1 R Package writing tutorial

From R ⇔C++

R by itself is a quick prototyping language that is capable of running large linear-algebra commands rather quickly. However, if your gibbs sampler, graph search algorithm, ... involves a lot of nested loops, sorting, linking, you might feel you need the added speed of compiled code. Some days this can give you ($\times 100$) speed improvement, sometimes it feels like ($\times 1.2$), it depends on how efficiently you've already coded in **R**, how many "`apply()`" and "`sum()`" you may have successfully used.

Also coding **C++** can multiply devolopment time ($\times 10$), even if you've already prototyped, so you're going to have to ask yourself whether those days spent stamping out memory bugs are worth it. Assuming you already know **C++**, and are familiar with reading **R** documentation, one can expect still to waste about 8 hours trying to realize by yourself what order of operations you need to get these two doohickeys to match. This document intends to be a step-by-step tutorial for Windows users (Unix/Mac people have it a bit easier, but I don't own one), that hopes to reduce this hair-pulling experience to down around an only-somewhat-unreasonable 3 hours.

1.0.1 What is an R-package?

You're already familiar hopefully, with the packages available from:

```
Packages->Install Package(s)...
```

which download automatically from CRAN and can add to **R** everything from a front-end Stat-100-use GUI (`Rcmndr`) to better mixed regression (`lme4`) to 3d plotting (`plot3d` , `scatterplot3d`). If you're trying to write a standard function that doesn't exist in the base **R**, or you're trying to work with some sort of data, image files (Use `tuneR` for `.wav` files),what you're looking for might already be coded for you. A **R** package is a directory system containing **R**-code, data, documentation, and compiled C++ `.dll` files that can all be loaded together (eventually) with a single `library()` command. You don't actually need to write a **R** package to get your C code into there, but once you've already fought 7/10ths of the battle, you might as well complete the whole quest. This will save you a lot of button pushing and code copy-paste in the long run.

1.0.2 Documents to read

1. Writing **R**-Extentions [1]: The definitive 100 page documentation to package writing in **R**. While everything you need to know is indeed contained in here, it also contains a lot more information that you don't need to know, at least to get started. For the "neophyte", its better to learn by doing, which involves a step by step tutorial with sample code, which is what I'm trying to present here.

2. The **R** for Windows FAQ [2]: The problem with Windows is that the path from "writing C++ code from your favorite compiler", to " getting it loaded into **R**", is that there are a handful of pain-in-the-ass tricks necessary. I will try to demonstrate these tricks in order, this FAQ can try to explain the necessity/origin of these needs.

3. Rtools [3]: A set of programs that the Windows user will need to downlaod ($\approx$ 50 mb) to compile code correctly. More explanation to follow.

4. Making R-Packages Under Windows: A Tutorial [4, 5] : I owe Peter Rossi a great deal of debt for coming up with the first, most logical tutorial to writing R-packages. Unfortunately, it seems like the all important `test.zip` demo files are inaccessible from his website, which would be almost everything you need. Hence, this tutorial FAQ, which includes sample code that I've mostly written myself.

1.1 Step 1: Download R-tools

The R-tools (`http://www.murdoch-sutherland.com/Rtools/`) are a couple of R-utilities and compilers necessary to get your C++ code into `.dll` formatted specially for **R**. These include a `MinGW gcc/g++` compiler which has the `<R.h>` and `<Rmath.h>` headers already installed. Unix-familiar users will recognize the joy/heartache of g++, and the accompanying `Makefile`'s. If you're accustomed to free-to-use Visual C++, Borland C++, you can try your best to translate the "**R** for Windows FAQ" advice into a working method, but I can't help you here.

At this point, all you have to do is download and run `Rtools.exe` to get the basic R-tools installed, choose the simplest directory names. `TCL/TK` is a universal graphics package (used to write `Rcmdr` and you probably don't need it.

1. Download/install Perl (`http://www.activestate.com/Products/activeperl/`) Apprently if you want to write your code in Perl, have a blast, I'm only vaguely familiar with the language. Largely, the Perl is used to compile your **R**-package documentation. Downloading is free and easy from this website, so you might as well do it.

2. Download/install Miktex `http://www.miktex.org` : This is also to compile Package documentation if you write any. Having Miktex, if you're an academic statistician, is something you're going to want to do eventually (it's how I wrote this pdf). I recomend TeXnicCenter (`http://www.toolscenter.org/`) as free windows text editor program that meshes nicely to write Latex

3. Download/install "Innosetup". Unless you are some sort of superuser and wish to rebuild and compile the complete **R**`.exe` from the source code, you do not need to do this. I repeat, if you are writing your own **R** package, you do not need to do this.

1.1.1 Awkward Step 1A: Adjust your PATH directory.

This is a weird one but, you've got to do it so that your **R** tools can all talk to each other.

