Rcpp: Seamless R and C++ integration

by Dirk Eddelbuettel and Romain François

Abstract The **Rcpp** package simplifies integrating C++ code with R. It provides a consistent C++ class hierarchy that maps various types of R objects (vectors, functions, environments, ...) to dedicated C++ classes. Object interchange between R and C++ is managed by simple, flexible and extensible concepts which include broad support for C++ STL idioms. C++ code can be compiled, linked and loaded on the fly. Flexible error and exception code handling is provided. **Rcpp** substantially lowers the barrier for programmers wanting to combine C++ code with R.

Introduction

R is an extensible system. The 'Writing R Extensions' manual (R Development Core Team, 2009a) describes in detail how to augment R with compiled code, focussing mostly on the C language. The R API described in 'Writing R Extensions' is based on a set of functions and macros operating on `SEXP`, the internal representation of R objects. In this article, we discuss the functionality of the **Rcpp** package, which simplifies the usage of C++ code in R. Combining R and C++ is not a new idea, so we start with a short review of other approaches and give some historical background on the development of **Rcpp**.

The current version of **Rcpp** combines two distinct APIs. The first—which we call 'classic **Rcpp** API'—exists since the first version of **Rcpp**. The second API, enclosed in the `Rcpp` C++ namespace, is a newer codebase which we started to develop more recently. This article highlights some of the key design and implementation choices: lightweight encapsulation of R objects in C++ classes, automatic garbage collection strategy, code inlining, data interchange between R and C++ and error handling.

Several examples are included to illustrate the benefits of using **Rcpp** as opposed to the traditional R API. Many more examples are available within the package, both as explicit examples and as part of the numerous unit tests.

Historical Context

Rcpp first appeared in 2005 as a contribution to the **RQuantLib** package (Eddelbuettel and Nguyen, 2009) before becoming a CRAN package in early 2006. Several releases followed in quick succession; all of these were under the name **Rcpp**. The package was then renamed to **RcppTemplate** and several more releases followed during 2006 under the new name. However, no new releases or updates were made during 2007, 2008 and most of 2009.

Given the continued use of the package, it was revitalized. New releases, using the initial name **Rcpp**, started in November 2008. These already included an improved build and distribution process, additional documentation, and new functionality—while retaining the existing interface. This constitutes the 'classic **Rcpp**' interface (described in the next section) which will be maintained for the foreseeable future.

Yet C++ coding standards continued to evolve (Meyers, 2005). So starting in 2009 the codebase was significantly extended and numerous new features were added. Several of these are described below in the section on the 'new **Rcpp**' interface. This new API is our current focus, and we intend to both extend and support it going forward.

Comparison

Integration of C++ and R has been addressed by several authors; the earliest published reference is probably Bates and DebRoy (2001). An unpublished paper by Java, Gaile, and Manly (2007) expresses several ideas that are close to some of our approaches, though not yet fully fleshed out. The **Rserve** package (Urbanek, 2009) was another early approach, going back to 2002. On the server side, **Rserve** translates R data structures into a binary serialization format and uses TCP/IP for transfer. On the client side, objects are reconstructed as instances of Java or C++ classes that emulate the structure of R objects.

The packages **rcppbind** (Liang, 2008), **RAbstraction** (Armstrong, 2009a) and **RObjects** (Armstrong, 2009b) are all implemented using C++ templates. However, neither has matured to the point of a CRAN release and it is unclear how much usage these packages are seeing beyond their own authors. CXXR (Runnalls, 2009) comes to this topic from the other side: its aim is to completely refactor R on a stronger C++ foundation. CXXR is therefore concerned with all aspects of the R interpreter, REPL loop, threading—and object interchange between R and C++ is but one part. A similar approach is discussed by Temple Lang (2009a) who suggests making low-level internals extensible by package developers in order to facilitate extending R. Another slightly different angle is offered by Temple Lang (2009b) who uses compiler output for references on the code in order to add bindings and wrappers. Lastly, the **RcppTemplate** package (Samperi, 2009) recently introduced a few new ideas yet decided to break with the 'classic **Rcpp**' API.

