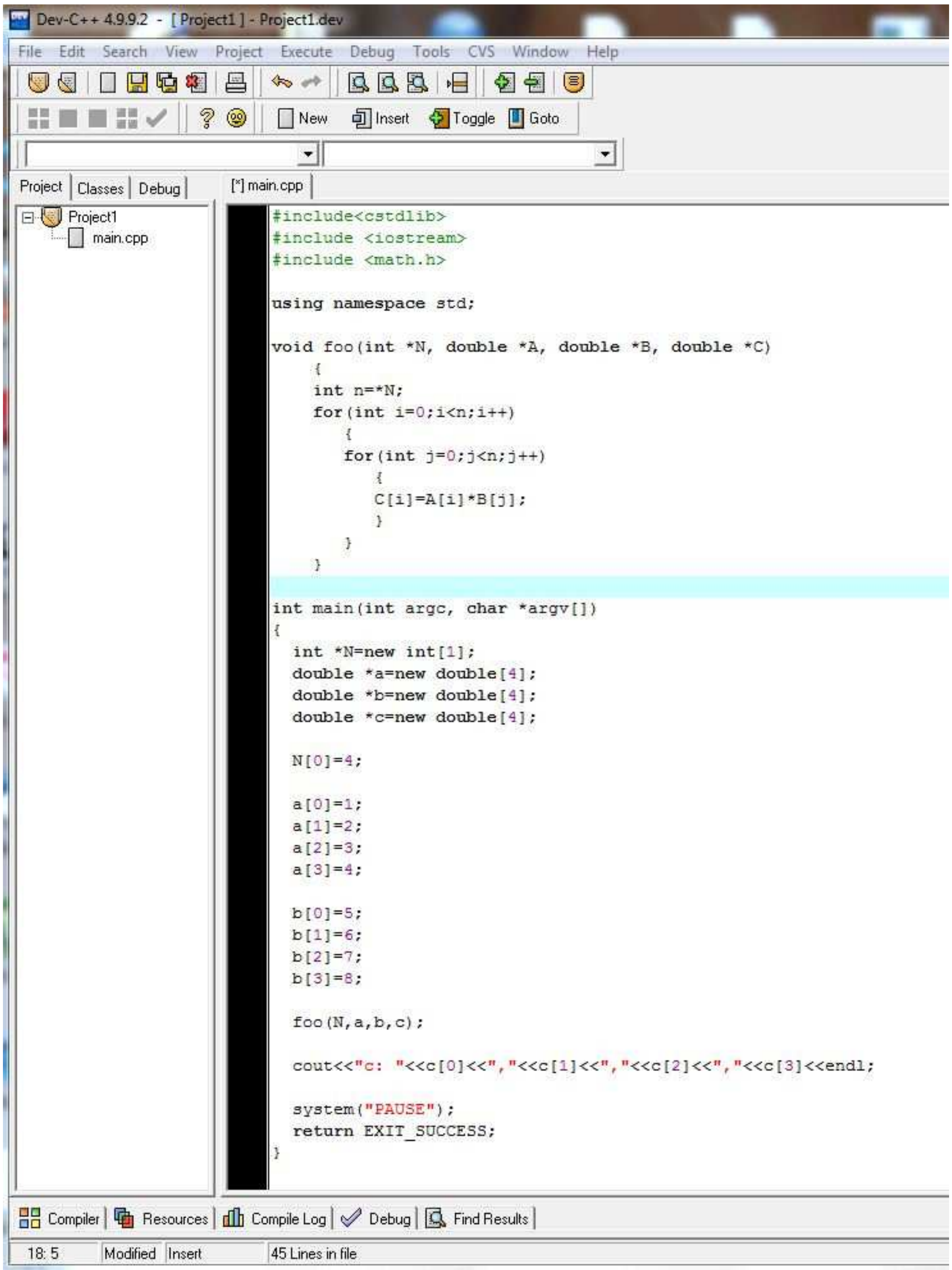
```
//we have to print the single components of c, as just cout<<c<<endl;
// would give us the address of c and cout<<*c<<endl;
//would only return the value of the first entry of c.

  cout<<"c: "<<c[0]<<","<<c[1]<<","<<c[2]<<","<<c[3]<<endl;

  //The following lines have already been there...
  system("PAUSE");
  return EXIT_SUCCESS;
  }
```

Your code and the output you get should now look like this:

CODE:



OUTPUT:

Really helpful is also the following link

`http://biostat.mc.vanderbilt.edu/wiki/Main/WritingRExtensions`

Make sure you have double-checked all divisions and multiplications, as some wrongly set types (`int` division instead of `double` division etc.) might hijack your results...

