Extending \textit{R} with C++: A Brief Introduction to Rcpp

Dirk Eddelbuettel\textsuperscript{a} and James Joseph Balamuta\textsuperscript{b}

\textsuperscript{a}Debian and R Projects; Chicago, IL, USA; edd@debian.org; \textsuperscript{b}Depts of Informatics and Statistics, Univ. of Illinois at Urbana-Champaign; Champaign, IL, USA; balamut2@illinois.edu

This version was compiled on September 14, 2017

\textit{R} has always provided an application programming interface (API) for extensions. Based on the \textit{C} language, it uses a number of macros and other low-level constructs to exchange data structures between the \textit{R} process and any dynamically-loaded component modules authors added to it. With the introduction of the \textit{Rcpp} package, and its later refinements, this process has become considerably easier yet also more robust. By now, \textit{Rcpp} has become the most popular extension mechanism for \textit{R}. This article introduces \textit{Rcpp}, and illustrates with several examples how the \textit{Rcpp Attributes} mechanism in particular eases the transition of objects between \textit{R} and C++ code. Your abstract will be typset here, and used by default a visually distinct font. An abstract should explain to the general reader the major contributions of the article.

Applications and case studies | statistical computing | computationally intensive methods | simulation

1. Introduction

The \textit{R} language and environment (\textit{R Core Team}, 2017a) has established itself as both an increasingly dominant facility for data analysis, and the \textit{lingua franca} for statistical computing in both research and application settings.

Since the beginning, and as we argue below, “by design”, the \textit{R} system has always provided an application programming interface (API) suitable for extending \textit{R} with code written in \textit{C} or \textit{Fortran}. Being implemented chiefly in \textit{R} and \textit{C} (with a generous sprinkling of \textit{Fortran} for well-established numerical subroutines), \textit{R} has always been extensible via a \textit{C} interface. Both the actual implementation and the \textit{C} interface use a number of macros and other low-level constructs to exchange data structures between the \textit{R} process and any dynamically-loaded component modules authors added to it.

\textit{A C} interface will generally also be accessible to other languages. Particularly noteworthy here is the \textit{C++} language, developed originally as a ‘better \textit{C}’, which is by its design very interoperable with \textit{C}. And with the introduction of the \textit{Rcpp} package (Eddelbuettel and Français, 2011; Eddelbuettel, 2013; Eddelbuettel et al., 2017), and its later refinements, this process of extending \textit{R} has become considerably easier yet also more robust. To date, \textit{Rcpp} has become the most popular extension system for \textit{R}. This article introduces \textit{Rcpp}, and illustrates with several examples how the \textit{Rcpp Attributes} mechanism (Allaire et al., 2017) in particular eases the transition of objects between \textit{R} and \textit{C++} code.

Background. \textit{Chambers} (2008, p. 3) provides a very thorough discussion of desirable traits for a system designed to program with data, and the \textit{R} system in particular. Two key themes motivate the introductory discussion. First, the Mission is to aid exploration in order to provide the best platform to analyse data: “to boldly go where no one has gone before.” Second, the Prime Directive is that the software systems we build must be trustworthy: “the many computational steps between original data source and displayed result must all be trustful.” The remainder of the book then discusses \textit{R}, leading to two final chapters on interfaces.

Chambers (2016, p. 4) builds and expands on this theme. Two core facets of what “makes” \textit{R} are carried over from the previous book. The first states what \textit{R} is composed of: \textit{Everything that exists in \textit{R} is an object.} The second states how these objects are created or altered: \textit{Everything that happens in \textit{R} is a function call.} A third statement is now added: \textit{Interfaces to other software are part of \textit{R}.}

This last addition is profound. If and when suitable and performant software for a task exists, it is in fact desirable to have a (preferably also performant) interface to this software from \textit{R}. Chambers (2016) discusses several possible approaches for simpler interfaces and illustrates them with reference implementations to both Python and Julia. However, the most performant interface for \textit{R} is provided at the subroutine level, and rather than discussing the older \textit{C} interface for \textit{R}, Chambers (2016) prefers to discuss \textit{Rcpp}. This article follows the same school of thought and aims to introduce \textit{Rcpp} to analysts and data scientists, aiming to enable them to use—and create—further interfaces for \textit{R} which aid the mission while staying true to the prime directive. Adding interfaces in such a way is in fact a natural progression from the earliest designs for its predecessor \textit{S} which was after all designed to provide a more useable ‘interface’ to underlying routines in \textit{Fortran}.