A critical comparison of these packages that addresses relevant aspects such API features, performance, usability and documentation would be a welcome addition to the literature, but is beyond the scope of this article.

Classic Rcpp API

The core focus of **Rcpp**—particularly for the earlier API described in this section—has always been on allowing the programmer to add C++-based functions. We use this term in the standard mathematical sense of providing results (output) given a set of parameters or data (input). This was facilitated from the earliest releases using C++ classes for receiving various types of R objects, converting them to C++ objects and allowing the programmer to return the results to R with relative use.

This API therefore supports two typical use cases. First, one can think of replacing existing R code with equivalent C++ code in order to reap performance gains. This case is conceptually easy as there may not be (built- or run-time) dependencies on other C or C++ libraries. It typically involves setting up data and parameters—the right-hand side components of a function call—before making the call in order to provide the result that is to be assigned to the left-hand side. Second, **Rcpp** facilitates calling functions provided by other libraries. The use resembles the first case: data and parameters are passed via **Rcpp** to a function set-up to call code from an external library.

An illustration can be provided using the time-tested example of a convolution of two vectors. This example is shown in sections 5.2 (for the `.C()` interface) and 5.9 (for the `.Call()` interface) of 'Writing R Extensions' (R Development Core Team, 2009a). We have rewritten it here using classes of the classic **Rcpp** API:

```
#include <Rcpp.h>

RcppExport SEXP convolve2cpp(SEXP a,SEXP b) {
  RcppVector<double> xa(a);
  RcppVector<double> xb(b);
  int nab = xa.size() + xb.size() - 1;

  RcppVector<double> xab(nab);
  for (int i = 0; i < nab; i++) xab(i) = 0.0;

  for (int i = 0; i < xa.size(); i++)
    for (int j = 0; j < xb.size(); j++)
       xab(i + j) += xa(i) * xb(j);

  RcppResultSet rs;
  rs.add("ab", xab);
  return rs.getReturnList();
}
```

We can highlight several aspects. First, only a single header file `Rcpp.h` is needed to use the **Rcpp** API. Second, given two `SEXP` types, a third is returned. Third, both inputs are converted to templated[1] C++ vector types, here a standard `double` type is used to create a vector of doubles from the template type. Fourth, the usefulness of these classes can be seen when we query the vectors directly for their size—using the `size()` member function—in order to reserve a new result type of appropriate length whereas use based on C arrays would have required additional parameters for the length of vectors a and b, leaving open the possibility of mismatches between the actual length and the length reported by the programmer. Fifth, the computation itself is straightforward embedded looping just as in the original examples in the 'Writing R Extensions' manual (R Development Core Team, 2009a). Sixth, a return type (`RcppResultSet`) is prepared as a named object which is then converted to a list object that is returned. We should note that the `RcppResultSet` supports the return of numerous (named) objects which can also be of different types.

We argue that this usage is already much easier to read, write and debug than the C macro-based approach supported by R itself. Possible performance issues and other potential limitations will be discussed throughout the article and reviewed at the end.

New Rcpp API

More recently, the **Rcpp** API has been dramatically extended, leading to a complete redesign, based on the usage experience of several years of **Rcpp** deployment, needs from other projects, knowledge of the internal R API, as well as current C++ design approaches. This redesign of `Rcpp` was also motivated by the needs of other projects such as `RInside` (Eddelbuettel and François, 2010) for easy embedding of R in a C++ applications and `RProtoBuf` (François and Eddelbuettel, 2010) that interfaces with the protocol buffers library.

Rcpp Class hierarchy

The `Rcpp::RObject` class is the basic class of the new **Rcpp** API. An instance of the `RObject` class encapsulates an R object (`SEXP`), exposes methods that are appropriate for all types of objects and transparently manages garbage collection.