6. Having checked that your C Code is working, you now need to extract the important part, which is the function `foo` you have written.

   Copy and paste the `void` function `foo` (ONLY the void function) into an empty editor file, add – at least – the headers

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

   and save it, using the extension .c (or .cpp) (e.g. foo.c, or foo.cpp).

CHAPTER 4

Compiling C Code

1. Open the shell/console/prompt/command line (ger.: Eingabeaufforderung)
2. Using the commands cd../cd access the directory of R and then access the folder bin.

   In my case it has been like

3. Remember this path and make sure you transfer the file containing your C Code (foo.c) to exactly this directory (use the 'save as' option).
4. Back in the command line, we are still left with

   ```
   C:/Program Files/R/R-2.12.1/bin>
   ```

   Now you need to enter the command

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

   and press `Return`.

   This should return some lines of the form:

```
C:\Program Files\R\R-2.12.1\bin>R CMD SHLIB foo.c
cygwin warning:
  MS-DOS style path detected: C:/PROGRA~1/R/R-212~1.1/etc/i386/Makeconf
  Preferred POSIX equivalent is: /cygdrive/c/PROGRA~1/R/R-212~1.1/etc/i386/Makec
onf
  CYGWIN environment variable option "nodosfilewarning" turns off this warning.
  Consult the user's guide for more details about POSIX paths:
    http://cygwin.com/cygwin-ug-net/using.html#using-pathnames
gcc -I"C:/PROGRA~1/R/R-212~1.1/include"         -O3 -Wall  -std=gnu99 -c foo.c -
o foo.o
gcc -shared -s -static-libgcc -o foo.dll tmp.def foo.o -LC:/PROGRA~1/R/R-212~1.1
/bin/i386 -lR

C:\Program Files\R\R-2.12.1\bin>
```

5. Now, if you check the directory where you put your file foo.c (In my case, this would be the directory `C:/Program Files/R/R-2.12.1/bin`.), you should notice two other files, which appeared: foo.dll, as well as foo.o. The .dll file is the one we are interested in.

CHAPTER 5

Loading C Code

Use the command `dyn.load`, putting the file path and name in quotation marks:

```
dyn.load("C:/Program Files/R/R-2.12.1/bin/foo.dll")
```

Careful, each time you want to change your C Code be sure to unload the Code in R using dyn.unload:

```
dyn.unload("C:/Program Files/R/R-2.12.1/bin/foo.dll")
```

Change your C Code in the editor, compile it (cf. Chapter 4) and load the new .dll file once more using the command dyn.load:

```
dyn.load("C:/Program Files/R/R-2.12.1/bin/foo.dll")
```

CHAPTER 6

.C extensions

6.1 Running .C in R

Calling the C Code is done by

```
>exmpl<-.C("foo",as.integer(N), as.double(A),as.double(B), as.double(C))
```

Typing

```
>exmpl
```

now yields a list of objects. Why is that?
As the C function is of type `void`, nothing is returned. Returning works via the arguments of the function. Thus, the call of the C function returns a list containing

as a first entry, the value of N,
as a second entry, the vector A,
as a third entry, the vector B
and as a fourth entry, the vector C
– just in the order of the arguments passed to the function.