1. Right-click on `My Computer` and select `Properties`

2. Select the `Advanced` tab and click on `Environmental Variables`

3. In the `System variables` box, scroll down to the Variable-item "Path".

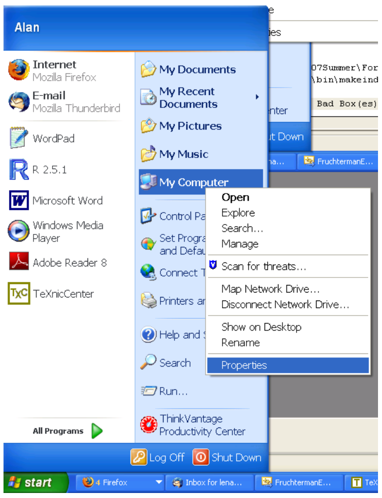
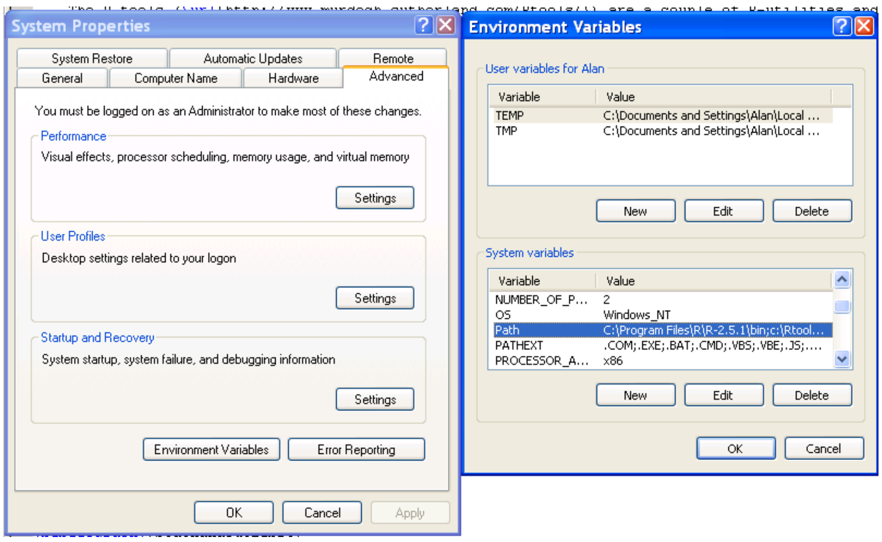


Figure 1: Properties Menu of My Computer



You must be logged on as an Administrator to make most of these changes.

Visual effects, processor scheduling, memory usage, and virtual memory

Desktop settings related to your logon

System startup, system failure, and debugging information

User variables for Alan

Variable	Value
TEMP	C:\Documents and Settings\Alan\Local ...
TMP	C:\Documents and Settings\Alan\Local ...

System variables

Variable	Value
NUMBER_OF_P...	2
OS	Windows_NT
Path	C:\Program Files\R\R-2.5.1\bin;c:\Rtool...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	x86

Figure 2: Opening System Properties->Environment Variable window

4. To edit the "Path" Variable, click Edit and you'll get a frustrating tiny box. Copy the "Variable value" into a text editor because you're going to have to modify it.

5. The Path variable is for when Windows starts to look for files that you or some program tries to execute, but don't happen to be in the working directory. Windows then searches these directories in the order of mention under its configuration Path variable. The order is important. You're going to want to make sure that this variable contains in order (these should come first)

   - `c:\R\bin` or `C:\Program Files\R\R-2.5.1\bin`
     or wherever your **R**-executable is stored
   - `c:\Rtools\bin`
   - `c:\Rtools\MinGW`
   - `c:\Perl\bin`
   - `C:\Program Files\MiKTeX 2.6\miktex\bin`
   - And then everything else you previously had stored under that Variable Value

   You'll probably have to alter these directories as I've supplied them, if you've installed **R**, Perl, etc. to alternative directories. Directories should be separated by a semi-colon ";".
   Here's my updated Path varaible:

```
C:\Program Files\R\R-2.5.1\bin;c:\Rtools\bin;c:\Rtools\MinGW\bin;c:\Perl\bin;
C:\Program Files\MiKTeX 2.6\miktex\bin;C:\Program Files\ThinkPad\Utilities;
%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;
C:\Program Files\ATI Technologies\ATI.ACE\;C:\Program Files\Common Files\Adobe\AGL;
C:\Program Files\Diskeeper Corporation\Diskeeper\;
C:\Program Files\IBM ThinkVantage\Client Security Solution;
C:\Program Files\ThinkPad\ConnectUtilities;
C:\Program Files\Intel\Wireless\Bin\;C:\PROGRA~1\SecureFX
```

6. Copy your updated Path variable up to the variable value of the Edit screen (minimize a few of the windows you have open and you'll probably find the edit box still open behind it), click OK in the edit box, and then OK in the Environmental Variables Box.