The most important aspect of the `RObject` class is that it is a very thin wrapper around the `SEXP` it encapsulates. The `SEXP` is indeed the only data member of an `RObject`. The `RObject` class does not interfere with the way R manages its memory, does not perform copies of the object into a suboptimal C++ representation, but instead merely acts as a proxy to the object it encapsulates so that methods applied to the `RObject` instance are relayed back to the `SEXP` in terms of the standard R API.

[1] C++ templates allow functions or classes to be written somewhat independently from the template parameter. The actual class is instantiated by the compiler by replacing occurrences of the templated parameter(s).

The `RObject` class takes advantage of the explicit life cycle of C++ objects to manage exposure of the underlying R object to the garbage collector. The `RObject` effectively treats its underlying `SEXP` as a resource. The constructor of the `RObject` class takes the necessary measures to guarantee that the underlying `SEXP` is protected from the garbage collector, and the destructor assumes the responsibility to withdraw that protection.

By assuming the entire responsibility of garbage collection, `Rcpp` relieves the programmer from writing boiler plate code to manage the protection stack with `PROTECT` and `UNPROTECT` macros.

The `RObject` class defines a set of member functions applicable to any R object, regardless of its type. This ranges from querying properties of the object (`isNULL`, `isObject`, `isS4`), management of the attributes (`attributeNames`, `hasAttribute`, `attr`) and handling of slots[2] (`hasSlot`, `slot`).

Derived classes

Internally, an R object must have one type amongst the set of predefined types, commonly referred to as SEXP types. R internals (R Development Core Team, 2009b) documents these various types. **Rcpp** associates a dedicated C++ class for most SEXP types, therefore only exposes functionality that is relevant to the R object that it encapsulates.

For example `Rcpp::Environment` contains member functions to manage objects in the associated environment. Similarly, classes related to vectors (`IntegerVector`, `NumericVector`, `RawVector`, `LogicalVector`, `CharacterVector`, `GenericVector` and `ExpressionVector`) expose functionality to extract and set values from the vectors.

The following sub-sections present typical uses of **Rcpp** classes in comparison with the same code expressed using functions of the R API.

Numeric vectors

The following code snippet is taken from Writing R extensions (R Development Core Team, 2009a). It creates a `numeric` vector of two elements and assigns some values to it.

```
SEXP ab;
PROTECT(ab = allocVector(REALSXP, 2));
REAL(ab)[0] = 123.45;
REAL(ab)[1] = 67.89;
UNPROTECT(1);
```

Although this is one of the simplest examples in Writing R extensions, it seems verbose and it is not obvious at first sight to understand what is happening. Memory is allocated by `allocVector`; we must also supply it with the type of data (`REALSXP`) and the number of elements. Once allocated, the `ab` object must be protected from garbage collection[3]. Lastly, the `REAL` macro returns a pointer to the beginning of the actual array; its indexing does not resemble either R or C++.

Using the `Rcpp::NumericVector` class, the code can be rewritten:

```
Rcpp::NumericVector ab(2) ;
ab[0] = 123.45;
ab[1] = 67.89;
```

The code contains fewer idiomatic decorations. The `NumericVector` constructor is given the number of elements the vector contains (2), this hides a call to the `allocVector` we saw previously. Also hidden is protection of the object from garbage collection, which is a behavior that `NumericVector` inherits from `RObject`. Values are assigned to the first and second elements of the vector as `NumericVector` overloads the `operator[]`.

With the most recent compilers (e.g. GNU g++ >= 4.4) which already implement parts of the next C++ standard (C++0x) currently being drafted, the preceding code may even be reduced to this:

```
Rcpp::NumericVector ab = {123.45, 67.89};
```

Character vectors

A second example deals with character vectors and emulates this R code

```
> c("foo", "bar")
```

Using the traditional R API, the vector can be allocated and filled as such:

```
SEXP ab;
PROTECT(ab = allocVector(STRSXP, 2));
SET_STRING_ELT( ab, 0, mkChar("foo") );
SET_STRING_ELT( ab, 1, mkChar("bar") );
UNPROTECT(1);
```

This imposes on the programmer knowledge of `PROTECT`, `UNPROTECT`, `SEXP`, `allocVector`, `SET_STRING_ELT`, and `mkChar`.

