cxxPack User Guide

R/C++ Tools for Literate Statistical Practice

Dominick Samperi

June 14, 2010

Contents

1 Introduction

The **cxxPack** package facilitates the process of building R packages and research compendiums that make heavy use of both R and **C++**. It extends the R package **Rcpp** by providing an application layer on the **C++** side, and it extends **Sweave** by making it possible to create vignettes with embedded R and **C++** code chunks. The package includes **C++** classes that model commonly used R data structures like data frames and time series, and it provides an extensible collection of tools including special functions and a financial date library.

This document serves as a user guide for the **cxxPack** package and also as an example of how to create a vignette that contains both R and **C++** code chunks. The same technology can be used to create research compendiums following the ideas of reproducible research [1, 4] and literate statistical practice, or LSP ([6],[9], [8]).

Recall that a vignette is a file with extension `.Rnw` that is normally stored in the package subdirectory `inst/doc`. It contains LaTeX source with embedded R code chunks (delimited using special character sequences). The **cxxPack** package permits **C++** chunks to be included as well. These **C++** code chunks can be compiled on the fly to create a shared library that is called from an R chunk in the usual way using `.Call()`. **C++** code chunks are compiled using the R function `loadcppchunk()`.

`Sweave` transforms a vignette file into a TeX file with suffix `.tex` that can be processed with `pdflatex`, `bibtex`, etc. In the process it executes each R code chunk that it finds and places the output into the target TeX file, optionally preceded by the R code itself.

Several packages have been built on the **cxxPack** framework and will be released shortly including: **FractalPack** (time series), **CreditRiskPack** (credit modeling), **VolSurfPack** (volatility surfaces including implied trees), **ComplexSysPack** (complex networks, fractal structures, etc.), and **BondPack** (fixed-coupon bond calculator with support for many "odd" features).

See `http://www.stat.uni-muenchen.de/~leisch/Sweave/` for more information on `Sweave` including the latest version of the Sweave User Manual. In this connection also see [8]. For details about package creation see the *Writing R Extensions* document available at the R web site. For more information about the **Rcpp** package see [5]. For information about the **zoo** time series package see [11]. For information about the **RUnit** see [2]. For general information on the design of R, S4 classes, and foreign language interfaces, see [3].

This document is organized as follows. Section 2 explains how to use the **Sweave** extensions with the help of the obligatory "hello world" program and a simple dot product example. We also explain how vignettes in R packages are processed.

Section 3 explains how to create a package that uses the **cxxPack** and **Rcpp** libraries. This is done with the help of a bare bones skeleton or template package that can be used as the starting point when creating a new package. We do not use the R function `package.skeleton()`. Instead we provide a minimal self-contained example package that the user can learn from and modify as needed.

Section 4 presents a number of examples using most of the classes from **cxxPack**, without getting into a lot of detail regarding syntax. For more details on the individual classes see Section 6.

Sections 5 and 6 discuss details about the **Rcpp** classes that we use, and about the **cxxPack** classes, respectively.

There are two appendices that discuss advanced topics like exception handling and compatibility issues. The user should at least skim through this material to prevent surprises.

We will follow the following color conventions. Code that is in a vignette file will be colored cyan, **C++** source code that is included in the final output by **Sweave** will be colored red, and R commands and output that are written to the final output will be colored blue. Vignette code will only be shown in the following section to explain how **Sweave++** is used.

Incidentally, to get an R code chunk to display as is, that is, to not be executed by **Sweave**, it is not enough to use the `Verbatim` environment. Each line of the block must begin with a blank space.

2 Using Sweave++

2.1 Preliminaries

The package vignette file `cxxPackGuide.Rnw` will be used to illustrate how to use the new features of **Sweave**.[1] It is located in the package directory `cxxPack/inst/doc/`. **C++** source code chunks that are to be loaded on the fly are placed into the directory `cxxPack/inst/doc/cpp/`.[2]

At the top of `cxxPackGuide.Rnw` there is the following important line:

```
\usepackage[nogin]{Sweave++}
```

Note that we use **Sweave++.sty** instead of the standard **Sweave.sty** style file. **Sweave++.sty** is part of the package skeleton (or template) that has been prepared for use with **cxxPack**.

Near the top of `cxxPackGuide.Rnw` **Sweave** options are specified using LaTeX commands `\SweaveOpts` and `\setkeys`. These settings are recommended for use with **Sweave++**. For details see the **Sweave** User Manual.

Here is the first R code chunk in the file. It's name is "`lib`" and it does not generate any output (`echo=FALSE`). It simply loads the package library and sets the flag `compile` that is used to control the behavior of `loadcppchunk()`). When `compile` is `FALSE` the compilation step is suppressed (and the build process runs faster).

A log of the build process is written to `cxxPack/inst/doc/cpp/compile.log`. The name of the log file can be changed using the `logfile` parameter of `loadcppchunk()`. See the man page for more information.

```
<<lib,echo=FALSE>>=
library(cxxPack)
compile=TRUE
@
```

The syntax (called "noweb" syntax) is very simple. A Chunk begins with a line of the form `<<name,options>>=`, and it ends when a line beginning with `@` is encountered.

2.2 Hello World

Now let's include our first **C++** code chunk, the obligatory "hello world" example.

```
\cppinclude[red]{testHello}
```

This includes `cxxPack/inst/doc/cpp/testHello.cpp` in verbatim mode (colored red). This is what you get after processing with **Sweave**:

```
#include <cxxPack.hpp>
RcppExport SEXP testHello() {
    return Rcpp::wrap("hello world");
}
```

All R objects are accessed through pointers of type `SEXP` on the **C++** side. What is happening here is that the **C++** string `"hello world"` is copied to R's address space and a pointer to it (of type `SEXP`) is returned by `Rcpp::wrap()`. This value is then returned by `testHello()`.
Here is an R code chunk that calls `testHello`:

```
<<testHello.R,echo=TRUE>>=
<<lib>>
loadcppchunk('testHello')
.Call('testHello')
@
```

[1] The `.Rnw` suffix derives from '`R`' and '`noweb`', and the code chunk syntax follows that of '`noweb`'.

[2] I might be helpful compare this vignette with the much simpler one that is part of the template package **MyPack**—see Section 3.

This chunk is named `testHello.R`, and we specify that the output should be echoed to the target TEX file. Note that this chunk refers to the previously defined one named `lib`. This is what **Sweave** produces in the final report:

```
> library(cxxPack)
> compile=TRUE
> loadcppchunk('testHello')
> .Call('testHello')

[1] "hello world"
```

We see here that **Sweave** did a kind of macro substitution, expanding the `lib` chunk. Then `loadcppchunk()` is used to compile `testHello.cpp`, create a shared library (in the same directory), and load this library, making the symbol `testHello` accessible from R. The function `testHello` is called via the `.Call()` interface as shown.

Of course, exported functions that are part of the package shared library can be called without first calling `loadcppchunk()` because they are made accessible by the `library` command in the `lib` chunk above.

2.3 Dot product

Next we consider a slightly more interesting example. Consider the function defined in `testDotProduct.cpp` (located in `cxxPack/inst/doc/cpp/`). To include this source file we use:

```
\cppinclude[red]{testDotproduct}
```

After **Sweave** processing we get:

```
#include <cxxPack.hpp>
RcppExport SEXP testDotproduct(SEXP x, SEXP y) {
    BEGIN_RCPP
    Rcpp::NumericVector nv1(x), nv2(y);
    double sum=0;
    for(int i=0; i < nv1.size(); ++i)
        sum += nv1(i)*nv2(i);
    return Rcpp::wrap(sum);
    END_RCPP
}
```

This function simply computes the dot product of two input vectors that appear as `SEXP`'s on the **C++** side. The `RcppExport` directive ensures that the symbol `testDotproduct` is exported from the library to which the compiled object file is written. The **cxxPack** client header file `cxxPack.hpp`, automatically includes the client header file for **Rcpp**, `Rcpp.h`.

The markers `BEGIN_RCPP` and `END_RCPP` should always bracket the code in a function that will be called from R—Appendix A.1 explains why. The use of `Rcpp::NumericVector` should be self-explanatory, and we use `Rcpp::wrap()` just as we did previously to get a `SEXP` representation for the answer to be returned to R. For more details on the syntax see Section 5.1.

Note that `Rcpp::NumericVector` is a proxy class in the sense that `nv1` refers directly to the R object pointed to by `x`, and similarly for `nv2` and `y`. In particular, the R vector is not copied. Section 5.2 explains the benefits and costs of this implementation.

Here is an R code chunk that calls this function. The call to `loadcppchunk()` takes care of compiling the function, creating a shared library (for example, `testDotProduct.so`), and loading this library.

```
<<testDotProduct.R,echo=TRUE>>=
<<lib>>
loadcppchunk('testDotproduct',compile=compile)
x <- 1:5
```

```
y <- 1:5
sum(x*y)
.Call('testDotproduct',x,y)
@
```

The newly created library is used to resolve the reference in the `.Call()`, where two vector inputs are supplied. Note that we nave *not* passed `PACKAGE='cxxPack'` to `.Call()`, because `testDotproduct` is not defined in the package shared library.[3]

Here is what we get after **Sweave** processing:

```
> library(cxxPack)
> compile=TRUE
> loadcppchunk('testDotproduct',compile=compile)
> x <- 1:5
> y <- 1:5
> sum(x*y)

[1] 55

> .Call('testDotproduct',x,y)

[1] 55
```

When `compile` is `FALSE` here the compilation step is skipped, and `loadcppchunk()` just loads the library, which must have been created previously. This is useful for a vignette like this one that contains many uses of `loadcppchunk()`. Processing is much faster if compilation is not required.

Important Note: If vignette processing fails with an error about not being able to open a shared object file, a common cause is that `compile=FALSE` *here. Use this feature only after a successful run where all libraries are built, and do not forget to set it back to* `TRUE` *in the* `lib` *chunk above.*

2.4 Processing a vignette

R packages are normally available from CRAN in several formats including: source archive (`.tar.gz` suffix), windows binary (`.zip` suffix), and MacOS X binary (`.tgz` suffix). Since MacOS is very similar to Linux most of our comments about Linux should apply to MacOS, and we will say no more about MacOS in this document.