The rest of the paper is structured as follows. We start by discussing possible first steps, chiefly to validate correct installations. This is followed by an introduction to simple \textit{C++} functions, comparison to the \textit{C} API, a discussion of packaging with \textit{Rcpp} and a linear algebra example. The appendix contains some empirical illustrations of the adoption of \textit{Rcpp}.
per. However, many external resources exist that provide detailed installation guides for R toolchains in Windows and macOS. 

As a first step, and chiefly to establish that the toolchain is set up correctly, consider a minimal use case such as the following:

```r
library("Rcpp")
evalCpp("2 + 2")
```

# [1] 4

Here the Rcpp package is loaded first via the library() function. Next, we deploy one of its simplest functions, evalCpp(), which is described in the Rcpp Attributes vignette (Allaire et al., 2017). It takes the first (and often only) argument—a character object—and evaluates it as a minimal C++ expression. The value assignment and return are implicit, as is the addition of a trailing semicolon and more. In fact, evalCpp() surrounds the expression with the required ‘glue’ to make it a minimal source file which can be compiled, linked and loaded. The exact details behind this process are available in-depth when the verbose option of the function is set. If everything is set up correctly, the newly-created R function will be returned.

While such a simple expression is not interesting in itself, it serves a useful purpose here to unequivocally establish whether Rcpp is correctly set up. Having accomplished that, we can proceed to the next step of creating simple functions.

### 3. A first C++ function using Rcpp

As a first example, consider the determination of whether a number is odd or even. The default practice is to use modular arithmetic to check if a remainder exists under x mod 2. Within R, this can be implemented as follows:

```r
isOddR <- function(num = 10L) {
  result <- (num %% 2L == 1L)
  return(result)
}
```

The operator %% implements the mod operation in R. For the default (integer) argument of ten used in the example, 10 mod 2 results in zero, which is then mapped to FALSE in the context of a logical expression.

Translating this implementation into C++, several small details have to be considered. First and foremost, as C++ is a [statically-typed] language, there needs to be additional (compile-time) information provided for each of the variables. Specifically, a type, i.e. the kind of storage used by a variable must be explicitly defined. Typed languages generally offer benefits in terms of both correctness (as it is harder to accidentally assign to an ill-matched type) and performance (as the compiler can optimize code based on the storage and cpu characteristics). Here we have an int argument, but return a logical, or bool for short. Two more smaller differences are that each statement within the body must be concluded with a semicolon, and that return does not require parentheses around its argument. A graphical breakdown of all aspects of a corresponding C++ function is given in Figure 3.

When using Rcpp, such C++ functions can be directly embedded and compiled in an R script file through the use of the cppFunction() provided by Rcpp Attributes (Allaire et al., 2017). The first parameter of the function accepts string input that represents the C++ code. Upon calling the cppFunction(), and similarly to the earlier example involving evalCpp(), the C++ code is both compiled and linked, and then imported into R under the name of the function supplied (e.g. here isOddCpp()).

```r
library("Rcpp")
cppFunction("
bool isOddCpp(int num = 10) {
  bool result = (num % 2 == 1);
  return result;
}"
)
isOddCpp(42L)
```

# [1] FALSE

### 4. Extending R via its C API

Let us first consider the case of ‘standard R’, i.e. the API as defined in the core R documentation. Extending R with routines written using the C language requires the use of internal macros and functions documented in Chapter 5 of Writing R Extensions (R Core Team, 2017b).