Using the **Rcpp::CharacterVector** class, we can express the same code more concisely:

```
CharacterVector ab(2) ;
ab[0] = "foo" ;
ab[1] = "bar" ;
```

R and C++ data interchange

In addition to classes, the **Rcpp** package contains two functions to perform conversion of C++ objects to R objects and back.

[2]The member functions that deal with slots are only applicable on S4 objects; otherwise an exception is thrown.

[3]Since the garbage collection can be triggered at any time, not protecting an object means its memory might be reclaimed too soon.

C++ to R : wrap

The C++ to R conversion is performed by the `Rcpp::wrap` templated function. It uses advanced template meta programming techniques[4] to convert a wide and extensible set of types and classes to the most appropriate type of R object. The signature of the `wrap` template is:

```
template <typename T>
SEXP wrap(const T& object) ;
```

The templated function takes a reference to a 'wrappable' object and converts this object into a `SEXP`, which is what R expects. Currently wrappable types are :

- primitive types, `int`, `double`, ... which are converted into the corresponding atomic R vectors;
- `std::string` which are converted to R atomic character vectors;
- STL containers such as `std::vector<T>` or `std::list<T>`, as long as the template parameter type `T` is itself wrappable;
- STL maps which use `std::string` for keys (e.g. `std::map<std::string,T>`); as long as the type `T` is wrappable;
- any type that implements implicit conversion to `SEXP` through the `operator SEXP();`
- any type for which the `wrap` template is partially or fully specialized.

Wrappability of an object type is resolved at compile time using modern techniques of template meta programming and class traits.

The following code snippet illustrates that the design allows composition:

```
std::vector< std::map<std::string,int> > v;
std::map< std::string, int > m1;
std::map< std::string, int > m2;

m1["foo"] = 1; m1["bar"] = 2;
m2["foo"] = 1; m2["bar"] = 2; m2["baz"] = 3;

v.push_back( m1) ;
v.push_back( m2) ;
Rcpp::wrap( v ) ;
```

The code creates a list of two named vectors, equal to the result of this R statement:

```
list( c( bar = 2L, foo = 1L) ,
      c( bar = 2L, baz = 3L, foo = 1L) )
```

R to C++ : as

The reversed conversion is implemented by variations of the `Rcpp::as` template. It offers less flexibility and currently handles conversion of R objects into primitive types (bool, int, std::string, ...), STL vectors of primitive types (`std::vector<bool>`, `std::vector<double>`, etc ...) and arbitrary types that offer a constructor that takes a `SEXP`. In addition as can be fully or partially specialized to manage conversion of R data structures to third-party types.

Implicit use of converters

The converters offered by `wrap` and `as` provide a very useful framework to implement the logic of the code in terms of C++ data structures and then explicitly convert data back to R.

In addition, the converters are also used implicitly in various places in the `Rcpp` API. Consider the following code that uses the `Rcpp::Environment` class to interchange data between C++ and R.

```
// assuming the global environment contains
// a variable 'x' that is a numeric vector
Rcpp::Environment global =
    Rcpp::Environment::global_env()

// extract a std::vector<double> from
// the global environment
std::vector<double> vx = global["x"] ;

// create a map<string,string>
std::map<std::string,std::string> map ;
map["foo"] = "oof" ;
map["bar"] = "rab" ;

// push the STL map to R
global["y"] = map ;
```

In the first part of the example, the code extracts a `std::vector<double>` from the global environment. This is achieved by the templated `operator[]` of `Environment` that first extracts the requested object from the environment as a `SEXP`, and then outsources to `Rcpp::as` the creation of the requested type.