1.1.2 Step 1B: Figuring out if you did the previous awkward step successfully, using Command Prompt

Command Prompt is the old DOS interface, that we old farts (born early 80s) remember fondly from yesteryear. To open Command Prompt

Open `start->All Programs->Accessories->Command Prompt`

You will get a Command Prompt window. Now you should try to type the following commands into the window to test whether you get output

- "`Path`" – Learn what your path variable looks like
- "`R.exe`" – Open up **R**, should boot up from any directory if you've done this correctly. `q()` to quit.

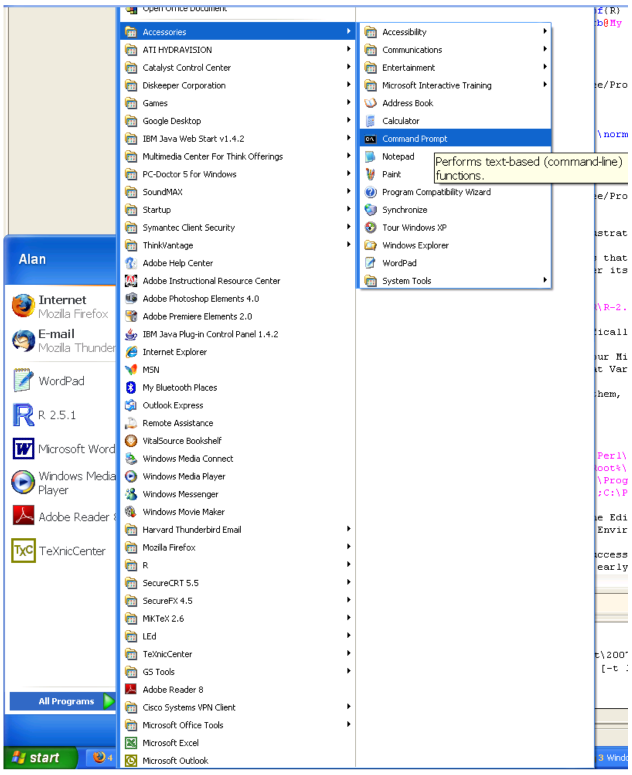


Figure 3: Opening Command Prompt

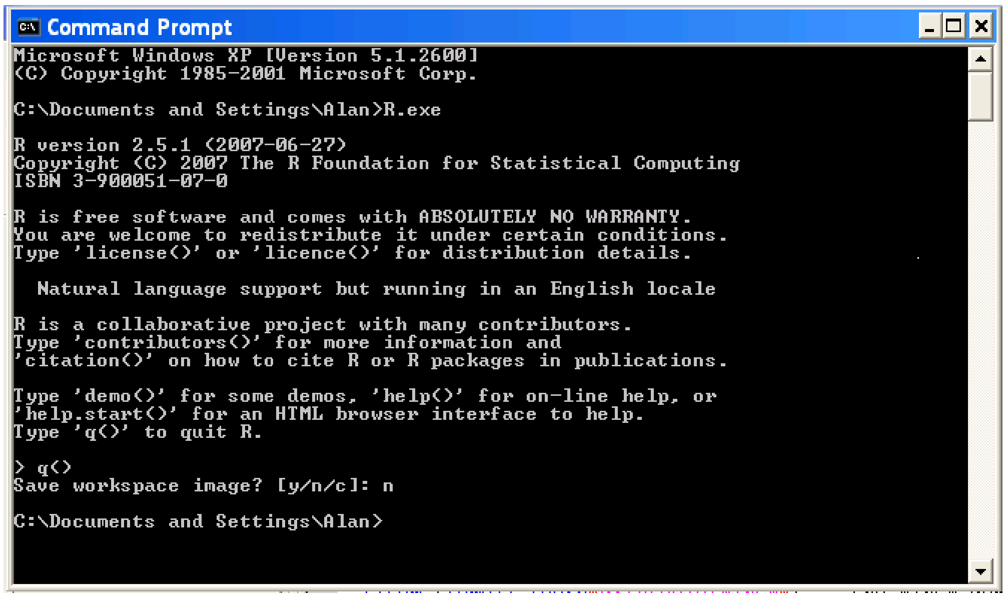
- "`gcc --help`" – Test MinGW
- "`perl --help`" – Test Perl
- "`Tex --help`" – Test Miktex
- "`grep`" – Test to see that Rtools is connected

If these all boot up you've done your Path variable coding correctly [4].