You may then access the entries of the list via

```
>exmpl[[1]]
```

to get N, or

```
>exmpl[[2]][2]
```

if you are interested in the second component of vector A.
In our case, we would like to extract C, which means we want

```
>exmpl[[4]].
```

6.2 N2N

Here's just some stuff I came across when trying to implement my .C extensions. Hopefully this advice might save you some time in case you experience the same difficulties:
If you intend to work with complex numbers, including the header `<complex.h>` is not of any use:

`http://blog.hokietux.net/?p=139`

There is, however, also the way to write your own complex class in C and include it. I have never tried this when calling .C functions in R, so I have no practical experience on what to be careful about when doing that.

(Some information on that can be found in 'Writing extensions in R' (p.76) `http://cran.r-project.org/doc/manuals/R-exts.pdf`)

The easiest way – which I chose and which works perfectly fine – is that you treat real and imaginary part of the complex numbers separately, which means that instead of calling `foo` using an argument of some complex type, you just construct and call `foo` with two (real-valued) arguments.

('Writing extensions in R' (p.69) mentions to pass an argument of (R-type) complex and treat this argument as being of (C-type) `Rcomplex*`. Having included the header `<R.h>`, this should apparently work, but I have not tried it yet.)

CHAPTER 7

.Call extensions

When using .C functions, R copies the arguments first, before passing them on to the function `foo`. After the usage of `foo`, the arguments are - again - copied and passed to R as a list object. However, when we are dealing with huge data frames, a good idea is to use .Call, as the arguments of the function in this case are only references and no actual copies.

7.1 Adjusting C Code

1. In your editor-file foo.c, include the header

   ```
   #include <Rinternals.h>
   ```

   (`<Rdefines.h>` is also a possibility, but we won't deal with this here, as different commands are necessary.)

2. Constrasting .C, the function we now need to use is not of type `void` (which meant that the function didn't actually return anything), but of type `SEXP` (which means that some object of type `SEXP` is returned, so be sure to remember to put the command 'return' in your code. If you don't want to return anything, use

   ```
    return R_NilValue
   ```

3. The arguments of your functions ought to also be of type `SEXP`. Which means that our function `foo`, formerly

   ```
   void foo(int *N, double *A, double *B, double *C)
       {
       ...
       }
   ```

   becomes

   ```
   SEXP foo(SEXP N, SEXP A, SEXP B, SEXP C)
       {
       ...
       return R_NilValue;
       }
   ```

Some basic Information on SEXP:

The following extract is taken from 'Writing extensions in R' (http://cran.r-project.org/doc/manuals/R-exts.pdf)

The R object types are represented by a C structure defined by a typedef `SEXPREC` in `Rinternals.h`. A variable of type `SEXP` is simply a pointer to a `SEXPREC`.
Some `SEXPREC`s in C are equivalent to R data type: The most important ones are

SEXPTYPE	R equivalent
REALSXP	numeric with storage mode double
INTSXP	integer
REALSXP	complex
LGLSXP	logical
STRSXP	character
NILSXP	NULL
STRSXP	character

What does this mean? If you pass an (R-type) `numeric` argument to the function, the C-function is called with an argument of (C-type) `REALSXP`.

More information on `SEXPTYPES`:

http://svn.r-project.org/R/trunk/src/include/Rinternals.h

If you need to change types (e.g. function argument `SEXP A` turned out to be of type `INTSXP` and want to change it to `REALSXP`), you use

```
PROTECT(A.new=coerceVector(A,INTSXP));
```

(Protection is necessary, because we create a new object `A.new`! See 4. for more information on the `PROTECT` issue.)

You –theoretically – get all available commands from checking the R-folder

```
...R/R-2.12.1/include/Rinternals.h
```

this document is quite scary though, so I might just tell you about the commands I think are necessary for 'basic' use.

An introduction to using .Call in R can be found in

http://www.biostat.jhsph.edu/~bcaffo/statcomp/files/dotCall.pdf

However, I find this introductions gets too confusing too quickly (the beginning is pretty helpful, though), when you don't have any experience using the C structure defined by the typedef `SEXPREC` in `Rinternals.h`.

4. Usage of `PROTECT` and `UNPROTECT`

If you want to create (and finally return) an R object in your C Code, it is advisable, to use `PROTECT` and `UNPROTECT`, as the memory of R is - from time to time- garbage collected. Just imagine R being some miffty, pessimistic and cleaning addicted housewife, cleaning up especially stuff you want to keep. So make sure you protect your variables! If you create an object in your C code, you must tell R that the object is in use, by using PROTECT on a pointer to the object.