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

SEXP convolve2(SEXP a, SEXP b) {
  int na, nb, nab;
  double *xa, *xb, *xab;
  SEXP ab;

  a = PROTECT(coerceVector(a, REALSXP));
  b = PROTECT(coerceVector(b, REALSXP));
  na = length(a); nb = length(b);
  nab = na + nb - 1;
  ab = PROTECT(allocVector(REALSXP, nab));
  xa = REAL(a); xb = REAL(b); xab = REAL(ab);
  for (int i = 0; i < nab; i++)
    xab[i] = 0.0;
  for (int i = 0; i < na; i++)
    for (int j = 0; j < nb; j++)
      xab[i + j] += xa[i] * xb[j];
  UNPROTECT(3);
  return ab;
}
```

This function computes a convolution of two vectors supplied on input, a and b, which is defined to be $a_{i+1} = \sum_{j=0}^{nb} a_i \cdot b_j$. Before computing the convolution (which is really just the three lines involving two nested for loops with indices i and j), a total of ten lines of mere bookkeeping are required. Vectors a and b are coerced to double, and a results vector ab is allocated. This expression involves three calls to the PROTECT macro for which a precisely matching UNPROTECT is required as part of the interfacing of internal memory allocation. The vectors are accessed through pointer equivalents xa, xb and xab; and the latter has to be explicitly zeroed prior to the convolution calculation involving incremental summary at index i + j.

### 5. Extending R via the C++ API of Rcpp

Using the idioms of Rcpp, the above example can be written in a much more compact fashion—leading to code that is simpler to read and maintain.
#include "Rcpp.h"
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector convolve_cpp(const NumericVector& a, const NumericVector& b) {
  // Declare loop counters, and vector sizes
  int i, j, na = a.size(), nb = b.size(), nab = na + nb;

  // Create vector filled with 0
  NumericVector ab(nab);

  // Crux of the algorithm
  for (i = 0; i < na; i++) {
    for (j = 0; j < nb; j++) {
      ab[i + j] += a[i] * b[j];
    }
  }

  // Return result
  return ab;
}

To deploy such code from within an R script or session, first save it into a new file—which could be called convolve.cpp—in either the working directory, a temporary directory or a project directory. Then from within the R session, use Rcpp::sourceCpp("convolve.cpp") (possibly using a path as well as the filename). This not only compiles, links and loads the code within the external file but also adds the necessary “glue” to make the Rcpp function available in the R environment. Once the code is compiled and linked, call the newly-created convolve_cpp() function with the appropriate parameters as done in previous examples.

What is notable about the Rcpp version is that it has no PROTECT or UNPROTECT which not only frees the programmer from a tedious (and error-prone) step but more importantly also shows that memory management can be handled automatically. The result vector is already initialized at zero as well, reducing the entire function to just the three lines for the two nested loops, plus some variable declarations and the return statement. The resulting code is shorter, easier to read, comprehend and maintain. Furthermore, the Rcpp code is more similar to traditional R code, which reduces the barrier of entry.

### 6. Data Driven Performance Decisions with Rcpp

When beginning to implement an idea, more so an algorithm, there are many ways one is able to correctly implement it. Prior to the routine being used in production, two questions must be asked:

1. Does the implementation produce the correct results?
2. What implementation of the routine is the best?

The first question is subject to a binary pass-fail unit test verification while the latter question is where the details of an implementation are scrutinized to extract maximal efficiency from the routine. The quality of the best routine follows first and foremost from its correctness. To that end, R offers many different unit testing frameworks such as RUnit by Burger et al. (2015), which is used to construct Rcpp’s 1385+ unit tests, and testthat by Wickham (2011). Only when correctness is achieved is it wise to begin the procedure of optimizing the efficiency of the routine and, in turn, selecting the best routine.

Optimization of an algorithm involves performing a quantitative analysis of the routine’s properties. There are two main approaches to analyzing the behavior of a routine: theoretical analysis¹ or an empirical examination using profiling tools.² Typically, the latter option is more prominently used as the routine’s theoretical properties are derived prior to an implementation being started. Often the main concern regarding an implementation in R relates to the speed of the algorithm as it impacts how quickly analyses can be done and reports can be provided to decision makers. Coincidently, the speed of code is one of the key governing use cases of Rcpp. Profiling R code will reveal shortcomings related to loops, e.g. for, while, and repeat; conditional statements, e.g. if-else

¹Theoretical analysis is often directed to describing the limiting behavior of a function through asymptotic notation, commonly referred to as Big O and denoted as $O()$.

²Within base R, profiling can be activated by using Rprof() for individual command timing information, Rprof() for memory information, and Sys.time() for a quick overall execution timing. Additional profiling R packages such as profvis by Chang and Luraschi (2017), Rprof by Tandon and Hocking (2015), and benchmarking packages have extended the ability to analyze performance.
if-else and switch; and recursive functions, i.e. a function written
in terms of itself such that the problem is broken down on each
call in a reduced state until an answer can be obtained. In contrast,
the overhead for such operations is significantly less in C++. Thus,
critical components of a given routine should be written in Rcpp to
capture maximal efficiency.

Returning to the second question, to decide which implementation
works the best, one needs to employ a benchmark to obtain quantifiable results. Benchmarks are an ideal way to quantify how
well a method performs because they have the ability to show the
amount of time the code has been running and where bottlenecks
exist within functions. This does not imply that benchmarks are
completely infallible as user error can influence the end results.
For example, if a user decides to benchmark code in one R session
and in another session performs a heavy computation, then the
benchmark will be biased (if “wall clock” is measured).

There are different levels of magnification that a benchmark can
provide. For a more macro analysis, one should benchmark data using benchmark(test = func(), test2 = func2()), a function from the rbenchmark R package by Kusnierczyk (2012). This form of benchmarking will be used when the computation is
more intensive. The motivating example isOdd() (which is only
able to accept a single integer) warrants a much more microscopic
timing comparison. In cases such as this, the objective is to obtain
precise results in the amount of nanoseconds elapsed. Using the
microbenchmark function from the microbenchmark R package by Mersmann (2015) is more helpful to obtain timing information.
To perform the benchmark:

```r
library("microbenchmark")
results <- microbenchmark(isOddR = isOddR(12L),
 isOddCpp = isOddCpp(12L))
# print(summary(results)[, c(1:7)], digits=1)
```

By looking at the summary of 100 evaluations, we note that
the Rcpp function performed better than the equivalent in R by
achieving a lower run time on average. The lower run time in this
part is not necessarily critical as the difference is nanoseconds on a
trivial computation. However, each section of code does contribute
to a faster overall runtime.

7. Random Numbers within Rcpp: An Example of Rcpp Sugar

Rcpp connects R with C++. Only the former is vectorized: C++ is not.
Rcpp Sugar, however, provides a convenient way to work with
high-performing C++ functions in a similar way to how R offers
vectorized operations. The Rcpp Sugar vignette (Eddelbuettel and
François, 2017b) details these, as well as many more functions
directly accessible to Rcpp in a way that should feel familiar to R users. Some examples of Rcpp Sugar functions include special
math functions like gamma and beta, statistical distributions and
random number generation.

We will illustrate a case of random number generation. Consider
drawing one or more N(0,1)-distributed random variables. The
very simplest case can just use evalCpp():

```r
evalCpp("R::rnorm(0, 1)"")
```

# [1] -0.4642687

By setting a seed, we can make this reproducible:

```r
set.seed(123)
evalCpp("R::rnorm(0, 1)"")
```

# [1] -0.5604756

One important aspect of the behind-the-scenes code generation
for the single expression (as well as all code created via Rcpp Attributes) is the automatic preservation of the state of the random
number generators in R. This means that from a given seed, we
will receive identical draws of random numbers whether we access
them from R or via C++ code accessing the same generators (via
the Rcpp interfaces). To illustrate, the same number is drawn via R
code after resetting the seed:

```r
set.seed(123)
# Implicit mean of 0, sd of 1
rnorm(1)
```

# [1] -0.5604756

We can make the Rcpp Sugar function rnorm() accessible from
R in the same way to return a vector of values:

```r
set.seed(123)
evalCpp("Rcpp::rnorm(3)"")
```

# [1] -0.5604756 -0.2301775 1.5587083

Note that we use the Rcpp:: namespace explicitly here to contrast the vectorised Rcpp::rnorm() with the scalar R::rnorm() also
provided as a convenience wrapper for the C API of R.

And as expected, this too replicates from R as the very same
generators are used in both cases along with consistent handling
of generator state permitting to alternate:

```r
set.seed(123)
R::rnorm(3)
```

# [1] -0.5604756 -0.2301775 1.5587083

8. Translating Code from R into Rcpp: A Bootstrapping Example

Statistical inference relied primarily upon asymptotic theory until
Efron (1979) proposed the bootstrap. Bootstrapping is known to
be computationally intensive due to the need to use loops. Thus, it
is an ideal candidate to use as an example. Before starting to write
C++ code using Rcpp, prototype the code in R.