1.1.3 Step 1C: Setting TMPDIR

1. While you're at it, with the enivornmental variables window open, add a new variable if it's not their yet called `TMPDIR`. And set it to whatever the other Temporary directories you've got floating around seem to be (probably `C:\WINDOWS\TEMP` or `C:\Rtools\TEMP` if you make sure they exist.). Do this so that `R CMD check` will run correctly once you're ready to compile a complete package. Don't pick a directory with spaces.

2. What's that? You've set a `TMPDIR` and Windows won't recognize it? Does `R CMD check` still give you missing directory messages? Same with me. So the other alternative is to make sure that `C:\TEMP` exists on your harddrive. This is the default requirement. I generated the folder, and afterwards R-Check would run.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Alan>R.exe

R version 2.5.1 (2007-06-27)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> q()
Save workspace image? [y/n/c]: n

C:\Documents and Settings\Alan>
```

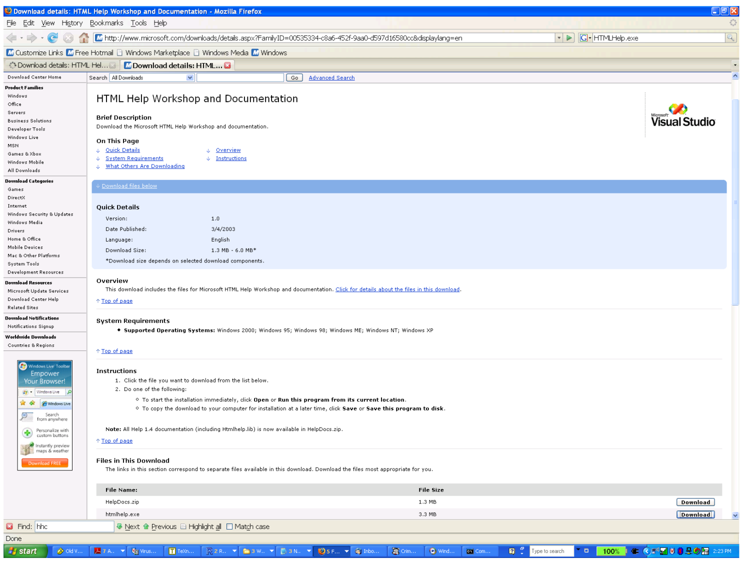
Figure 4: Command Prompt window

1.1.4 Step 1D: Getting HTMLHelp.exe

Visit link

http://www.microsoft.com/downloads/details.aspx?FamilyID=00535334-c8a6-452f-9aa0-d597d16580ccdisplaylang=en

or google a page looking for HTMLHelp.exe


Download details: HTML Help Workshop and Documentation - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.microsoft.com/downloads/details.aspx?FamilyID=00535334-c8a6-452f-9aa0-d597d16580cc&displaylang=en
HTMLHelp.exe
Customize Links Free Hotmail Windows Marketplace Windows Media Windows
Download Center Home
Search All Downloads Go Advanced Search
Visual Studio
Next Previous Highlight all Match case

HTML Help Workshop and Documentation

Brief Description
Download the Microsoft HTML Help Workshop and documentation.

On This Page
Quick Details · Overview · System Requirements · Instructions · What Others Are Downloading

Download files below

Quick Details

Version:	1.0
Date Published:	3/4/2003
Language:	English
Download Size:	1.3 MB - 6.0 MB*

*Download size depends on selected download components.

Overview
This download includes the files for Microsoft HTML Help Workshop and documentation. Click for details about the files in this download.

Top of page

System Requirements
- **Supported Operating Systems:** Windows 2000; Windows 95; Windows 98; Windows ME; Windows NT; Windows XP

Top of page

Instructions
1. Click the file you want to download from the list below.
2. Do one of the following:
   - To start the installation immediately, click **Open** or **Run this program from its current location**.
   - To copy the download to your computer for installation at a later time, click **Save** or **Save this program to disk**.

Note: All Help 1.4 documentation (including Htmlhelp.lib) is now available in HelpDocs.zip.

Top of page

Files in This Download
The links in this section correspond to separate files available in this download. Download the files most appropriate for you.

File Name:	File Size	
HelpDocs.zip	1.3 MB	Download
htmlhelp.exe	3.3 MB	Download

Figure 5: Downloading HTMLHelp.exe

This is another step necessary to getting R CMD check to opperate correctly on Windows machines, this creates Windows office help files correctly. Afterwards you're going to want to add that new directory created which contains `hhc`. (Probably `C:\Program Files\HTML` Help Workshop) to your `Path` variable (got to do that again!!).

1.1.5 1E. Get a useful General-Coding Text Editing program

You don't want to use Notepad or Wordpad to write your **R** code or your documentation files or your `.cc` **C++** code because it will save your files with unnerving `.txt` endings, add a lot of bad CRLF carriage returns to all of your endlines. So far I recommend crimson editor.

http://www.crimsoneditor.com/english/download.html

But it doesn't seem like this text editor satisfactorilly eliminate the CRLF returns, I'll try and find a better one.

1.2 Step 2 Creating an R-package directory

A **R** library package is a set of directories, that, when named correctly, **R** will be able to load during Run-time as a library under the `library()` command. If you've already got all of the **R** structures booted up in **R** you can get a library going using the `package.skeleton()` command. However, if you're starting from scratch you might as well start making folders. In whatever directory you're going to make your package, named however you'd like your package named (keep it short), create the folders

- `man` – Documentation
- `R` – All of your functions coded in **R** will go here
- `src` – Source code (C++, Perl, Fortran etc), uncompiled `.cc`, `.hh`, and compiled `.dll` files.
- `data` – **R** saved data files.
- Other optional folders, you can read up the R-ext [1] document for more, or look into the package directory at your favorite packages.

Only the `R` directory is necessary if you've written a strictly `R` based package, this is where `R` will look for functions to install.
Create an example package in directory `c:/RPackages/Test/`. The name of this package will be "Test".

1.2.1 Root Directory files

Two files should be contained in the Root directory (in this case `c:/RPackages/Test/`).

1. `DESCRIPTION` as described in section 1.1.1 of R-Extentions [1]. This is a statement about the contents which creates the chief manual page. Here, as per Peter Rossi's advice [4]:

```
Package: < name of the package>
    Version: < number of the form n.n -- start with 0.0>
    Date: < date in yyyy-mm-dd format >
    Title: < short title  try to be a descriptive as possible
               no more than one line >
    Author: <your name (s  use comma) followed by your email address
                 enclosed in < brackets >>.
    Maintainer: < one name  dont end in comma,period >
    Depends: < version of R  use a recent version  its free, why not upgrade?
                 e.g. R (> = 2.0.1)
    Description: < 5-10 line description. Highlight major features and
    Intended audience >
    License: GPL (version 2 or later)
```

Or for this test package:

```
Package: test
 Version: 0.0
 Date: 2007-15-01
 Title: Test Package
 Author: Alan Lenarcic <herecomesthespam@harvard.edu>.
 Maintainer: Alan Lenarcic <welcomespam@harvard.edu>
 Depends: R (>= 2.0.1)
 Description:  A set of demonstration functions using C, C++, R-code
 License: GPL (version 2 or later)
```

2. `NAMESPACE` The namespace identifies for **R** what files/functions it should be looking for It contains two functions

   - `UseDynLib( ..., ... )` – in here, put the names of the `.dll` files you'll be booting up here (don't worry, we'll get to that.
   - `export( ..., ...)` – in here, put the names of **R** functions of interest that you'll be booting up, often declared in `.R` files.

   In our case let's go with :

   File: `NAMESPACE`

```
useDynLib(XDemo)
export(XDemoAutoC)
```

3. Now that we have defined the skeleton, our next steps are to create the C++ code that will compile into `XDemo.dll` and then create the `.R` file `XDemo.R` which will contain code for the **R**-function `XDemoAutoC()` .

1.3 Actual C++ Code

You're most itching to learn how C++ interfaces with R, here's your chance. Copy the following three code files, save them in your `/src/` directory.

XDemo.cc:

1.
```
///////////////////////////////
//   XDemo.cc
//        C++ Function File
#include "XDemo.h"
#include "R.h"         // R memory io
#include "Rmath.h"     // R math functions
Xclass::Xclass(double *InputVec, int LengthInput) {
this->XLength = LengthInput;
this->XVec = (double*) Calloc( this->XLength,
                      double ); // Calloc is R memory Call
for (int ii = 0; ii < XLength; ii++) {
 XVec[ii] = InputVec[ii];
}
}
Xclass::~Xclass() {
Free(XVec);
}
/////////////////////////////////////////////
//     Xclass:AutoCor()
// Returns AutoCorrelation of Given Vector
double* Xclass::AutoCor() {
int DoDrops = (int) floor( XLength / 4); // AutoCor to +- L/4 positions;
double *ReturnVec = (double *) Calloc( DoDrops, double);
double RecordSum;
int ii, jj;
for (ii = 0; ii < DoDrops; ii++) {
RecordSum = 0;
for (jj = ii; jj < XLength-1; jj++) {
           RecordSum += (XVec[jj+1]-XVec[jj]) * (XVec[jj-ii+1] - XVec[jj-ii]);
}
ReturnVec[ii] = RecordSum / ((double) XLength - ii );
}
return(ReturnVec);
}