Before the command `return`, all PROTECTs and UNPROTECTs must be in balance. Which means, if you have PROTECTed 5 objects on your way through the code, you need to use UNPROTECT(5), which unprotects the last 5 objects.

PROTECTion is not needed for function arguments, as R already knows that they are in use.

5. Now we are going to adapt the remaining code of the function `foo`. Currently we are stuck with

```
SEXP foo(SEXP N, SEXP A, SEXP B, SEXP C)
    {
    ...
    return R_NilValue;
    }
```

First thing to change: We don't want to return nothing, but some vector C. Notice: As we can return something, it is no longer necessary to include this unspecified vector C in our list of arguments and return C via the argument of the function `foo`. We can just kick C out of out function header.
One more function argument we might want to remove (leave it if you want), is N, because the value of N is the length of vector A. Without the function argument N, we would just have to use the command `length(A);`, to extract this information from A.

Now, as we don't have any C yet, we need to generate our C (`SEXP C;`),
tell the program that C ought to be a (R-type) `numeric` (i.e. C-type REALSXP) vector of the same length as the function argument A (`C=allocVector(REALSXP,length(A))` or, if we write `int n=length(A);` first, `C=allocVector(REALSXP,n)`)
and protect our new vector C, as it's a new object we create!

To make sure we don't forget, we immediately add an `UNPROTECT(1);` at the end of our code (Remember, the PROTECTs and UNPROTECTs have to be balanced!)

```
SEXP foo(SEXP A, SEXP B)
    {
    SEXP C;
    int n=length(A);

    PROTECT( C=allocVector(REALSXP,n));
    ...
    UNPROTECT(1);
    return(C);
    }
```

Next step is the `for` loops. They basically remain the same. Only thing that changes is the way we assign values to the vector C.
Before, this was done by

```
C[i]=A[i]*B[j];
```

We could now try using this, but would fail. The thing is, we need to put the command REAL in front of our vectors, before we can access them, meaning to set, for example, the second component of vector A to 2 (indexing starts at zero...) would require the command

```
REAL(a)[1]=2;
```

To save us from typing `REAL` all the time, we could also just use the following shortcuts

```
double *ra=REAL(A);
double *rb=REAL(B);
double *rc=REAL(C);
```

We are now able to deal with our 'ordinary' pointers and can thus write:

```
rc[i]=ra[i]*rb[j];
```

Without the 'shortcuts' we would have

```
REAL(C)[i]=REAL(A)[i]*REAL(B)[j];
```

which is just as correct.

All in all, our function `foo` now looks like this

```
SEXP foo(SEXP A, SEXP B)
    {
    SEXP C;
    int n=length(A);

    PROTECT( C=allocVector(REALSXP,n));


    double *ra=REAL(A);
    double *rb=REAL(B);
    double *rc=REAL(C);

    for(int i=0;i<n;i++)
       {
       for(int j=0;j<n;j++)
          {
           rc[i]=ra[i]*rb[j];
          }
       }
    UNPROTECT(1);
    return(C);
    }
```

7.2 N2N

Don't index with double variables, e.g.

```
SEXP a;
double i=3;
REAL(a)[i]=14;
```

won't work.

Careful about the way C indexes matrices. They are interpreted as some giant vector and filled up columnwise. For example:

```
SEXP f;
PROTECT( f=allocMatrix(REALSXP,3,2) );
```

yields a 3x2 matrix and if you want to fill the second column, you ought to write

```
double *rf=REAL(f);

rf[0+3]=14;
rf[1+3]=14;
rf[2+3]=14;
```

Write numbers like $1.0 * 10^{-15}$ in the R-style using $1e$-15! Make sure you use `fabs` when intending to compute the absolute value of, for example, `rf[2]`:

```
fabs(rf[2]);
```

because

```
abs(rf[2]);
```

doesn't work. I spent quite some time figuring this out...