The package **cxxPack** includes a vignette defined by `cxxPack/inst/doc/cxxPackGuide.Rnw`. More generally, R packages can contain vignettes defined by files with suffix `.Rnw` in the package subdirectory `inst/doc`. Since vignettes can include code chunks that refer to package functions the package library needs to be built in order to process vignettes in that package. In the case of the **cxxPack** package the end result of this processing is the PDF file `cxxPackGuide.pdf` (in the same subdirectory).

Next we explain how to process `cxxPackGuide.Rnw`. First we explain how to do this as a side effect of the package build process. Then we show how to do it manually after the package has been installed. The latter method is useful during development of the package (and the vignette) because it does not require a complete pacakge rebuild after each edit.

First, make sure all packages that **cxxPack** depends on have been installed. This can be done by starting R and running:

```
> install.packages(c('Rcpp','RUnit','zoo'))
```

See Section 3 for more information on this. Then downlaod the source archive `cxxPack_7.0.3.tar.gz` from CRAN (the latest version number may be different, of course).

The source archive can be unpacked and rebuilt like this:

[3]If it was defined in both libraries and the `PACKAGE` options was not used, the local version would be used (not the package library) because that library was loaded last.

```
$ tar -xvzf cxxPack_7.0.3.tar.gz
$ mv cxxPack_7.0.3.tar.gz cxxPackCRAN.tar.gz
$ R CMD build cxxPack
```

The first command will unpack the archive with root directory `cxxPack` in the current working directory (containing `cxxPack/R`, `cxxPack/src`, etc.). The second command renames the source archive since otherwise it will be overwritten by the build process. The third command builds the package source archive (what we started with). It will contain all of the source code, documentation, etc. for the package, as well as the vignette PDF file.

In order to create the vignette PDF file the package shared library needs to be built (because code chunks may call package functions), but the shared library is not included in the source archive. Thus two important outputs of the build process here are `cxxPack_7.0.3.tar.gz` (the source archive) and `cxxPackGuide.pdf` (processed vignette, included in the source archive).

To build a source archive without vignette processing use:

```
$ R CMD build --no-vignettes cxxPack
```

If vignette processing was not previously done the result will be an source archive `cxxPack_7.0.3.tar.gz` that is missing vignette PDF files. Source archives at CRAN normally contain vignette PDF files.

The `--no-vignettes` option is useful for building a source archive (or a Windows binary) in cases where the vignettes have already been processed. For example, the Windows binary for **cxxPack** can be built from its source archive as follows:

```
$ R CMD build --binary --no-vignettes cxxPack
```

The output file is `cxxPack_7.0.3.zip`. This assumes that all of the necessary Windows tools have been installed (see Section 3.3).

When processing a vignette with R and **C**++ code chunks shared libraries and other intermediate files are created that are only needed for this processing. These files should not go into an archive intended for distribution (either `tar.gz` or `zip`). This cleanup normally happens automatically at build time. If necessary a manual cleanup can be done as explained below.

Before a final INSTALL or submission to CRAN a package is normally checked using:

```
$ R CMD check cxxPack_7.0.tar.gz
```

A package source archive can be installed (under UNIX) using:

```
$ R CMD INSTALL cxxPack_7.0.tar.gz
```

and the vignette can be viewed by starting R and using:

```
> vignette('cxxPackGuide')
```

Under Windows the package is normally installed from a Windows binary (`zip` file). This can be done by starting R and using:

```
> install.packages('cxxPack_7.0.zip')
> vignette('cxxPackGuide')
```

cxxPack can also be installed directly from CRAN using:

```
> install.packages('cxxPack')
```

The system will present a list of repositories to choose from.

The cycle of editing the `.Rnw` file, then building the package, then installing the package, then starting R to view the vignette (PDF file) is not very convenient. Thus for development purposes a vignette can be processed by hand as follows.

First, make sure the environment variable `R_HOME` points to the R home directory (for example, `/usr/local/lib/R`). To process the **cxxPack** vignette by hand use:

```
$ cd cxxPack/inst/doc
$ sh ./makepdf.sh cxxPackGuide
```

Here a simple shell script is used to transform `cxxPackGuide.Rnw` into `cxxPackGuide.pdf`. The script file is self-explanatory, it simply uses **Sweave** to generate the corresponding TEX file, then uses the standard tools `pdflatex`, `bibtex`, etc. to generate the final PDF file.[4]

A log of all uses of `loadcppchunk()` is written to `cxxPack/inst/doc/cpp/compile.log`. The same directory contains all generated object files and libraries.

The directories `inst/doc` and `inst/doc/temp` also contain a number of intermediate files. All of the intermeditae files can be deleted using:

```
$ cd cxxPack/inst/doc
$ sh ./makeclean.sh
```

But note that this precludes the use of the `compile=FALSE` option of `loadcppchunk()`. To use `compile=FALSE` the intermediate files (including shared libraries) should not be deleted.

While in this development mode it is possible to include **C++** source files directly from the package `src` directory using:

```
\srcinclude[red]{myfunc}
```

This can be useful for developing a vignette (or research compendium) in parallel with the actual research, resembling a kind of unit testing.

Important Note: the `srcinclude` *command cannot be used in packages intended for distribution because they will not pass* `R CMD check`. *The reason is that the path to the* `src` *directory is invalid during* `check`. *This command is intended for use during package/vignette development only.*

Finally, a remark about development under Windows. When building a package under Windows with vignette processing the intermediate files are *not* automatically deleted, so the output file (`.tar.gz` or `.zip`) will comtain many large files that are not needed. To create a clean windows source archive that does not contain a lot of "junk" use:

```
$ cd cxxPack/inst/doc
$ sh ./makepdf.sh cxxPackGuide
$ sh ./makeclean.sh
$ cd ../../..
$ R CMD build --no-vignettes cxxPack
```

Of course, if the vignette PDF file has already been generated and there are no intermediate files then only the last command is needed. The output source archive is `cxxPack_7.0.3.tar.gz`. To generate a Windows binary archive (`cxxPack_7.0.3.zip`) add the `--binary` option.

2.5 Stangle

While not important for our purposes we mention that there is another program related to **Sweave** named **Stangle**. Instead of "weaving" source code and text, it extracts all of the code chunks ("untangles them"). This is how it would be used to untangle the R code chunks from `cxxPackGuide.Rnw`:

```
$ R CMD Stangle cxxPackGuide.Rnw
```

The R chunks are written to R scripts using the chunk name, so in our case one of the generated scripts is `testHello.R`. To run it stand-alone simply start R and `source()` this file. Alternatively, `Rscript` can be used:

```
$ Rscript testHello.R
```

Note that if there is graphics output (for example, `testFFT.R`), the second method will not work.

[4]Only the most basic shell syntax is used so this should work with most modern shells, including the one that is shipped with Windows **Rtools**.

3 R Package Creation Quick Start

3.1 Generic comments

The purpose of this section is to indicate how to create a new package that employs **cxxPack** (and **Rcpp**) as quickly as possible with minimal fuss. For this purpose we will use the archive `cxxPack/inst/template/MyPack_1.0.tar.gz` that comes with **cxxPack**. It is a bare bones package that is pre-configured to use **cxxPack**, **Rcpp**, **zoo** and **RUnit** (unit testing package). All of these packages must be installed before the template can be used. The template package also includes a skeleton vignette that has embedded **C++** code chunks.

To install **cxxPack** along with all of the packages that it depends on use:

```
> install.packages('cxxPack')
```

You will be prompted for a location to download from. Select one that is nearby.

The build process under Windows is very similar to the one under Linux thanks to a collection of UNIX emulation tools named **Rtools** and a Windows version of TEX named **MikTeX**. The Linux case will be discussed in the next section, and the changes needed for Windows will be indicated in Section 3.3

3.2 Linux

To unpack the source archive for the template package use:

```
$ tar -xvzf MyPack_1.0.tar.gz
```

This will create a directory hierarchy rooted at `MyPack` including `MyPack/R`, `MyPack/man`, `MyPack/src`, `MyPack/inst/doc`, etc.

Normally the user would insert new source files into the respective subdirectories, change `MyPack/DESCRIPTION` to reflect the new package name and author, update `MyPack/NAMESPACE`, etc., and every occurrence of `MyPack` would be replaced with the new package name.

Let's assume for the time being that we will keep the name `MyPack` (useful for quick tests). To build a source achive use:

```
$ R CMD build MyPack
```

This will create the package shared library in order to process the vignette (`MyPack/inst/doc/MyPackDoc.Rnw`), and the final result `MyPack_1.0.tar.gz` will contain the package source plus the vignette PDF file (`MyPackDoc.pdf`).

To install the package (with its vignettes) use:

```
$ R CMD INSTALL MyPack_1.0.tar.gz
```

The package can also be installed by starting R and using:

```
> install.packages('MyPack_1.0.tar.gz')
```

Now R can be started and the package loaded in the usual way using the `library()` function. The package includes a function `MyTest()` that is defined in `MyPack/R/MyTest.R` and documented in `MyPack/man/MyTest.Rd`. It is basically the exported interface for a **C++** function that is defined in `MyPack/src/MyPack.cpp`. The returned value from this function is assigned the (S3) class `MyTest`, and a print method for this class is defined in `MyPack/R/MyTest.R`. Both `MyTest()` and the associated print method are exported in `MyPack/NAMESPACE`. See *Writing R Extensions* for additional information on this.

After loading the **MyPack** package the `MyTest()` man page can be viewed using `?MyTest`, and this function can be invoked directly with no arguments. This will cause the `print` method for `MyTest` to be called, displaying the values that were returned.

Looking at `MyPack/src/MyTest.cpp` we see that the function expects two arguments, a double value, and a data frame. It converts that input `SEXP`'s to the appropriate **C++** data types. In the case of the input data frame, this is done in two essentially equivalent ways, illustrating that

Rcpp::as<>() behaves essentially like a SEXP constructor (see Section 5.1 for more details). Finally, the class Rcpp::List is used to build and return four named items, two doubles, and two data frames (the last two are equal, of course).

Now looking at MyPack/R/Mytest.R, we see how the class MyTest is assigned to the return value, and we see how the print function print.MyTest fetches the items that are returned and prints them. In the case of the data frames (with class data.frame), printing is dispatched to print.data.frame().

It should be noted that S3 class dispatching like this can be done in a cleaner object-oriented fashion using the newer S4 classes (and generic methods), but this requires a little more work. See Section 4.6 for an example.

The file MyPack/src/Makevars shows how compiler and linker flags are set so that the headers and libraries of **cxxPack** (and **Rcpp**) are found at build time. Of course, both of these packages must be installed before MyPack can be built. There are commented lines in Makevars that indicate how external libraries can be added. For more sophisticated auto-configuration (under UNIX) see the sample files in MyPack/inst/examples.