In the second part of the example, the `operator[]` delegates to wrap the production of an R object based on the type that is passed in (`std::map<std::string,std::string>`), and then assigns the object to the requested name.

The same mechanism is used throughout the API. Examples include access/modification of object attributes, slots, elements of generic vectors (lists), function arguments, nodes of dotted pair lists and language calls and more.

[4] A discussion of template meta programming is beyond the scope of this article.

Environment: Using the **Rcpp** API

```
Environment stats("package:stats");
Function rnorm = stats["rnorm"];
return rnorm(10, Named("sd", 100.0));
```

Environment: Using the R API

```
SEXP stats = PROTECT(
  R_FindNamespace( mkString("stats")));
SEXP rnorm = PROTECT(
  findVarInFrame( stats, install("rnorm")));
SEXP call  = PROTECT(
  LCONS( rnorm,
    CONS(ScalarInteger(10),
      CONS(ScalarReal(100.0),R_NilValue))));
SET_TAG( CDDR(call), install("sd") );
SEXP res = PROTECT(eval(call, R_GlobalEnv));
UNPROTECT(4) ;
return res ;
```

Language: Using the **Rcpp** API

```
Language call("rnorm",10,Named("sd",100.0));
return call.eval();
```

Language: Using the R API

```
SEXP call  = PROTECT(
  LCONS( install("rnorm"),
    CONS(ScalarInteger(10),
      CONS(ScalarReal(100.0),R_NilValue))));
SET_TAG( CDDR(call), install("sd") ) ;
SEXP res = PROTECT(eval(call, R_GlobalEnv));
UNPROTECT(2) ;
return res ;
```

Table 1: **Rcpp** versus the R API: Four ways of calling `rnorm(10L, sd=100)` in C / C++. We have removed the `Rcpp::` prefix from the examples for readability; this corresponds to adding a statement `using namespace Rcpp;` in the code

Function calls

The next example shows how to use **Rcpp** to emulate the R code `rnorm(10L, sd=100.0)`. As shown in table 1, the code can be expressed in several ways in either **Rcpp** or the standard R API. The first version shows the use of the `Environment` and `Function` classes by **Rcpp**. The second version shows the use of the `Language` class, which manage calls (LANGSXP). For comparison, we also show both versions using the standard R API.

This example illustrates that the **Rcpp** API permits us to work with code that is easier to read, write and maintain. More examples are available as part of the documentation included in the **Rcpp** package, as well as among its over one hundred and ninety unit tests.

Using code 'inline'

Extending R with compiled code also needs to address how to reliably compile, link and load the code. While using a package is preferable in the long run, it may be too involved for for quick explorations. An alternative is provided by the **inline** package (Sklyar, Murdoch, Smith, and Eddelbuettel, 2009) which compiles, links and loads a C, C++ or Fortran function—directly from the R prompt using a simple function `cfunction`. It was recently extended to work with **Rcpp** by allowing for the use of additional header files and libraries. This works particularly well with the **Rcpp** package where headers and the library are automatically found if the appropriate option `Rcpp` to `cfunction` is set to `TRUE`.

The use of **inline** is possible as **Rcpp** can be installed and updated just like any other R package using *e.g.* the `install.packages()` function for initial installation as well as `update.packages()` for upgrades. So even though R / C++ interfacing would otherwise require source code, the **Rcpp** library is always provided ready for use as a pre-built library through the CRAN package mechanism.[5]

The library and header files provided by **Rcpp** for use by other packages are installed along with the **Rcpp** package making it possible for **Rcpp** to provide the appropriate `-I` and `-L` switches needed for compilation and linking. So internally, **inline** makes uses of the two functions `Rcpp:::CxxFlags()` and `Rcpp:::LdFlags()` that provide this information (and which are also used by `Makevars` files of other packages). Here, however, all this is done behind the

[5] This presumes a platform for which pre-built binaries are provided. **Rcpp** is available in binary form for Windows and OS X users from CRAN, and as a `.deb` package for Debian and Ubuntu users. For other systems, the **Rcpp** library is automatically built from source during installation or upgrades.

scenes without the need for explicitly setting compiler or linker options.