```r
# Function declaration
bootstrap_r <- function(ds, B = 1000) {

  # Preallocate storage for statistics
  boot_stat <- matrix(NA, nrow = B, ncol = 2)

  # Number of observations
  n <- length(ds)

  # Perform bootstrap
  for(i in seq_len(B)) {
    # Sample initial data
    gen_data <- ds[ sample(n, n, replace=TRUE) ]
    # Calculate sample data mean and SD
    boot_stat[i,] <- c(mean(gen_data),
```
In the Rcpp version of the bootstrap function, there are a few additional changes that occurred during the translation. In particular, the use of Rcpp::runif(n, 0, n) enclosed by floor(), which rounds down to the nearest integer, in place of sample(n, n, replace = TRUE) to sample row ids. This is an equivalent substitution since equal weight is being placed upon all row ids and replacement is allowed.\footnote{For more flexibility in sampling see Christian Gunning’s Sample extension for RcppArmadillo and Rcpp Gallery: Using the RcppArmadillo-based Implementation of R’s sample() or consider using the Rcpp::sample() sugar function added in 0.12.9 by Nathan Russell.} Note that the upper bound of the interval, n, will never be reached. While this may seem flawed, it is important to note that vectors and matrices in C++ use a zero-based indexing system, meaning that they begin at 0 instead of 1 and go up to n – 1 instead of n, which is unlike R’s system. Thus, an out of bounds error would be triggered if n was used as that point does not exist within the data structure. The application of this logic can be seen in the span the for loop takes in C++ when compared to R. Another syntactical change is the use of () in place of [] while accessing the matrix. This change is due to the governance of C++ and its comma operator making it impossible to place multiple indices inside the square brackets.

To validate that the translation was successful, first run the C++ function under the same data and seed as was given for the R function.

```r
# Use the same seed use in R and C+
set.seed(883)
# Perform bootstrap with C++ function
result_cpp <- bootstrap_cpp(initdata)
```

Next, check the output between the functions using R’s all.equal() function that allows for an $\epsilon$-neighborhood around a number.

```r
# Compare output
all.equal(result_r, result_cpp)
```

\[ 1 \] TRUE

Lastly, make sure to benchmark the newly translated Rcpp function against the R implementation. As stated earlier, data is paramount to making a decision related to which function to use in an analysis or package.

```r
library(rbenchmark)