3.3 Windows

To build under Windows the **Rtools** collection must be downloaded and installed. **Rtools** can be found at http://www.murdoch-sutherland.com/Rtools/index.html. The tools include a UNIX shell (sh), rm, ls, tar, etc. Also included are perl and the MinGW version of the GNU **C**++ compiler (g++).

Note that the version of Rtools must be compatible with the version of R that is installed. See the Web site for more information. By default **Rtools** is installed into c:\Rtools.

Another important tool set needed to process vignettes is the Windows implementation of TeX named **MikTeX**. After downloading **Rtools** and **MikTeX**, the search path can be set using something like (change versions as needed):

```
set R_HOME=c:\Program Files\R\R-2.11.1
set PATH=%R_HOME%\bin;%PATH%
set PATH=c:\Rtools\bin;%PATH%
set PATH=c:\Rtools\MinGW\bin;%PATH%
set PATH=c:\Rtools\perl\bin;%PATH%
set PATH=c:\Program Files\MikTeX 2.7\miktex\bin;%PATH%
```

With the environment set the MyPack package can be used as in the Linux case, except that "R CMD" needs to be replaced with "Rcmd" in some cases. For example, to build a source archive use:

```
$ Rcmd build MyPack
```

To create a Windows binary when vignettes have already been processed (PDF files exist) use:

```
$ Rcmd build --binary --no-vignettes MyPack
```

This will create MyPack_1.0.zip.

A Windows binary can be installed by starting R and using:

```
> install.packages('MyPack_1.0.zip')
```

3.4 Package creation checklist

The definitive reference on R package creation is of course the *Writing R Extensions* manual that can be found at the R web site. Here is a quick checklist on package creation steps employing the MyPack package template:

1. Replace all occurrence of MyPack with the new package name.

2. Update the DESCRIPTION file.

3. Add **C**++ source to MyPack/src as needed.

4. Add R scripts to `MyPack/R` as needed.
5. Add documentation files for R functions to `MyPack/man`.
6. Add demos to `MyPack/demo` as needed, and update `MyPack/demo/00Index`.
7. Add data files to `MyPack/data` as needed.
8. Create other directories that are to be moved to `MyPack` in `MyPack/inst` if there are any.
9. Add vignette files (`.Rnw` files) to `MyPack/inst/doc` as needed.
10. Modify `Makevars` if there are external libraries
11. Optionally create unit tests in `MyPack/inst/unitTests`.

4 Examples

4.1 High Frequency Time Series

In this example the **C**++ function shown below will be called from the R code chunk that follows it. As can be seen from the R code two objects of R's datetime type (`POSIXct`) are passed, where the second is 50 minutes larger than the first. The function `testHighFreqSeries` computes a time series of standard normal values, where the time index starts at the first datetime supplied, and then increases by 10 minute increments until the datetime is larger than the second one supplied (not included in the series).

```
#include <cxxPack.hpp>
RcppExport SEXP testHighFreqSeries(SEXP start_, SEXP end_) {
    BEGIN_RCPP
    RcppDatetime start(start_);
    RcppDatetime end(end_);
    std::vector<RcppDatetime> index;
    std::vector<double> obs;
    GetRNGstate(); // initialize R's random number generator.
    RcppDatetime datetime = start;
    int dt = 60*10; // 10 minute intervals
    while(datetime < end) {
        index.push_back(datetime);
        obs.push_back(norm_rand()); // standard normal
        datetime = datetime + dt;
    }
    PutRNGstate(); // finished with random number generator
    cxxPack::ZooSeries zoo(obs, index); // ordered but not regular
    cxxPack::ZooSeries zooreg(obs, index, 1.0/dt); // regular (liks ts)
    Rcpp::List rl;
    rl["zoo"] = Rcpp::wrap(zoo);
    rl["zooreg"] = Rcpp::wrap(zooreg);
    return rl;
    END_RCPP
}
```

```
> library(cxxPack)
> compile=TRUE
> startDatetime = Sys.time()
> endDatetime = startDatetime + 60*50 # fifty minutes later
> loadcppchunk('testHighFreqSeries',compile=compile)
> z = .Call('testHighFreqSeries', startDatetime, endDatetime)
> attributes(z$zooreg)
```

```
$class
[1] "zooreg" "zoo"

$frequency
[1] 0.001666667

$index
[1] "2010-06-14 16:33:20 EDT" "2010-06-14 16:43:20 EDT"
[3] "2010-06-14 16:53:20 EDT" "2010-06-14 17:03:20 EDT"
[5] "2010-06-14 17:13:20 EDT"
```

```
> z$zooreg
```

```
2010-06-14 16:33:20 2010-06-14 16:43:20 2010-06-14 16:53:20 2010-06-14 17:03:20
          1.6579315          -1.7336059           0.5044739           0.1899842
2010-06-14 17:13:20
         -0.7272726
```

The class `RcppDatetime` is used to model R's datetime objects, and the final time series is returned as an object of type `ZooSeries`. Actually two `ZooSeries` objects are created from the same data, the second of which is regular because the frequency is specified (see lines 18–19). This is similar to the way the `zoo()` function works on the R side (see the man page).

See the interface file `cxxPack/inst/include/ZooSeries.hpp` for more information on what constructors and methods are available for the `ZooSeries` class.

4.2 Payment Schedule

In this example a payment schedule is computed based on the input start and end dates and other parameters (time is measured in days here, not seconds). The **C**++ function defined below is called the the R code chunk that follows it.

The schedule consists of the nth specified weekday (3rd Friday here) in each month after the start date, but not exceeding the end date. After computing the schedule payments are computed for all dates after the first based on the number of days since the last date in the schedule, counted using the specified day count convention (30/360 ISDA in this case).

We could pass in each parameter as a separate `SEXP` like we did in the last example, but to illustrate how input lists are processed we use a named list instead. The R code below shows how the list is defined and passed to **C**++. On the **C**++ side the input list is processed with the help of the `Rcpp::List` class (lines 5–10). The parameters are fetched from the list by name as a `SEXP` that is then converted to the appropriate type using `Rcpp::as<>()`.

After fetching the parameters the schedule is computed with the help of the `nthWeekday` method of `FinDate` (lines 11–28). Then the payments are computed with the help of the `diffDays` class function (lines 29–47), storing the results into vectors that will be used to construct a data frame to be returned as the final result.

The data frame (type `DataFrame` is constructed by specifying a vector of `FrameColumn`'s that are in turn constructed from the vectors just computed (lines 48–54).

See the interface file `cxxPack/inst/include/DataFrame.hpp` for more information on what constructors and methods are available for the `DataFrame` class.

```
1 #include <cxxPack.hpp>
2 RcppExport SEXP testPaymentSchedule(SEXP params_) {
3     BEGIN_RCPP
4
5     // Fetch params.
6     Rcpp::List params(params_);
7     cxxPack::FinDate start(Rcpp::as<cxxPack::FinDate>(params["start"]));
8     cxxPack::FinDate end = Rcpp::as<cxxPack::FinDate>(params["end"]);
9     int nth = Rcpp::as<int>(params["nth"]);
```

```
    int weekday = Rcpp::as<int>(params["weekday"]);
    double coupon = Rcpp::as<double>(params["coupon"]);

    int nextMonth, nextYear;
    cxxPack::FinDate date = start.nthWeekday(nth, weekday);
    cxxPack::FinDate lastDate = date;
    std::vector<cxxPack::FinDate> dateVec;

    // Get schedule
    while(date <= end) {
        if(date >= start) // could have nthWeekday < start in first month.
            dateVec.push_back(date);
        if(date.getMonth() == cxxPack::Dec) {
            nextMonth = 1;
            nextYear = date.getYear()+1;
        }
        else {
            nextMonth = date.getMonth()+1;
            nextYear = date.getYear();
        }
        date = cxxPack::FinDate(cxxPack::Month(nextMonth),1,nextYear);
        date = date.nthWeekday(nth, weekday);
    }

    // Computer payments and insert in data frame.
    std::vector<std::string> colNames(4);
    colNames[0] = "Date";
    colNames[1] = "Days";
    colNames[2] = "Pmt";
    colNames[3] = "Priority"; // High, Low, Med (factor column)
    int nrow = dateVec.size();
    std::vector<std::string> rowNames(nrow);
    std::vector<int> colDays(nrow);
    std::vector<double> colPmt(nrow);
    std::vector<std::string> priority(nrow); // factor observation
    rowNames[0] = "1";
    colDays[0] = 0; colPmt[0] = 0; priority[0] = "Low";
    for(int i=1; i < nrow; ++i) {
        rowNames[i] = cxxPack::to_string(i+1);
        colDays[i] = cxxPack::FinDate::diffDays(dateVec[i-1],dateVec[i],
                                        cxxPack::FinEnum::DC30360I);
        colPmt[i] = 100*coupon*colDays[i]/360.0;
        priority[i] = (i%2 == 0) ? "Med" : "High"; // arbitrary
    }

    cxxPack::Factor factor(priority);
    std::vector<cxxPack::FrameColumn> cols(0);
    cols.push_back(cxxPack::FrameColumn(dateVec));
    cols.push_back(cxxPack::FrameColumn(colDays));
    cols.push_back(cxxPack::FrameColumn(colPmt));
    cols.push_back(cxxPack::FrameColumn(factor));
    cxxPack::DataFrame df(rowNames, colNames, cols);

    return df;
    END_RCPP
}
```

Here is an R code chunk that exercises the payment schedule function:

```
> library(cxxPack)
> compile=TRUE
> startDate = as.Date('2010-04-15')
> endDate = as.Date('2011-02-28')
> nth = 3
> weekday = 5 # 3rd Friday
> coupon = .05 # coupon 5%
> params = list(start=startDate, end=endDate,
+               nth=nth, weekday=weekday, coupon=coupon)
> loadcppchunk('testPaymentSchedule',compile=compile)
> .Call('testPaymentSchedule', params)
```

```
         Date Days       Pmt Priority
1  2010-04-16    0 0.0000000      Low
2  2010-05-21   35 0.4861111     High
3  2010-06-18   27 0.3750000      Med
4  2010-07-16   28 0.3888889     High
5  2010-08-20   34 0.4722222      Med
6  2010-09-17   27 0.3750000     High
7  2010-10-15   28 0.3888889      Med
8  2010-11-19   34 0.4722222     High
9  2010-12-17   28 0.3888889      Med
10 2011-01-21   34 0.4722222     High
11 2011-02-18   27 0.3750000      Med
```