double *XDemo(double *InputVec, int LengthInput) {
Xclass *MyXClass = new Xclass(InputVec, LengthInput);
double *Returner = MyXClass->AutoCor();
delete(MyXClass);  // Delete memory caused by constructing Xclass;
return( Returner );
}
```

XDemo.h:

2.
```
  ////////////////////////////
//   XDemo.h   Header File
//
class Xclass {
private:
   int XLength;
public:
   int GetXLength();
   double *XVec;
   Xclass(double *InputVec, int LengthInput);  // Constructor
   ~Xclass(); // Destructor;
   double *AutoCor();
};


double *XDemo(double *InputVec, int LengthInput);
```

`XDemo_main.cc`:

3.
```
  ///////////////////////////////
//   XDemo_main.cc
//     C interaction file
#include "XDemo.h"
#include "R.h"      // R functions
#include "Rmath.h"  // R math
//  Functions Passed to C++ from R must be passed in C extern format
//    All variables are passed to C by reference (pointers);
//    All output of functions is "void" (adjustments made via reference change)
extern "C" {
void DemoAutoCor(double *RetV, int *pLwant, double *InputVec, int *pLengthInput) {
      double *AutoOutput = XDemo(InputVec, pLengthInput[0]);
  int ii;
  int MaxTake = pLwant[0];
  if ( (int) floor(pLengthInput[0] / 4) < MaxTake) {
    MaxTake = (int) floor(pLengthInput[0] / 4);
  }
  for (ii = 0; ii <MaxTake; ii++) {
    RetV[ii] =AutoOutput[ii];
  }
  Free(AutoOutput); // Free Memory created by function

  double UseLessNormal = .4 + rnorm(0.0, 1.0) * 2;
                // Completely Useless Generation of Normal
  Rprintf("DemoAutoCor:: Completely Useless Normal = %.3f\n", UseLessNormal);
R_FlushConsole();
    R_ProcessEvents();
  return; // Return Nothing.
    }
}
```

1.3.1 Explaining the files.

As you see, there are three files. `XDemo.cc` contains the bulk of the C++ code. `XDemo.h` is the header file `XDemo_main.cc` is the main function wrapped up in a header file. You should notice multiple ideas.

- "`R.h` " and "`Rmath.h`" included.

  These are the main C++ libraries for use with R.

- `XDemo_main.cc` is wrapped in an `extern "C" {...}` statement.

  R itself cannot process C++ code. Instead it only runs C code! It seems silly, and I don't know why it's true. But all of your class constructor/destructor declarations should be included in a file that is NOT contained in the extern statement. The functions that will be available to R are wrapped in the extern statement, and cannot include class declarations / definitions of class functions / overloaded operators / C++ things. However, you can run and create a class within these functions, C will treat these like defined structures and remain willfully ignorant of their C++ roots. This extern statement is weird, but you have to make it.

- " `Rprintf(...)` "statement

  This prints to the standard output of the **R** console, just like `printf()` . Useful for figuring out what you are doing. Must be followed immediately by the `R_FlushConsole()` , `R_ProcessEvents` commands to get **R** to actually display this writing immediately. (At the moment these commands are called, **R** writes all of the text it has been waiting to write since the last `R_FlushConsole()` command.

- " `Calloc( int LengthArrah, type)` " statement

  This is the `malloc/calloc` for use with **R** inline code. There is a little bit better memory managment involved here. Free arrays/structures developed with `Calloc()` with the statement `Free( ... )`

- " `rnorm(0.0, 1.0)` " calls **R** written random number generators to give random result. Before running C code, a statement "`set.seed(666)` " will fix the seed to a deterministic location. Ensuring that during testing, you get the same sequence of random numbers when code is reran.

1.3.2 Compiling that code

1. Open Command Prompt window as before:

   `start->All Programs->Accessories->Command Prompt`

2. Change directories to the `\src\` directory: `cd  c:/RPackages/Test/src`

3. Compile code with `CMD SHLIB` command :

   ```
   R CMD SHLIB XDemo.cc XDemo_main.cc -o XDemo.dll
   ```

   As we see, after the `R CMD SHLIB` command the files we want to compile are listed (with `XDemo.cc` first, followed by `XDemo_main.cc`), and then the `-o` statement anounces where to save the compiled `XDemo.dll` file.

1.4 R code for interacting with C

Now to learn how to use these functions in **R**, how to load them into **R** with " `dyn.load()` ", and how to call the functions with "`.C()` ".

1.4.1 The load-in

`dyn.load()` usage is simple. In the **R**-console, or anywhere in **R**-code, to load in a `.dll`

- For our compiled `XDemo.dll` that means we need a **R**-code line:

  `dyn.load("c://RPackages//Test//src//XDemo.dll")`

  If you typed the correct directory name you should get no warning messages.

- To learn whether the desired `DemoAutoCor()` function is accessible we can type

  `is.loaded("DemoAutoCor")`

  which will return `TRUE` if the loading has occured correctly. If you get a `FALSE` statement, we're in trouble and need to double-back to see where you went wrong. Was your operable function saved in an `extern()` statement? Did the whole file really complile, no warning messages in the `extern()` output of the `R CMD SHLIB` command?

- `dyn.unload("c://RPackages//Test//src//XDemo.dll")` will unload the `.dll` . Don't do this now. It just happens that often you'll want to recompile your **C++** code because there is a bug. Assuming **R** still has the `.dll` loaded, `R CMD SHLIB` will not let you compile unless the `.dll` is unloaded from all instances of **R**.

1.4.2 Calling a C++ function in R

The operable command is " `.C()` ". This will call the correct **C++** function, you supply the function **R** `list()")` object that contains all of the input variables that you will pass by pointer, as well as the output variables that you pass by pointer. Here's the operable example for our Demo function:

```
 x = rnorm(30,0,1);
UseData = x + c( 0, .3 * x[1:29]) + c(0,0, .1 * x[1:28]);   ## Creates Some Data

   OutLength  = floor(length(x) / 4)                        ## Desired Output Length

#### Now call to .C()
OutPut =.C("DemoAutoCor", OutVec=as.double( vector("numeric", OutLength) ),
              OutLength = as.integer(OutLength),
             InputVector = as.double(UseData),
             InputLength = as.integer(length(UseData))
           );

 ### Gives the full outputted list() object:
OutPut
```

This should demonstrate a correct call to `.C()` . First give the name of the desired C++ function, followed by named objects, which will be collected as a list and sent to the final output. These named objects should be given `as.VARIABLETYPE()` statements that correspond to the correct necessary type. From the **R**-exts pdf [1]:

R storage mode	**C** type	**FORTRAN** type
`logical`	`int *`	`INTEGER`
`integer`	`int *`	`INTEGER`
`double`	`double *`	`DOUBLE PRECISION`
`complex`	`Rcomplex *`	`DOUBLE COMPLEX`
`character`	`char **`	`CHARACTER*255`
`raw`	`unsigned char *`	`none`

Read up on `.C()` in the documentation, with this example in hand you now know all you need to know to get started complining **C++** for use in **R**.

1.4.3 Passing a String to C++ from R

As an "interesting" piece of information, if you want to give a string to C++, you should pass it in a `as.charachter()` format. As stated, **C++** will receive this as a type `char **` That indeed means a pointer to a pointer. You will find that if you passed say a string in this method, such as say `as.charachter("This String")` . You will find that the statement:

```
void CFunction( ..., char **InputString, ... ) {
 Rprintf("The String is Stored as %s \n", InputString[0]);
}
```

works. In other words, the string you want is a character array stored in the zeroth position of first index of the "array of arrays" you sent to it.

1.5 Finishing a package

All of the **R**-code you choose to use should be saved in the `PACKAGENAME/R/` directory, preferably in files headed `FILENAME.R` . They should be self running code without errors or warning messages (when used correctly). Predominantly it's best to have these files define **R** functions which wrap a correct `.C()` call so that users never need to call them. The advice in most package manuals is to have 10 times as much **R** code as **C++** code. Since **C++** is unwieldy, and will often require the generation of data tables/objects/classes, I don't think this is possible. However, if in the line of your code, you seem to need a computation that **R** already solves rather quickly (such as matrix multiplication/inversion/eigenvalues). Call those operations in your **R** code, leaving to your `.C()` commands only functions which require loops-within-loops or memory linking computations that are best when raw coded in **C++**.

1.5.1 Example .R file

- File name `XDEMOAUTOC.R` :

```
XDemoAutoC <- function(  UseData,  OutLength = -999, PlotToggle = FALSE) {
       if (OutLength == -999) {
```

```
                OutLength = floor(length(UseData) / 4);
            }
          #### Now call to .C()
          OutPut =.C("DemoAutoCor", OutVec=as.double( vector("numeric", OutLength) ),
                      OutLength = as.integer(OutLength),
                      InputVector = as.double(UseData),
                      InputLength = as.integer(length(UseData))
                    );
          if(PlotToggle == TRUE) {
              xxxn = 0:(length(OutPut$OutVec)-1);
              plot(OutPut$OutVec~xxxn, type="n", main="Autocorrelation Estimates");
              for (ii in 1:length(xxxn)) {
                lines( c(0, OutPut$OutVec[ii])~c(xxxn[ii], xxxn[ii]), lwd=3, col="black" );
              }
          }
          return(OutPut$OutVec)
     }
```

- Save this into the `c:/Rpackages/test/R/` directory.

1.5.2 Writing man files

You should now write a `.Rd` file, for your function, which will help configure the manuals so that other people can use your function. Here, for my test example:

File name `XDemoAutoC.Rd` :