The convolution example provided above can be rewritten for use by **inline** as shown below. The function body is provided by the character variable `src`, the function header is defined by the argument `signature`—and we only need to enable `Rcpp=TRUE` to obtain a new function `fun` based on the C++ code in `src` where we also switched from the classic **Rcpp** API to the new one:

```
src <- '
  Rcpp::NumericVector xa(a);
  Rcpp::NumericVector xb(b);
  int n_xa = xa.size(), n_xb = xb.size();

  Rcpp::NumericVector xab(n_xa + n_xb - 1);
  for (int i = 0; i < n_xa; i++)
    for (int j = 0; j < n_xb; j++)
       xab[i + j] += xa[i] * xb[j];
  return xab;
'
fun <- cfunction(
    signature(a="numeric", b="numeric"),
    src, Rcpp=TRUE)
```

The main difference to the previous solution is that the input parameters are directly passed to types `Rcpp::NumericVector`, and that the return vector is automatically converted to a `SEXP` type through implicit conversion. Also in this version, the vector `xab` is not initialized because the constructor already performs initialization to match the behavior of the R function `numeric`.

Using STL algorithms

The C++ Standard Template Library (STL) offers a variety of generic algorithms designed to be used on ranges of elements (Plauger, Stepanov, Lee, and Musser, 2000). A range is any sequence of objects that can be accessed through iterators or pointers. All **Rcpp** classes from the new API representing vectors (including lists) can produce ranges through their member functions `begin()` and `end()`, effectively supporting iterating over elements of an R vector.

The following code illustrates how **Rcpp** might be used to emulate a simpler[6] version of `lapply` using the `transform` algorithm from the STL.

```
src <- '
  Rcpp::List input(data);
  Rcpp::Function f(fun) ;
  Rcpp::List output(input.size());
  std::transform(
   input.begin(), input.end(),
   output.begin(),
   f ) ;
  output.names() = input.names() ;
  return output ;
  '
cpp_lapply <- cfunction(
  signature(data="list", fun = "function"),
  src, Rcpp = TRUE )
```

We can use this to calculate a summary of each column of the `faithful` dataset included with R.

```
> cpp_lapply( faithful, summary )
$eruptions
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
1.600   2.163   4.000   3.488   4.454   5.100

$waiting
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 43.0    58.0    76.0    70.9    82.0    96.0
```

Error handling

Code that uses both R and C++ has to deal with two concurrent error handling models. **Rcpp** simplifies this and allows both systems to work together.

C++ exceptions in R

The traditional way of dealing with C++ exceptions in R is to catch them through explicit try/catch blocks and convert this exception into an R error manually.

In C++, when an application throws an exception that is not caught, a special function (called the terminate handler) is invoked. This typically causes the program to abort. **Rcpp** takes advantage of this mechanism and installs its own terminate handler which translates C++ exceptions into R conditions. The following code gives an illustration.

```
> fun <- cfunction(signature(x = "integer"),'
+  int dx = as<int>(x) ;
+   if( dx > 10 )
+      throw std::range_error("too big") ;
+   return wrap(dx*dx) ;
+ ', Rcpp = TRUE,
+  includes = "using namespace Rcpp;" )
> tryCatch( fun(12),
+ "std::range_error" = function(e)
+    writeLines( conditionMessage(e) )
+  )
too big
```

R error in C++

R currently does not offer C-level mechanisms to deal with errors. To overcome this problem, **Rcpp** uses the `Rcpp::Evaluator` class to evaluate an expression with an R-level `tryCatch` block. The error, if

[6] The version of `lapply` does not include usage of the ellipsis (`...`).

any, that occurs while evaluating the function is then translated in terms of an C++ exception.