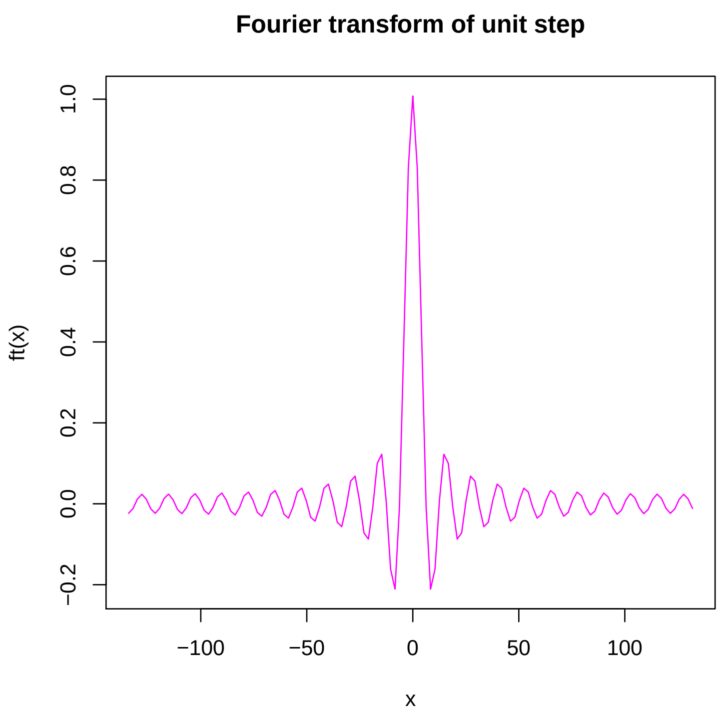
4.3 Call R's Fast Fourier Transform from C++

The **C**++ function below exercises a **C**++ interface to R's fast Fourier transform, `cxxPack::fft1d()`. It permits the programmer to work in terms of **C**++ types like `std::complex<double>`. There is some copy overhead here because `std::vector` types must be transformed to R vector types.

```
#include <cxxPack.hpp>
/**
 * Calls R's fft() with step function input.
 * Also works with Rcpp::ComplexVector.
 */
RcppExport SEXP testFFT() {
    BEGIN_RCPP
    int N = 128, fac = 1;
    double u0 = -1.5, du = 3.0/N, dx=2*3.14159265/N/du, x0 = -N*dx/2;
    std::vector<double> u(N);
    std::vector<double> x(N);
    std::vector<std::complex<double> > f(N); // f(u) = 1 on [-.5,.5]
    double funcval = 0;
    for(int i=0; i < N; ++i) {
        u[i] = u0 + i*du;
        x[i] = x0 + i*dx;
        funcval = (u[i] >= -0.5 && u[i] < 0.5) ? 1.0 : 0.0;
        f[i] = std::complex<double>(fac*funcval, 0);
        fac = -fac;
    }
    std::vector<std::complex<double> > cresult = cxxPack::fft1d(f);
    std::vector<double> result(N);
    fac = (N/2 % 2 == 0) ? 1 : -1;
```

```
    for(int i=0; i < N; ++i) {
        result[i] = fac*du*cresult[i].real();
        fac = -fac;
    }
    Rcpp::List rl;
    rl["x"] = Rcpp::wrap(x);
    rl["ft"] = Rcpp::wrap(result);
    return rl;
    END_RCPP
}
```

```
> library(cxxPack)
> compile=TRUE
>   loadcppchunk('testFFT',compile=compile)
> foo <- .Call('testFFT')
> plot(foo$x, foo$ft, type='l',main='Fourier transform of unit step',
+      xlab='x',ylab='ft(x)',col='magenta')
```



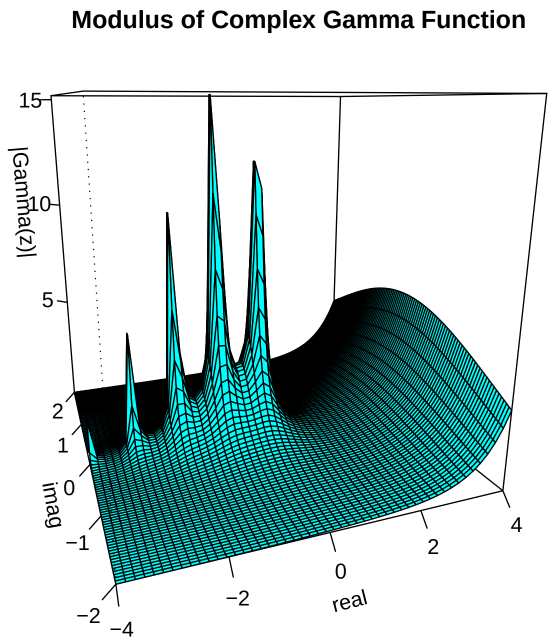
4.4 Special Functions: Complex Gamma

The complex gamma function (and the fast Fourier transform) are useful tools that have been applied in some recent credit risk management studies. This function is not currently available as part of the R core, and after implementing it I learned that there is another version in the `Rmetrics` package `fAsianOptions`. That version is written in FORTRAN, while the one in this package is written in **C**++.

Incidentally, one of the motivations for this package was to collect useful general purpose functions like this in one place, at least until they are provided as part of the R core.

Here is some R code that exercises the complex gamma function from this package. It simply evaluates the function on a rectangular grid of complex numbers and plots the modulus of the result vs z.

```
> library(cxxPack)
> compile=TRUE
> complexify <- function(x,y) {
+   complex(real=x, imaginary=y)
+ }
> Nreal <- 50
> Nimag <- 100
> rl <- seq(-4,4,length.out=Nreal)
> im <- seq(-2,2,length.out=Nimag)
> z <- outer(rl, im, complexify)
> gamma <- cxxPack::cgamma(z)
> persp(rl, im, abs(gamma),ticktype='detailed',theta=-20,
+       main='Modulus of Complex Gamma Function',col='cyan',
+       xlab='real',ylab='imag',zlab='|Gamma(z)|')
```



The complex gamma function from the `fAsianOptions` package yields the same image, but there is a small problem: it drops the dimensions and returns a 1D vector instead of a matrix. This is easily fixed by resetting the dimensions on the returned vector.

4.5 Root Finding and Optimization

The class `RootFinder1D` provides a **C**++-friendly interface to R's 1D root finder (`zeroin`), and the class `ConstrainedMinimizer` provides an interface to R's `L_BFGS_B` constrained minimizer. We only discuss `RootFinder1D` here.

Consider the trivial problem of solving for the root of $f(x) = x^2 - y$, given y. The following **C**++ code solves the problem, and since we can also do it by hand there is an easy way to check the answer.

```
#include <cxxPack.hpp>
/**
 * Test C++ interface to R's root finder.
 */
```

```
RcppExport SEXP testRootFinder(SEXP x) {
    BEGIN_RCPP
    double ysqr = Rcpp::as<double>(x);
    class PriceFunction : public cxxPack::Function1D {
        double ysqr;
    public:
        PriceFunction(double ysqr_) : cxxPack::Function1D(), ysqr(ysqr_) {}
        double value(double y) { return y*y - ysqr; }
    };
    cxxPack::RootFinder1D rootFinder;
    PriceFunction pr(ysqr);
    double root = rootFinder.solve(pr, 0, 100, 0.00001);
    return Rcpp::wrap(root);
    END_RCPP
}
```

Let's test it by computing the square root of 2:

```
> library(cxxPack)
> compile=TRUE
>   loadcppchunk('testRootFinder',compile=compile)
> .Call('testRootFinder', 2)

[1] 1.414212
```

The result looks good, so let me say a few words about the **C++** code. The `RootFinder1D` class has a method `solve` that expects an object of type `Function1D` as its first argument. The other arguments specify bounds and error tolerance. The class `Function1D` has a (virtual) method `value(x)` that is overridden in subclasses like `PriceFunction`, and the problem faced by `solve` is to find the root of `value(x) = 0`. Either it is able to do this and return the root, or it throws an exception.

In more realistic problems the class `PriceFunction` will have many other parameters besides the single value `y` that appears in this simple example, and root finding becomes non-trivial.

4.6 Bank Account Example: Persistent C++ Objects

This section illustrates how to use R's external pointers to implement persistent **C++** objects, that is, **C++** objects that maintain their state between R function calls (each call made using the `.Call` interface). Two implementations are presented, one that uses S4 classes, and a bare-bones version that does not.

One way to implement persistence is to use function closures as in the classic bank account example of [7]. This is now a well-known way to maintain state by attaching the defining environment to an R function. We will use a different approach based on external pointers following the discussion in [3], with the help of external pointer proxies provided by the `Rcpp` package.

The **C++** `BankAccount` class that will be manipulated from the R side is shown below. It contains the name and id of the account holder along with this customer's current balance. A trivial destructor has been added to illustrate some aspects of R's garbage collection.

```
class BankAccount {
public:
    std::string name;
    int id;
    double balance;
    BankAccount(std::string n, int i, double b)
        : name(n), id(i), balance(b) {}
    ~BankAccount() {
        Rprintf("BankAccount destructor called\n");
```

```
    }
};
```

The following **C++** class and associated functions will be used from R to create objects of type `BankAccount` and to manipulate these objects. There are methods to create a new `BankAccount` object, to make a deposit, and to show the current balance. Obviously a real-world application would include other methods.

The open account operation first creates a new `BankAccount` object, then uses its address to create an R external pointer with the help of the proxy class `Rcpp::XPtr`. The `true` flag supplied to the `Rcpp::XPtr` constructor tells it to register a call to the destructor for this class when R cleans up this external pointer (when it goes out of scope, for example). Finally, the external pointer is returned.