benchmark(r = bootstrap_r(initdata),
          cpp = bootstrap_cpp(initdata))[, 1:4]
```

<table>
<thead>
<tr>
<th>test replications elapsed relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>


Many of the previously illustrated Rcpp examples were directed primarily to show the gains in computational efficiency that are possible by implementing code directly in C++; however, this is only one potential application of Rcpp. Perhaps one of the most understated features of Rcpp is its ability to enable Chambers (2016)’s third statement of Interfaces to other software are part of R. In particular, Rcpp is designed to facilitate interfacing libraries written in C++ or C to R. Hence, if there is a specific feature within a C++ or C library, then one can create a bridge to it using Rcpp to enable it from within R.

An example is the use of C++ matrix algebra libraries like Armadillo (Sanderson and Curtin, 2016) or Eigen (Guennebaud et al., 2012). By outsourcing complex linear algebra operations to matrix libraries, the need to directly call functions within Linear

For more flexibility in sampling see Christian Gunning’s Sample extension for RcppArmadillo and Rcpp Gallery: Using the RcppArmadillo-based Implementation of R’s sample() or consider using the Rcpp::sample() sugar function added in 0.12.9 by Nathan Russell.
Fig. 2. Results of the bootstrapping procedure for sample mean and variance.

Algebra PACKage (LAPACK) (Anderson et al., 1999) is negated. Moreover, the Rcpp design allows for seamless transfer between object types by using automatic converters governed by `wrap()`, C++ to R, and as<T>(), R to C++ with the T indicating the type of object being cast into. These two helper functions provide a non-invasive way to work with an external object. Thus, a further benefit to using external C++ libraries is the ability to have a portable code base that can be implemented within a standalone C++ program or within another computational language.

Compute RNG draws from a multivariate Normal. A common application in statistical computing is simulating from a multivariate normal distribution. The algorithm relies on a linear transformation of the standard Normal distribution. Letting $Y_{m \times 1} = A_{m \times n} Z_{n \times 1} + b_{m \times 1}$, where $A$ is a $m \times n$ matrix, $b \in \mathbb{R}^m$, $Z \sim N(0, I_n)$, and $I_n$ is the identity matrix, then $Y \sim N_m(\mu = b, \Sigma = AA^T)$. To obtain the matrix $A$ from $\Sigma$, either a Cholesky or Eigen decomposition is required. As noted in ?, the Eigen decomposition is more stable in addition to being more computationally demanding compared to the Cholesky decomposition. For simplicity and speed, we have opted to implement the sampling procedure using a Cholesky decomposition. Regardless, there is a need to involve one of the above matrix libraries to make the sampling viable in C++.

Here, we demonstrate how to take advantage of the Armadillo linear algebra template classes (Sanderson and Curtin, 2016) via the RcppArmadillo package (Eddelbuettel and Sanderson, 2014; Eddelbuettel et al., 2016). Prior to running this example, the RcppArmadillo package must be installed using `install.packages("RcppArmadillo")`.\footnote{MacOS users may encounter `-lgfortran` and `-lquadmath` errors on compile with this package if the development environment is not appropriately setup. Section 2.16 of Rcpp FAQ provides details regarding the necessary gfortran binaries.}

```cpp
#include <RcppArmadillo.h> // 
[[Rcpp::depends(RcppArmadillo)]]
```

With this in mind, sampling from a multivariate normal distribution can be obtained in a straightforward manner. Using only Armadillo data types and values:

```cpp
#include <RcppArmadillo.h> // 
[[Rcpp::depends(RcppArmadillo)]]
```

```cpp
arma::mat rmvnorm(int n, 
const arma::vec& mu, 
const arma::mat& Sigma) { 
  unsigned int p = Sigma.n_cols;
```
As a result of using a random number generation (RNG), there is an additional requirement to ensure reproducible results: the necessity to explicitly set a seed (as shown above). Because of the (programmatic) interface provided by R to its own RNGs, this setting of the seed has to occur at the R level via the set.seed() function as no (public) interface is provided by the R header files.

### Faster linear model fits

As a second example, consider the problem of estimating a common linear model repeatedly. One use case might be the simulation of size and power of standard tests. Many users of R would default to using lm(), however, the overhead associated with this function greatly impacts speed with which an estimate can be obtained. Another approach would be to take the base R function lm.fit(), which is called by lm(), to compute estimated \( \hat{\beta} \) in just about the fastest time possible. However, this approach is also not viable as it does not report the estimated standard errors. As a result, we cannot use any default R functions in the context of simulating finite sample population effects on inference.

One alternative is provided by the fastlm() function in RcppArmadillo (Eddelbuettel et al., 2016).

```cpp
#include <RcppArmadillo.h>

// First draw N x P values from a N(0,1)
Rcpp::NumericVector draw = Rcpp::rnorm(n*p);

// Instantiate an Armadillo matrix with the
// drawn values using advanced constructor
// to reuse allocated memory
arma::mat Z = arma::mat(draw.begin(), n, p,
false, true);

// Simpler, less performant alternative
// arma::mat Z = Rcpp::as<arma::mat>(draw);