Performance comparison

In this section, we illustrate how C++ features may well come with a price in terms of performance. However, as users of **Rcpp**, we do not need to compromise performance for ease of use.

As part of the redesign of **Rcpp**, data copy is kept to the absolute minimum. The `RObject` class and all its derived classes are just a container for a `SEXP`. We let R perform all memory management and access data though the macros or functions offered by the standard R API. In contrast, some data structures of the classic **Rcpp** interface such as the templated `RcppVector` used containers offered by the standard template library to hold the data, requiring explicit copies of the data from R to C++ and back.

Here we illustrate how to take advantage of `Rcpp` to get the best of both worlds. The classic **Rcpp** translation of the convolve example from R Development Core Team (2009a) appears twice above where the second example showed the use with the new API.

The implementation of the `operator[]` is designed as efficiently as possible, using both inlining and caching, but even this implementation is still less efficient than the reference C implementation described in R Development Core Team (2009a).

In order to achieve maximum efficiency, the reference implementation extracts the underlying array pointer `double*` and works with pointer arithmetics, which is a built-in operation as opposed to calling the `operator[]` on a user-defined class which has to pay the price of object encapsulation.

Modelled after containers of the C++ STL, the `NumericVector` class provides two member functions `begin` and `end` that can use used to retrieve respectively the pointer to the first and past-to-end elements of the underlying array. We can revisit the code to take advantage of this feature :

```
#include <Rcpp.h>

RcppExport SEXP convolve4cpp(SEXP a, SEXP b){
    Rcpp::NumericVector xa(a);
    Rcpp::NumericVector xb(b);
    int n_xa = xa.size() ;
    int n_xb = xb.size() ;
    Rcpp::NumericVector xab(n_xa + n_xb - 1);

    double* pa = xa.begin() ;
    double* pb = xb.begin() ;
    double* pab = xab.begin() ;
    int i,j=0;
    for (i = 0; i < n_xa; i++)
        for (j = 0; j < n_xb; j++)
            pab[i + j] += pa[i] * pb[j];

    return xab ;
}
```

We have benchmarked the various implementations by averaging over 1000 calls of each function with `a` and `b` containing 100 elements each.[7] The timings are summarized in the table below:

Implementation	Time in millisec	Relative to R API
R API (as benchmark)	32	
`RcppVector<double>`	354	11.1
`NumericVector::operator[]`	52	1.6
`NumericVector::begin`	33	1.0

Table 2: Performance for convolution example

The first implementation, using the traditional R API, unsurprisingly appears to be the most efficient. It takes advantage of pointer arithmetics and does not pay the price of object encapsulation. This provides our base case.

The second implementation—from the classic **Rcpp** API—is clearly behind in terms of efficiency. The difference is mainly caused by the many unnecessary copies that the `RcppVector<double>` class performs. First, both objects (`a` and `b`) are copied into C++ structures (`xa` and `xb`). Then, the result is constructed as another `RcppVector<double>` (`xab`) that is filled using the `operator()` which checks at each access that the index is suitable for the object. Finally, `xab` is converted back to an R object.

The third implementation—using the more efficient new **Rcpp** API—is already orders of magnitude faster than the preceding solution. Yet it illustrates the price of object encapsulation and of calling an overloaded `operator[]` as opposed to using pointer arithmetics.

Finally, the last implementation comes very close to the base case and shows the code using the new API can essentially as fast as the R API base case while being easier to write.

Summary

The `Rcpp` package greatly simplifies integration of compiled C++ code with R.

The class hierarchy allows manipulation of R data structures in C++ using member functions and operators directly related to the type of object being used, therefore reducing the level of expertise required to master the various functions and macros offered by the internal R API. The classes assume the entire responsibility of garbage collection of objects, relieving the programmer from book-keeping operations with

[7] The code for this example is contained in the directory `inst/examples/ConvolveBenchmarks` in the **Rcpp** package.

the protection stack and enabling him/her to focus on the underlying problem.