The operation of the deposit and show methods should be clear. They are passed the external pointer that was created by open, and through this pointer they access the fields of the corresponding **C++** object.

```
#include <cxxPack.hpp>
/**
 * Bank account class used to illustrate proxy pattern.
 */
class BankAccount {
public:
    std::string name;
    int id;
    double balance;
    BankAccount(std::string n, int i, double b)
        : name(n), id(i), balance(b) {}
    ~BankAccount() {
        Rprintf("BankAccount destructor called\n");
    }
};

/**
 * BankAccount open account method.
 */
RcppExport SEXP testBankOpen(SEXP name, SEXP id, SEXP balance) {
    BEGIN_RCPP
    BankAccount *p = new BankAccount(Rcpp::as<std::string>(name),
                                     Rcpp::as<int>(id),
                                     Rcpp::as<double>(balance));
    Rcpp::XPtr<BankAccount> xp(p, true);
    return xp;
    END_RCPP
}

/**
 * BankAccount deposit method.
 */
RcppExport SEXP testBankDeposit(SEXP xp_, SEXP amt) {
    BEGIN_RCPP
    Rcpp::XPtr<BankAccount> xp(xp_);
    double oldval = xp->balance;
    xp->balance += Rcpp::as<double>(amt);
    Rcpp::List rl;
    rl["name"] = Rcpp::wrap(xp->name);
    rl["oldbal"] = Rcpp::wrap(oldval);
```

```
    rl["curbal"] = Rcpp::wrap(xp->balance);
    return rl;
    END_RCPP
}

/**
 * BankAccount show balance method.
 */
RcppExport SEXP testBankShow(SEXP xp_) {
    BEGIN_RCPP
    Rcpp::XPtr<BankAccount> xp(xp_);
    Rcpp::List rl;
    rl["name"] = Rcpp::wrap(xp->name);
    rl["id"] = Rcpp::wrap(xp->id);
    rl["curbal"] = Rcpp::wrap(xp->balance);
    return rl;
    END_RCPP
}
```

Here is the first version of the R side of the solution. It does not use S4 classes. First, two accounts are created and the corresponding external pointers are stored in `bob.ptr` and `mary.ptr`, resp. These pointers are then used to perform a few transactions (with the results shown below each transaction).

To illustrate how R's garbage collection can be used to automatically cleanup **C**++ objects that are no longer used, we zero out `bob.ptr`. This causes R to cleanup the corresponding R object that `bob.ptr` was pointing to the next time it does a garbage collection sweep.

To see what happens we force garbage collection using `gc()`. The first thing we see is that the `BankAccount` destructor was called, which should not be surprising because we registered this call when we created the external pointer above. The `gc()` call also dumps some technical information about the status of R's memory. Normally explicit calls to `gc()` are not necessary.

```
> library(cxxPack)
> compile=TRUE
>   loadcppchunk('testBankAccount',compile=compile)
>   bob.ptr <- .Call('testBankOpen', 'Bob Jones', 101, 0.0)
>   mary.ptr <- .Call('testBankOpen', 'Mary Smith', 121, 0.0)
>   .Call('testBankShow', bob.ptr)

$name
[1] "Bob Jones"

$id
[1] 101

$curbal
[1] 0

>   .Call('testBankShow', mary.ptr)

$name
[1] "Mary Smith"

$id
[1] 121

$curbal
[1] 0
```

```
>   .Call('testBankDeposit', mary.ptr, 120.50)

$name
[1] "Mary Smith"

$oldbal
[1] 0

$curbal
[1] 120.5

>   .Call('testBankDeposit', mary.ptr,  50.00)

$name
[1] "Mary Smith"

$oldbal
[1] 120.5

$curbal
[1] 170.5

>   .Call('testBankShow', mary.ptr)

$name
[1] "Mary Smith"

$id
[1] 121

$curbal
[1] 170.5

>   bob.ptr <- 0
>   gc()

BankAccount destructor called
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 252812 13.6     407500 21.8   350000 18.7
Vcells 234538  1.8     786432  6.0   786428  6.0
```

The second S4 version requires that we define an S4 class and associated methods. See [3] for details about the S4 classes and generic methods.

```
> library(cxxPack)
> compile=TRUE
> setClass("BankAccount",
+          representation(extptr = "externalptr"))

[1] "BankAccount"

> setMethod("initialize", "BankAccount", function(.Object, name, id) {
+   .Object@extptr = .Call('testBankOpen', name, id, 0)
+   .Object
+ })

[1] "initialize"

> setGeneric("deposit",
+            function(object,amt) { standardGeneric("deposit") })
```

```
[1] "deposit"

> setMethod("deposit", "BankAccount",
+           function(object, amt) {
+             .Call('testBankDeposit', object@extptr, amt)
+           })

[1] "deposit"

> setMethod("show", "BankAccount",
+           function(object) {
+             .Call('testBankShow', object@extptr)
+           })

[1] "show"
```

Finally, here are some BankAccount transactions using the S4 class and methods just defined. Obviously this R code is easier to read and is more type-safe. Basically what we have done here is use S4 classes to implement the well-known proxy pattern.

```
> library(cxxPack)
> compile=TRUE
>   bob.acct <- new("BankAccount", 'Bob Jones', 101)
>   mary.acct <- new("BankAccount", 'Mary Smith', 121)
>   show(bob.acct)

$name
[1] "Bob Jones"

$id
[1] 101

$curbal
[1] 0

>   show(mary.acct)

$name
[1] "Mary Smith"

$id
[1] 121

$curbal
[1] 0

>   deposit(mary.acct, 120.50)

$name
[1] "Mary Smith"

$oldbal
[1] 0

$curbal
[1] 120.5

>   deposit(mary.acct, 50.00)
```

```
$name
[1] "Mary Smith"

$oldbal
[1] 120.5

$curbal
[1] 170.5
```

```
>   show(mary.acct)
```

```
$name
[1] "Mary Smith"

$id
[1] 121

$curbal
[1] 170.5
```

```
>   bob.acct <- 0
>   gc()
```

```
BankAccount destructor called
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 254963 13.7     467875   25   407500 21.8
Vcells 236268  1.9     786432    6   786428  6.0
```

5 Rcpp classes

5.1 Rcpp in a Nutshell

Since the focus of the **cxxPack** package is on the **C++** application layer we do not need all of the tools provided by the **Rcpp** package. The tools that we use are summarized in Figure 1.

The `Rcpp::as<T>()` template function and `Rcpp::wrap()` are used to map between R objects (`SEXP`'s) and **C++** objects, as in:

```
T d = Rcpp::as<T>(s};
SEXP s = Rcpp::wrap(d);
```

In most cases `Rcpp::as<T>()` has the same effect as an explicit construction, so the following are equivalent:

```
T d(sexp);
T d = Rcpp::as<T>(sexp};
```

The `Rcpp::List` class is the workhorse that enables us to fetch parameters by name from an input list (in a function call), or to build up a list of named results that can be returned to R.

`Rcpp::NumericVector` is a proxy class for an R double vector. For example, if `s` is a `SEXP` pointing to an R double vector, we can write to the R vector using:

```
Rcpp::NumericVector nv(s);
nv(0) = 3.14;
```

To get a copy of the original R object (and not just a proxy/wrapper) use: `Rcpp::clone()`.

Note that the proxy class `Rcpp::CharacterVector` is not a vector of `std::string`.[5] Nevertheless, it provides convenience operators that enable the user to work with objects of this class naturally like this:

[5] It is actually a typedef for `Rcpp::Vector<STRSXP>`, a template class that is parametrized by the underlying R data type.

Rcpp tool	Purpose
`Rcpp::as<T>()`	used to map SEXP to a **C**++ object (or proxy)
`Rcpp::wrap()`	used to map **C**++ object to a SEXP
`Rcpp::List`	proxy class for an R list (named entries, arb type)
`Rcpp::NumericVector`	proxy class for R double vector
`Rcpp::IntegerVector`	proxy class for R integer vector
`Rcpp::ComplexVector`	proxy class for R complex vector
`Rcpp::NumericMatrix`	proxy class for R double matrix
`Rcpp::IntegerMatrix`	proxy class for R integer matrix
`Rcpp::ComplexMatrix`	proxy class for R complex matrix
`Rcpp::CharacterVector`	proxy class for R character vector
`Rcpp::Function`	proxy class for an R function
`Rcpp::Environment`	proxy class for an R environment
`Rcpp::XPtr`	proxy class for an R external pointer
`Rcpp::clone()`	makes a copy of a proxy object
`RcppDate`	classic date class
`RcppDatetime`	classic datetime class
`BEGIN_RCPP`	macro marking the start of a **C**++ zone
`END_RCPP`	macro marking the end of a **C**++ zone

Figure 1: Selected **Rcpp** classes and functions

```
Rcpp::CharacterVector cv(5);
cv(0) = "hello world";
cv(1) = std::string("again");
if(std::string(cv(1)) == "again") return 1;
```

The `Rcpp::Function` class can be used to make calls to R functions. For example, if `s` is a `SEXP` pointing to an R function that takes two real arguments and returns a real result, the function can be called from **C**++ using, for example:

```
Rcpp::Function func(s);
double result = Rcpp::as<double>(func(3.5,8.9));
```

The return value is a `SEXP` pointing to the answer in R's address space, so `Rcpp::as<double>()` is used to fetch the double value.

The `Rcpp::Environment` class can be used to fetch a function from a particular R package. For example:

```
Rcpp::Environment stats("package:stats");
Rcpp::Function fft = stats.get("fft");
```

The `Rcpp::XPtr` class provides a simplified interface to R external pointers. These pointers can refer to memory that is managed by C/**C**++ classes that are external to R (part of an R package, for example). The example in Section 4.6 illustrates how to use this class to implement persistent **C**++ objects, that is, objects that maintain their state between R function calls.

The macros `BEGIN_RCPP` and `END_RCPP` are used to mark the beginning and end of **C**++ code sections or zones where errors can only be signalled using **C**++ exceptions—R exceptions are not allowed. We refer to all of the code braketed by these macros, including all of the code reachable (by function calls) from this section a **C**++ *zone*. This makes exception handling possible in most situations—see Section A.1.

Finally, the `RcppDate` and `RcppDatetime` classes model R's `Date` and `POSIXct` (datetime) types, respectively. They are part of what the authors call the "classic Rcpp API." This API is not part of the `Rcpp` namespace and it is no longer being actively developed. For this reason a new date class `cxxPack::FinDate` was defined for use in the financial date library of **cxxPack**. For the user's convenience most of the classes of **cxxPack** include support for `RcppDate` and `RcppDatetime` date types.