// Generate a sample from the Transformed
// Multivariate Normal
arma::mat Y = arma::repmat(mu, 1, n).t() +
Z * arma::chol(Sigma);

return Y;
```

The interface is very simple: a matrix \( X_{n \times p} \) of regressors, and a dependent variable \( y_{n \times 1} \) as a vector. We invoke the standard Armadillo function solve() to fit the model \( y \sim X \). We then compute residuals, and extract the (appropriately scaled) diagonal of the covariance matrix, also taking its square root, in order to return both estimates \( \hat{\beta} \) and \( \hat{\sigma} \).

### 10. Rcpp in Packages

Once a project containing compiled code has matured to the point of sharing it with collaborators\(^6\) or using it within a parallel computing environments, the ideal way forward is to embed the code within an R package. Not only does an R package provide a way to automatically compile source code, but also enables the use of the R help system to document how the written functions should be used. As a further benefit, the package format enables the use of unit tests to ensure that the functions are producing the correct output. Lastly, having a package provides the option of uploading to a repository such as CRAN for wider dissemination.

To facilitate package building, Rcpp provides a function Rcpp.package.skeleton() that is modeled after the base R function package.skeleton(). This function automates the creation of a skeleton package appropriate for distributing Rcpp.

```cpp
library("Rcpp")
Rcpp.package.skeleton("samplePkg")
```

This shows how distinct directories `man`, `R`, `src` are created for, respectively, the help files, pages with R code and files with C++ code. Generally speaking, all compiled code, be it from C, C++ or Fortran sources, should be placed within the `src` directory.

Alternatively, one can achieve similar results to using Rcpp.package.skeleton() by using a feature of the RStudio IDE. Specifically, while creating a new package project there is an option to select the type of package by engaging a dropdown menu to select “Package w/ Rcpp” in RStudio versions prior to v1.1.0. In RStudio versions later than v1.1.0, support for package templates has been added allowing users to directly create Rcpp-based packages that use Eigen or Armadillo.

---

\(^6\)We should note that this will use the standard LAPACK functionality via Armadillo whereas R uses an internal refinement of LINPACK (Dongarra et al., 1979) via pivoting, rendering the operation numerically more stable. That is an important robustness aspect—though common datasets on current hardware almost never lead to actual differences. That said, if in doubt, stick with the R implementation. What is shown here is mostly for exposition of the principles.

\(^7\)It is sometimes said that every project has two collaborators: self, and future self. Packaging code is best practices even for code not intended for public uploading.
Lastly, one more option exists for users who are familiar with the devtools R package by Wickham and Chang (2016). To create the R package skeleton use devtools::create("samplePkg"). From here, part of the structure required by Rcpp can be added by using devtools::use_rcpp(). The remaining aspects needed by Rcpp must be manually copied from the roxygen tags written to console and pasted into one of the package’s R files to successfully incorporate the dynamic library and link to Rcpp’s headers.

All of these methods take care of a number of small settings one would have to enable manually otherwise. These include an ‘Imports:’ and ‘LinkingTo:’ declaration in file DESCRIPTION, as well as ‘useDynLib’ and ‘importFrom’ in NAMESPACE. For Rcpp Attributes use, the compileAttributes() function has to be called. Similarly, to take advantage of its documentation-creation feature, the roxygenize() function from roxygen2 has to be called. Additional details on using Rcpp within a package scope are detailed in Eddelbuettel and François (2017c).

Fig. 3. Graphical annotation of the is_odd.cpp function.

11. Conclusion

R has always provided mechanisms to extend it. The bare-bones C API is already used to great effect by a large number of packages. By taking advantage of a number of C++ features, Rcpp has been able to make extending R easier, offering a combination of both speed and ease of use that has been finding increasingly widespread utilization by researchers and data scientists. We are thrilled about this adoption, and look forward to seeing more exciting extensions to R being built.

Acknowledgments. We thank Bob Rudis and Lionel Henry for comments and suggestion on an earlier draft of this manuscript. Furthermore, we appreciate the improved C++ annotated function graphic provided by Bob Rudis.

References


The R package Rcpp has always provided mechanisms to extend it. The bare-bones C API is already used to great effect by a large number of packages. By taking advantage of a number of C++ features, Rcpp has been able to make extending R easier, offering a combination of both speed and ease of use that has been finding increasingly widespread utilization by researchers and data scientists. We are thrilled about this adoption, and look forward to seeing more exciting extensions to R being built.

Acknowledgments. We thank Bob Rudis and Lionel Henry for comments and suggestion on an earlier draft of this manuscript. Furthermore, we appreciate the improved C++ annotated function graphic provided by Bob Rudis.