Data interchange between R and C++ code—performed by the `wrap` and `as` template functions—allows the programmer to write logic in terms of C++ data structures and facilitates use of modern libraries such as the standard template library and its containers and algorithms. The `wrap()` and `as()` template functions are extensible by design and can be used either explicitly or implicitly throughout the API. By using only thin wrappers around `SEXP` objects, the footprint of the `Rcpp` API is very lightweight, and does not induces a significant performance price.

The `Rcpp` API offers opportunities to dramatically reduce the complexity of code, which should improve code readability, maintainability and reuse.

Bibliography

W. Armstrong. *RAbstraction: C++ abstraction for R objects*, 2009a. URL http://github.com/armstrtw/rabstraction. Code repository last updated July 22, 2009.

W. Armstrong. *RObjects: C++ wrapper for R objects (a better implementation of RAbstraction*, 2009b. URL http://github.com/armstrtw/RObjects. Code repository last updated November 28, 2009.

D. M. Bates and S. DebRoy. C++ classes for R objects. In K. H. . F. Leisch, editor, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, TU Vienna, Austria, 2001.

D. Eddelbuettel and R. François. *RInside: C++ classes to embed R in C++ applications*, 2010. URL http://CRAN.R-project.org/package=RInside. R package version 0.2.2.

D. Eddelbuettel and K. Nguyen. *RQuantLib: R interface to the QuantLib library*, 2009. URL http://CRAN.R-project.org/package=RQuantLib. R package version 0.3.1.

R. François and D. Eddelbuettel. *RProtoBuf: R Interface to the Protocol Buffers API*, 2010. URL http://CRAN.R-project.org/package=RProtoBuf. R package version 0.1-0.

J. J. Java, D. P. Gaile, and K. E. Manly. R/Cpp: Interface classes to simplify using R objects in C++ extensions. Unpublished manuscript, University at Buffalo, July 2007. URL http://sphhp.buffalo.edu/biostat/research/techreports/UB_Biostatistics_TR0702.pdf.

G. Liang. *rcppbind: A template library for R/C++ developers*, 2008. URL http://r-forge.r-project.org/projects/rcppbind/. R package version 1.0.

S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, third edition, 2005. ISBN 978-0321334879.

P. Plauger, A. Stepanov, M. Lee, and D. R. Musser. *The C++ Standard Template Library*. Prentice Hall PTR, 2000. ISBN 978-0134376332.

R Development Core Team. *Writing R extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2009a. URL http://cran.r-project.org/doc/manuals/R-exts.html.

R Development Core Team. *R internals*. R Foundation for Statistical Computing, Vienna, Austria, 2009b. URL http://cran.r-project.org/doc/manuals/R-ints.html.

A. Runnalls. Aspects of CXXR internals. In *Directions in Statistical Computing*, University of Copenhagen, Denmark, 2009.

D. Samperi. *RcppTemplate: Rcpp R/C++ Object Mapping Library and Package Template*, 2009. URL http://CRAN.R-project.org/src/contrib/Archive/RcppTemplate. (Archived) R package version 6.1.

O. Sklyar, D. Murdoch, M. Smith, and D. Eddelbuettel. *inline: Inline C, C++, Fortran function calls from R*, 2009. URL http://CRAN.R-project.org/package=inline. R package version 0.3.4.

D. Temple Lang. A modest proposal: an approach to making the internal R system extensible. *Computational Statistics*, 24(2):271–281, May 2009a.

D. Temple Lang. Working with meta-data from C/C++ code in R: the RGCCTranslationUnit package. *Computational Statistics*, 24(2):283–293, May 2009b.

S. Urbanek. *Rserve: Binary R server*, 2009. URL http://www.rforge.net/Rserve/. R package version 0.6-1.

Dirk Eddelbuettel
Debian Project
Chicago, IL
USA
edd@debian.org

Romain François
Professionnal R Enthusiast
3 rue Emile Bonnet, 34 090 Montpellier
FRANCE
romain@r-enthusiasts.com