5.2 NumericVector copy semantics

Consider the following **C**++ code. The use of `Rcpp::List` should be self-explanatory.

```
#include <cxxPack.hpp>
/**
 * NumericVector copy semantics.
 */
RcppExport SEXP testNumericVector(SEXP x) {
    BEGIN_RCPP
    Rcpp::NumericVector nv(x);
    Rcpp::NumericVector av = nv;
    Rcpp::NumericVector cv = Rcpp::clone(Rcpp::NumericVector(x));
    nv(0) = 5; av(1) = 6; cv(2) = 7;
    Rcpp::List rl;
    rl["nv"] = Rcpp::wrap(nv);
    rl["av"] = Rcpp::wrap(av);
    rl["cv"] = Rcpp::wrap(cv);
    return rl;
    END_RCPP
}
```

This function expects a numeric vector argument and proxies this argument using the `Rcpp::NumericVector` class in `nv`. Then `av` is set equal to `nv`, with the result that `av` and `nv` both reference the same R memory (through a common `SEXP`). On the other hand, `cv` is a clone of the input vector, so it references a copy.[6]

Let us call this function with a real (double) vector:

```
> library(cxxPack)
> compile=TRUE
> x <- as.double(1:5)
> loadcppchunk('testNumericVector',compile=compile)
> .Call('testNumericVector', x)

$nv
[1] 5 6 3 4 5

$av
[1] 5 6 3 4 5

$cv
[1] 1 2 7 4 5

> x

[1] 5 6 3 4 5
```

Notice that the input vector `x` was modified by the changes made to `nv` and `av`, but it was not affected by the change made to `cv`, as expected. On the other hand, consider what happens when we pass an integer vector:

```
> library(cxxPack)
> compile=TRUE
> x <- 1:5
> .Call('testNumericVector', x)
```

[6]This is very similar to the way Java references and clone work. The author is grateful to Romain François for a helpful discussion on this.

```
$nv
[1] 5 6 3 4 5

$av
[1] 5 6 3 4 5

$cv
[1] 1 2 7 4 5

> x

[1] 1 2 3 4 5
```

Now the input vector is not changed. What happened is that to construct `nv` a cast had to be performed, and the end result is the `nv` and `av` both reference a *copy* of `x`, and `cv` references another copy.

Clearly there are situations where the behavior of `Rcpp::NumericVector` can be convenient, for example, direct access to R vectors can lead to faster computations. On the other hand, this example illustrates that there is a risk of unintended side-effects and other surprises. To be safe use `Rcpp::clone()` to force copying when performance is not an issue.

6 cxxPack classes

6.1 `CNumericVector` class and copy-by-value

C++ classes that model R vectors and matrices (rather than proxy them) have been implemented in **cxxPack**. The implementation makes use of the **C++** class `std::vector` that is part of the Standard Template Library (STL). This provides some leverage since necessary copy constructors are inherited from STL.

The classes are `CNumericVector`, `CNumericMatrix`, `CDateVector`, and `CDatetimeVector`. Here is a **C++** function that employs these classes.

```
#include <cxxPack.hpp>
/**
 * Test experimental CNumericVector, CNumericMatrix, CDateVector, etc.
 */
RcppExport SEXP testCNumericVector(SEXP vec_, SEXP mat_, SEXP dvec_, SEXP dtvec_) {
    BEGIN_RCPP
        cxxPack::CNumericVector cv1(vec_);
        cxxPack::CNumericMatrix cm(mat_);
        cxxPack::CDateVector dvec(dvec_);
        cxxPack::CDatetimeVector dtvec(dtvec_);
        cxxPack::CNumericVector cv2 = cv1; // uses STL copy constructors
        cv1(0) = 98;
        cm(1,2) = 99;
        dvec(0) = cxxPack::FinDate(cxxPack::Month(4), 15, 2010);
        dtvec(0) = RcppDatetime(14714.25*60*60*24); // 4/15/2010, 6AM GMT.

        Rcpp::List rl;
        rl["cv1"] = Rcpp::wrap(cv1);
        rl["cv2"] = Rcpp::wrap(cv2);
        rl["cm"] = Rcpp::wrap(cm);
        rl["dvec"] = Rcpp::wrap(dvec);
        rl["dtvec"] = Rcpp::wrap(dtvec);
        return rl;
    END_RCPP
}
```

Here is some R code that exercises this function...

```
> library(cxxPack)
> compile=TRUE
> vec <- as.double(1:5)
> mat <- matrix(as.double(1:12),3,4)
> dvec <- as.Date('2010-02-01') + 1:5
> dtvec <- Sys.time() + 1:5*24*60*60
> loadcppchunk('testCNumericVector',compile=compile)
> .Call('testCNumericVector', vec, mat, dvec, dtvec)

$cv1
[1] 98  2  3  4  5

$cv2
[1] 1 2 3 4 5

$cm
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5   99   11
[3,]    3    6    9   12

$dvec
[1] "2010-04-15" "2010-02-03" "2010-02-04" "2010-02-05" "2010-02-06"

$dtvec
[1] "2010-04-15 02:00:00 EDT" "2010-06-16 16:33:43 EDT"
[3] "2010-06-17 16:33:43 EDT" "2010-06-18 16:33:43 EDT"
[5] "2010-06-19 16:33:43 EDT"
```

The operation of the constructors should be clear. The line containing `cv2 = cv1` relies on the STL copy constructor to copy the underlying `std::vector`. The fact that the change to `cv1` does not affect `cv2` shows that these classes follow R's copy-by-value semantics.

We remark that the "classic API" class `RcppVector<double>` always made a copy, so it is a model class rather than a proxy class. Unfortunately, it never reached maturity and is no longer being actively developed by the **Rcpp** team (where the focus is more on proxy classes). Accordingly, we have added `Rcpp::wrap()` implementations for `RcppVector<double>` and `RcppMatrix<double>` to **cxxPack**.

6.2 Financial Date Library

The "classic API" classes `RcppDate` and `RcppDatetime` are minimal wrapper classes intended for use with R's `Date` and Datetime (or `POSIXct`) classes. Currently the legacy class `RcppResultSet` in **Rcpp** is used to pass objects of these types back to R. To eliminate the need for this we have implemented `Rcpp::wrap()` for both of these types.

To avoid conflicts with the legacy date functionality we have implemented a financial date library in terms of a new date class named `FinDate`. The library supports all of the usual day count conventions and has been used to implement a general purpose bond calculator (in another package not yet released).

There are also utility functions that can be used to compute the serial number used by various systems to represent a particular date (or datetime). The systems supported include R, Excel1900, Excel1904, QuantLib, IsdaCds, and Julian (i.e., Julian day number). These utility functions can be applied to objects of type `FinDate`, `RcppDate`, and `RcppDatetime`. There are **C++** and R interfaces to these utility functions, and there is a detailed R man page (see `?serialNumber`.

The file `cxxPack/inst/unitTests/runit.math.R` defines unit tests for the function `serialNumber()`. To run the tests use `runcxxPackTests()`.

The **C**++ function below exercises most of the features advertised above. It constructs two `FinDate`'s from input R `Date`'s. Then `d3` is defined to be February 28th, same year as the one associated with d1. Note the cast to the enumerated type `cxxPack::Month`. This helps to prevent confusion between m/d/y and d/m/y format because a month in the second spot will not be accepted.

Then `diff30360` is set equal to the number of days between the input dates using the ISDA 30/360 day count convention, and `diffACT` is set equal to the actual (calendar) number of days between the dates. `nthFriday` is set equal to the n-th Friday of the month that contains date `d1`. Finally, `excelnum` is set equal to the serial number used by Excel to represent date `d1`. There are two possible Excel formats—see the R man page for `serialNumber` for more information.

```
#include <cxxPack.hpp>
/**
 * Exercises the financial date library.
 */
RcppExport SEXP testDate(SEXP d1_, SEXP d2_) {
    BEGIN_RCPP
    cxxPack::FinDate d1(d1_), d2(d2_);
    cxxPack::FinDate d3(cxxPack::Month(2), 28, d1.getYear());
    int diff30360 = cxxPack::FinDate::diffDays(d1, d2,
                                               cxxPack::FinEnum::DC30360I);
    int diffACT = d2 - d1;
    cxxPack::FinDate nthFriday = d1.nthWeekday(3, cxxPack::Fri);
    double excelnum = cxxPack::serialNumber(d1, cxxPack::Excel1900);
    Rcpp::List rl;
    rl["d3"] = Rcpp::wrap(d3);
    rl["diff30360"] = Rcpp::wrap(diff30360);
    rl["diffACT"] = Rcpp::wrap(diffACT);
    rl["excelnum"] = Rcpp::wrap(excelnum);
    rl["nthFriday"] = Rcpp::wrap(nthFriday);
    return rl;
    END_RCPP
}
```

Let's test the function by supplying two dates and then checking that we get the same serial number when we use the version of `serialNumber` that is exposed as an R function:[7]

```
> library(cxxPack)
> compile=TRUE
> d1 <- as.Date('2010-05-15')
> d2 <- as.Date('2010-06-15')
> loadcppchunk('testDate',compile=compile)
> .Call('testDate',d1, d2)

$d3
[1] "2010-02-28"

$diff30360
[1] 30

$diffACT
[1] 31

$excelnum
[1] 40313
```

[7]Pasting this serial number into an Excel cell and formatting as a date should reveal `5/15/2010`, provided Excel is used on a PC with default options.

```
$nthFriday
[1] "2010-05-21"

> serialNumber(d1, 'Excel1900')

[1] 40313
```

6.3 DataFrame class

The class `DataFrame` can be used to build a **C++** representation for an R data frame. There is a constructor that takes a `SEXP` and it does what you would expect: builds a **C++** representation of the R data frame that this `SEXP` points to.