-
```
\name{XDemoAutoC}
\alias{XDemoAutoC}
\title{Demostration Autocorrelation Function}
\description{
     A R to C interfacing Autocorrelation function, does plots too.
}
\usage{
     XDemoAutoC(  UseData,  OutLength = -999, PlotToggle = FALSE)
}
\arguments{
     \item{UseData}{Time series vector input data}
     \item{OutLength}{Set's maximum 0...Outlength time lag used in program}
     \item{PlotToggle}{Plot upon completion?}
}
\references{ R core development.  "Writing R Extentions".  2007. }
\author{Alan Lenarcic, Harvard Statistics Department PhD student}
\keyword{ts}
\keyword{array}
\seealso{ ~ \code{acf} }
\examples{
    x = rnorm(30,0,1);
    UseData = x + c( 0, .3 * x[1:29]) + c(0,0, .1 * x[1:28]);   ## Creates Some Data
    XDemoAutoC(  UseData,  OutLength = 10, PlotToggle = TRUE);
}
```

- Save this into the `c:/Rpackages/test/man/` directory.

Understanding the `.Rd` comes easier if you look at a few help pages in the **R** function. Entries are written `\command{ ... }` in no particular order. Some explanation

1. `\name{ }` :, `\alias{ }` give some names to the function
2. `\title{ }` : short title for function. `\description{ ... }` is obvious.
3. `\usage{ ... }` : Show typical order of arguments in the functions.
4. `\arguments{ ... }` . Describe the arguments that the function takes. Item arguments, one `\item{ argument-name }{ argument-description }` : statment for each argument the function takes.
5. `\references{ ... }` . Some information about where you might have taken this algorithm from (you might be publishing this package!).
6. `\author{ ... }` : You made it.
7. `\keyword{ ... }` : One can locate a file `//Rhome//doc//KEYWORDS` which lists all of the standard key words that one might want your function searchable under. Use a separate line and `\keyword{ ... }` statement for each new keyword.
8. `\seealso{ ... }` : There's a way to get hyperlinks between your various **man** page entries, but I can't figure it out. The `\code{ ... }` command simply tells the text publisher to make the `...` within the statement print out in boxy-computer code like letters.
9. `\examples{ ... }` Give an example (pointing to data or randomly generating data), which uses the function and creates output that end users can read to learn how to use the function. Make sure this **R** code executes without any prior coding (from a fresh **R** boot), and that it doesn't generate errors or warnings. When `R CMD check` compiles and checks your code, it will try to execute all of your `\examples{ ... }` code. If the `\examples{ ... }` outputs errors/warnings, then package generation will fail.

1.5.3 Test those pages in the man directory

1. In Command Prompt window, change directories to `c:/Rpackages/test/man/`

```
cd c:
cd\Rpackages
cd test
cd man
```

2. Now check the manual page with `R CMD RD2txt`

```
R CMD RD2txt XDemoAutoC.Rd
```

This converts the man page to an acurate `.txt` file version. Hopefully output looks fine

And check again the manual page with `R CMD Rdconv`

3.
```
R CMD rdconv --type=html --output=XDemoAutoC.html XDemoAutoC.Rd
```

   This tests whether the man page can be created into a correct `Rhtmlv` file. Something is buggy about the way this command eats your lettering, I couldn't copy this command string from my **pdf** file to get this to work, I have to type it into the Command Prompt window directly.

1.5.4 Finishing it all off.

It's time to run `R CMD check` which will compile your code, combine it with the **R** code and manual pages to make a package.

1. In Command Prompt window, change directories to `c:/Rpackages/`

```
cd c:
cd\Rpackages
```

2. Now run `R CMD check` by entering into the command prompt:

```
R CMD check Test
```

   That is the `R CMD check` plus the instruction to pick the directory `Test` for content.

If you're lucky like me, then this will fail and you'll get a mesage (among a long list of other things. that:

```
ERROR library(test) cannot be loaded
```

I don't know what the issue happens to be. I managed to get my own package to compile, but not this demo package. However, if I copy my whole demo package as I designed into directory `c:/RPackages/XDemo` and then run `R CMD XDemo check` in the `c:/RPackages` directory, then I do run, getting a fully compiled package ddirectory. We've made it here, there's still some incidental bugs to be solved, but following this path gives a good chance to write your own successfuly package.

References

[1] T. R. C. D. Team, "Writing r-extentions v. 2.5.1," Tech. Rep., 2007. [Online]. Available: http://cran.r-project.org/doc/manuals/R-exts.html

[2] ——, "The r for windows faq v. 2.5.1," Tech. Rep., 2007. [Online]. Available: http://cran.r-project.org/bin/windows/base/rw-FAQ.html

[3] B. Ripley and D. Murdoch, "Building r for windows and rtools installer," 2007.

[4] P. Rossi, "Making r packages under windows: A tutorial," Tech. Rep.

[5] ——, "Adding functions written in r to c," Tech. Rep.