Conversely, the `DataFrame` **C++** object can be mapped to R's address space and represented by a `SEXP` through an `operator SEXP()` type cast. This means `Rcpp::wrap()` can be applied to a `DataFrame` object.[8]

The following **C++** code shows how a `DataFrame` object can be constructed from an input R data frame, and it also shows how such an object can be created from native **C++** data structures. In the second case the `DataFrame` is first "dimensioned" by specifying the row names, column names, and column types. Then the data is filled in. Note that this method does not permit column types `COLTYPE_LOGICAL` and `COLTYPE_FACTOR`. If the `DataFrame` must have columns of these types then the columns must be built separately and combined using a different constructor, as in the second example of this section.

```
#include <cxxPack.hpp>
/**
 * DataFrame demo without constructing columns separately.
 */
RcppExport SEXP testDataFrame1(SEXP dfin_) {
    BEGIN_RCPP
    cxxPack::DataFrame dfin(dfin_);
    int ncols = 3;
    int nrows = 2;
    std::vector<std::string> colNames(ncols);
    std::vector<std::string> rowNames(nrows);
    std::vector<int> colTypes(ncols);
    colNames[0] = "id"; colTypes[0] = cxxPack::FrameColumn::COLTYPE_INT;
    colNames[1] = "amount"; colTypes[1] = cxxPack::FrameColumn::COLTYPE_DOUBLE;
    colNames[2] = "date"; colTypes[2] = cxxPack::FrameColumn::COLTYPE_FINDATE;
    rowNames[0] = "r1"; rowNames[1] = "r2";
    cxxPack::DataFrame df(rowNames, colNames, colTypes);

    // Fill in data (can also use df[0].getInt(i), etc.)
    for(int i=0; i < nrows; ++i) {
        df["id"].getInt(i) = i+100;
        df["amount"].getDouble(i) = i+100.5;
        df["date"].getFinDate(i) = cxxPack::FinDate(cxxPack::Month(4),15,2010)+i;
    }

    Rcpp::List rl;
    rl["df"] = Rcpp::wrap(df);
    rl["dfin"] = Rcpp::wrap(dfin);
    return rl;
    END_RCPP
}
```

[8]This required a hack—see the technical notes on `Rcpp::wrap()`.

Here is the R code that exercises this function:

```
> library(cxxPack)
> compile=TRUE
> dfin <- data.frame(a=c(1,2,3), b=c('alpha', 'beta','gamma'))
> loadcppchunk('testDataFrame1',compile=compile)
> .Call('testDataFrame1',dfin)

$df
    id amount       date
r1 100  100.5 2010-04-15
r2 101  101.5 2010-04-16

$dfin
  a     b
1 1 alpha
2 2  beta
3 3 gamma
```

For our second example, here is the **C++** code for a function that builds a `DataFrame` with columns of all possible types. The types include `int`, `double`, `string`, `factor`, `bool`, `FinDate`, `RcppDate`, and `RcppDatetime`. In this case the user builds all of the columns separately, places them in a vector, and passes this vector along with the row and column names to the `DataFrame` constructor.

```
#include <cxxPack.hpp>
/**
 * DataFrame demo with separate construction of each column.
 */
RcppExport SEXP testDataFrame2() {
    BEGIN_RCPP

    int ncols = 8; // use all possible column types.
    int nrows = 3;

    std::vector<std::string> colNames(ncols);
    std::vector<std::string> rowNames(nrows);

    std::vector<int> colInt(nrows);
    std::vector<double> colDouble(nrows);
    std::vector<std::string> colString(nrows);
    std::vector<std::string> factorobs(nrows);
    std::vector<bool> colBool(nrows);
    std::vector<cxxPack::FinDate> colFinDate(nrows);
    std::vector<RcppDate> colRcppDate(nrows);
    std::vector<RcppDatetime> colRcppDatetime(nrows);

    colNames[0] = "int";
    colNames[1] = "dbl";
    colNames[2] = "str";
    colNames[3] = "fac";
    colNames[4] = "bool";
    colNames[5] = "findate";
    colNames[6] = "rcppdate";
    colNames[7] = "rcppdatetime";

    RcppDatetime dt0(14714.25*60*60*24); // 4/15/2010, 6AM GMT.
```

```
    for(int i=0; i < nrows; ++i) {
        rowNames[i] = cxxPack::to_string(i+1);
        colInt[i] = i+1;
        colDouble[i] = i+1.5;
        colString[i] = "test"+cxxPack::to_string(i+1);
        colBool[i] = i%2 == 0;
        colFinDate[i] = cxxPack::FinDate(cxxPack::Month(4),15,2010) + i;
        colRcppDate[i] = RcppDate(cxxPack::Month(4),15,2010) + i;
        colRcppDatetime[i] = dt0+(.25+i)*60*60*24;
        factorobs[i] = "a"+cxxPack::to_string(i+1);
    }
    cxxPack::Factor factor(factorobs);

    std::vector<cxxPack::FrameColumn> cols(0);
    cols.push_back(cxxPack::FrameColumn(colInt));
    cols.push_back(cxxPack::FrameColumn(colDouble));
    cols.push_back(cxxPack::FrameColumn(colString));
    cols.push_back(cxxPack::FrameColumn(factor));
    cols.push_back(cxxPack::FrameColumn(colBool));
    cols.push_back(cxxPack::FrameColumn(colFinDate));
    cols.push_back(cxxPack::FrameColumn(colRcppDate));
    cols.push_back(cxxPack::FrameColumn(colRcppDatetime));

    cxxPack::DataFrame df(rowNames, colNames, cols);
    return df;
    END_RCPP
}
```

Here is some R code that exercises this function:

```
> library(cxxPack)
> compile=TRUE
>   loadcppchunk('testDataFrame2',compile=compile)
> .Call('testDataFrame2')

  int dbl   str fac  bool    findate   rcppdate        rcppdatetime
1   1 1.5 test1  a1  TRUE 2010-04-15 2010-04-15 2010-04-15 08:00:00
2   2 2.5 test2  a2 FALSE 2010-04-16 2010-04-16 2010-04-16 08:00:00
3   3 3.5 test3  a3  TRUE 2010-04-17 2010-04-17 2010-04-17 08:00:00
```

6.4 Factor class

An R factor is modeled using the `Factor` class. Here is a **C++** function that constructs an object of this class from an input R factor, and also from native **C++** data structures.

```
#include <cxxPack.hpp>
/**
 * Construct a Factor from input object and from native data structures.
 */
RcppExport SEXP testFactor(SEXP factorin_) {
    BEGIN_RCPP
    cxxPack::Factor factorin(factorin_); // From R factor

    int nobs = 8;
    std::vector<std::string> obs(nobs);
    for(int i=0; i < nobs; ++i)
```

```
        obs[i] = "Level"+cxxPack::to_string((i+1)%3+1);
    cxxPack::Factor fac(obs); // Native constructor.

    Rcpp::List rl;
    rl["factorin"] = Rcpp::wrap(factorin);
    rl["fac"] = Rcpp::wrap(fac);
    return rl;
    END_RCPP
}
```

Here is an R chunk to test the function. The logic should be clear.

```
> library(cxxPack)
> compile=TRUE
> f <- as.factor(c('good', 'good', 'bad', 'good'))
> loadcppchunk('testFactor',compile=compile)
> .Call('testFactor',f)

$factorin
[1] good good bad  good
Levels: bad good

$fac
[1] Level2 Level3 Level1 Level2 Level3 Level1 Level2 Level3
Levels: Level1 Level2 Level3
```

6.5 ZooSeries class

The ZooSeries class models an R zoo time series. Since most of the other R time series types (ts, xts, timeSeries, etc.) can be converted to and from the zoo type (using as.zoo, as.xts, etc.) it is possible to work with these at the **C++** level using the zoo representation.

A zoo time series is assumed to be sorted on the index, but the timeSeries type does not make this assumption, for example. Ultimately the raw data for a time series is a sequence of (not necessarily ordered) index values and associated data observations. When each observation is a single value, we have parallel index and data vectors. When each observation consists of several values, the index vector refers to the rows of a matrix. The timeSeries type views a time series in this raw fashion, and sorts as needed (for example, to convert to zoo type).

This is similar to the way the ZooSeries class has been implemented. When a ZooSeries object is returned to R (via Rcpp::wrap()) it is always sorted on the index, as the zoo package expects. But the user is permitted to modify the ZooSeries representation, and this can result in a ZooSeries representation that is not sorted on the index (until it is returned to R).

For our first example, here is a **C++** function that constructs a ZooSeries object from an input zoo object, and also constructs such an object from native **C++** data structures. The index here is of type FinDate. The acceptable index types are int, double, FinDate, RcppDate, and, RcppDatetime.

```
#include <cxxPack.hpp>
/**
 * Test ZooSeries with scalar observations.
 */
RcppExport SEXP testZooSeries1(SEXP zooin_) {
    BEGIN_RCPP

    cxxPack::ZooSeries zooin(zooin_);

    int n = 3; // number of dates, one scalar observation per date.
```

```
    std::vector<cxxPack::FinDate> obsdates(n); // the index.
    std::vector<double>  obs(n); // the observations.

    for(int i=0; i < n; ++i) {
        obsdates[i] = cxxPack::FinDate(cxxPack::Month(4),15,2010) + i;
        obs[i] = 100.5 + i;
    }
    cxxPack::ZooSeries zoo(obs, obsdates);

    Rcpp::List rl;
    rl["zooin"] = Rcpp::wrap(zooin);
    rl["zoo"] = Rcpp::wrap(zoo);
    return rl;
    END_RCPP
}
```

Here is some R code to test this:

```
> library(cxxPack)
> compile=TRUE
> z <- zoo(rnorm(5), as.Date('2010-04-14')+1:5)
> loadcppchunk('testZooSeries1',compile=compile)
> .Call('testZooSeries1',z)

$zooin
2010-04-15 2010-04-16 2010-04-17 2010-04-18 2010-04-19
 0.7004203 -1.2081951 -0.1248114 -2.1250170  1.4808492

$zoo
2010-04-15 2010-04-16 2010-04-17
     100.5      101.5      102.5
```

For the next example we assume that three observations are made for each index value. We also assume that the series is regular. Here is the **C++** function.

```
#include <cxxPack.hpp>
/**
 * Test ZooSeries with vector observations.
 */
RcppExport SEXP testZooSeries2() {

    BEGIN_RCPP

    // Three observations per date.

    int n = 5; // number of dates.
    int m = 3; // number of observations per date.

    std::vector<cxxPack::FinDate> obsdates(n);
    std::vector<std::vector<double> > obs(n);

    int count = 0;
    for(int i=0; i < n; ++i) {
        obsdates[i] = cxxPack::FinDate(cxxPack::Month(4),15,2010) + i;
        std::vector<double> v(m);
        for(int j=0; j < m; ++j)
            v[j] = count++;
```

```
        obs[i] = v;
    }

    cxxPack::ZooSeries zoo(obs, obsdates);

    zoo.setFrequency(1);

    return zoo;
    END_RCPP
}
```

Here is the test...

```
> library(cxxPack)
> compile=TRUE
>   loadcppchunk('testZooSeries2',compile=compile)
> z <- .Call('testZooSeries2')
> class(z)

[1] "zooreg" "zoo"

> is.regular(z)

[1] TRUE

> z

2010-04-15  0  1  2
2010-04-16  3  4  5
2010-04-17  6  7  8
2010-04-18  9 10 11
2010-04-19 12 13 14
```

A Advanced Topics

A.1 Safer Hello World: Exceptions

It turns out that our implementation of `testHello()` in Section 2.2 above has a slight problem. If the **C**++ function `Rcpp::wrap()` were to throw an exception it is likely that R will crash (this is a remote possibly here because `Rcpp::wrap()` has been well-tested). To prevent this we can try to use **C**++ exception handling like this:

```
#include <cxxPack.hpp>
RcppExport SEXP testHello2() {
    SEXP ret = R_NilValue;
    try {
        ret = Rcpp::wrap("hello world");
    } catch(std::exception& ex) {
        Rf_error(ex.what());
    } catch(...) {
        Rf_error("Unknown exception");
    }
    return ret;
}

```

```
> library(cxxPack)
> compile=TRUE
> loadcppchunk('testHello2')
> .Call('testHello2')
```

```
[1] "hello world"
```

Unfortunately, using R's `Rf_error()` function amounts to throwing an R exception, and R exceptions do not mix well with **C++** exceptions. The author is grateful to Simon Urbanek for pointing out this potential problem.

It is important to understand that this incompatibility between R and **C++** exception handling has no impact on code that works normally (does not throw exceptions). In practice it means that if there is an exception it is generally not safe to assume that recovery is possible: the problem that caused the exception needs to be fixed before reliable computations can resume. Of course, if there is a serious runtime error R is likely to crash, and the problem needs to be researched and fixed in the usual way.

Romain François has implemented work-arounds that make recovery from an exceptions possible in most situations. For example, he has introduced macros `BEGIN_RCPP` and `END_RCPP` that can be used to implement a safer version of `testHello()` as follows:

```
#include <cxxPack.hpp>
/**
 * Safer hello world.
 */
RcppExport SEXP testSaferHello() {
    BEGIN_RCPP
        SEXP ret = Rcpp::wrap("hello world");
        if(ret != R_NilValue) // logical error
            throw std::range_error("SaferHello: wrap failed");
        return ret;
    END_RCPP
}
```

We have deliberately introduced a logical error here to illustrate how the exception mechanism works. Obviously the test should be `ret == R_NilValue`.

What happens here is that any **C++** exception that occurs in the code bracketed between `BEGIN_RCPP` and `END_RCPP` is transformed into an R exception and forwarded to R. Note that this trick assumes that `Rcpp::wrap()` will not throw an R exception—call `Rf_error()`–which could have the side effect of mixing R and **C++** exceptions. If there are problems `Rcpp::wrap()` should throw **C++** exceptions, it should not call `Rf_error()`.

The **C++** function `testSaferHello()` can be called in exactly the same way that we called `testHello()` above, but since it generates an exception (by design) `Sweave` would terminate while processing this document (and you would not be reading this). To prevent this we need to call `testSaferHello()` using R's exception management framework as follows:

```
> library(cxxPack)
> compile=TRUE
>   loadcppchunk('testSaferHello',compile=compile)
> handler <- function(str) { tmp=sub(".*): ", "", str); cat("C++ exception: ",tmp) }
> tryCatch(.Call('testSaferHello'), error = handler)
```

```
C++ exception:  SaferHello: wrap failed
```

What happened is that the **Rcpp** framework caught the **C++** exception that was thrown, converted it to an R exception with a long text description, and forwarded this R exception to R. On the R side the exception is caught using `tryCatch()`, and handled by the specified error handler. In this case the handler simply strips off part of the long description added by **Rcpp**, leaving only the text that was passed to the **C**++ exception framework, and the string `"C++ exception: "` is prepended.

There is another potential problem that is taken care of automatically by the **Rcpp** framework. If **C**++ code makes a call to an R function, that R function may throw an exception, which could again improperly mix R and **C**++ exceptions. What the **Rcpp** framework does is catch such an R exception and re-throw it as a **C**++ exception. Of course, this only works for function calls that are made using the **Rcpp** framework.

The important message from this section is that the **C**++ code in a function that is called from R must be bracketed between `BEGIN_RCPP` and `END_RCPP` as in this example, and the enclosed **C**++ code should not call R's `Rf_error()` function: if there is a problem throw a **C**++ exception.

A.2 Compatibility and Technical Notes

There a number of potential compatibility issues and OS-dependencies that the user of **cxxPack** (and **Rcpp**) would be aware of. It is important that users at least browse through this list to avoid wasting time on issues that are well-understood and for which work-arounds are available. The author is grateful to Simon Urbanek for pointing out the potential exception handling and static initializer issues.

Syntax When converting R code to **C**++ for improved performance don't forget to map '`<-`' to '`=`'. The R code '`x <- y`' happens to be valid **C**++ code, but it does not translate to '`x = y`' in **C**++! To avoid this mistake use '`=`' instead of '`<-`' in R code (they are equivalent).

Exceptions R and **C**++ exception handling cannot be used at the same time. Be sure to enclose the main block of **C**++ code that is called from R between the macros `BEGIN_RCPP` and `END_RCPP`. The R `Rf_error()` function should not be called inside such a **C**++ block—throw a **C**++ exception if there is a problem. See the last section for more info.

`main.c` The R main module is currently compiled using the C compiler, not the **C**++ compiler. This means static initializers in **C**++ code may not be called before `main()` is called as they should be by the **C**++ standard. One work-around is to use explicit initialization only.

It turns out that this problem does not occur in most situations because the shared library loading mechanism makes sure that **C**++ static initializers associated with objects in the library are called at the right time. If in the future R's `main()` function is compiled using **C**++ then this issue should disappear. Another possible solution would be to adopt **CXXR** as the standard, a **C**++ version of R that is currently under development—see [10].

In `cxxPack/inst/staticInitTest` the user will find a simple test program. Here a C main program calls a **C**++ function (in a dynamically linked library) that uses two statically initialized objects. The author knows of no environments where the static initializers are not called. Unfortunately, this test depends on the GNU g++ compiler!

`std::complex` Fast (unchecked) operations on a vector of R's `Rcomplex` type can be performed by using `std::complex<double>` as a "proxy." For example, if `rptr` is a pointer to `Rcomplex`, then we can set

```
std::complex<double> *cptr = reinterpret_cast<std::complex<double>*>(rptr);
```

See the implementation of `cgamma` in `cxxPack`, for an example. It is easy to see that the same idea can be applied to types `std::vector<double>` and `double*`, when the maximum possible performance is desired.

`Rcpp::wrap()` When a **C**++ class has the type conversion `operator SEXP()` defined `Rcpp::wrap()` should use it. Currently this requires a type cast, as in `Rcpp::wrap((SEXP)df)`. This creates an asymmetry in the way `Rcpp::wrap()` is used, sometimes the cast is needed, and other times it cannot be present, and the user would need detailed knowledge of the class to know which case applies. We have implemented a work-around for the classes of **cxxPack** so that the cast to `SEXP` is never needed. This is done for `DataFrame`, for example, by adding definitions to namespace `Rcpp` at the end of its source files (`DataFrame.hpp`, `DataFrame.cpp`).

An alternative strategy that employs template meta-programming was described recently in the document *Extending Rcpp*, part of the **Rcpp** package. Unfortunately, this requires rearranging header files in a way that is not straightforward for the classes of **cxxPack**. We may switch to this strategy later. Of course, what strategy is used is invisible to the user of the class.

`unloadcppchunk` If for any reason it is necessary to delete one of the shared libraries that are created during **Sweave** processing before it completes the library will need to be unloaded first. The function `unloadcppchunk()` can be used for this purpose (see man page). All loaded libraries are automatically unloaded when the **Sweave** processing completes, so `unloadcppchunk()` is not needed in most situations.

Linking It turns out that a package library is not always a shared library in the sense that this is true under Linux, and portability problems can arise. Accordingly, for maximum portability **cxxPack** (and **Rcpp**) create *static* client libraries in most environments (Linux is an exception).

C++0x The **Rcpp** package employs many of the latest innovations in **C**++ including the features documented in the **C**++ Technical Reference 1 (namespace `std::tr1`), template metaprogramming (embedding program logic in templates), and other features scheduled to be part of the new C++0x standard late in 2011. Note that some of these features may not be supported by all compilers. The **Rcpp** package checks what features are supported by a particular compiler before using them internally.

Windows Under Windows Vista it sometimes happens that a PDF file that is created by the build process is not accessible by the person who just created it! This tends to happen when the `Rtools` shell (sh) is used as part of the build process. For example, the PDF file that is created from the vignette may not be accessible from the command window (previously called the "DOS Window"). The work-around is to use Windows file explorer instead.

When copying script files from Windows to Linux a common problem is the extra line termination characters used under Windows. Linux/UNIX terminates lines with a newline, '`\n`', whereas Windows terminates lines with '`\r\n`'. Some Linux programs will be confused by the extra '`\r`' characters. To see if they are present use:

```
$ od -c in.sh
```

To strip them use:

```
$ tr -d "\r" < in.sh > out.sh
```

References

[1] J. B. Buckheit and D. L. Donoho. Wavelab and reproducible research. In *Wavelets and Statistics*, pages 55–81. Springer-Verlag, 1995.

[2] M. Burger, K. Juenemann, and T. Koenig. *RUnit: R Unit test framework*, 2009. R package version 0.4.25.

[3] J. M. Chambers. *Software for Data Analysis: Programming with R*. Springer-Verlag, New York, 2008.

[4] D. Donoho, A. Maleki, I. Rahman, M. Shahram, and V. Stodden. 15 years of reproducible research in computational harmonic analysis. *Computing in Science and Engineering*, 11(1):8–18, January 2009.

[5] D. Eddelbuettel and R. François. *Rcpp: Rcpp R/C++ interface package*, 2010. R package version 0.8.0.

[6] R. Gentleman and D. Temple Lang. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, 16(1):1–23, March 2007.

[7] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[8] F. Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In W. Härdle and B. Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg, 2002. ISBN 3-7908-1517-9.

[9] A. Rossini and F. Leisch. Literate statistical practice. UW Biostatistics Working Paper Series, Paper 194, March 2003.

[10] A. R. Runnalls. Aspects of CXXR internals. University of Kent working paper, 2009.

[11] A. Zeileis and G. Grothendieck. zoo: S3 infrastructure for regular and irregular time series. *Journal of Statistical Software*, 14(6):1–27, 2005.