Getting Started with Rcpp

Nick Ulle

Introduction

Compiled C and C++ routines can be called from R using the built-in `.Call()` function. R objects passed to these routines have type `SEXP`. A `SEXP` is a pointer to an encapsulated structure that holds the object's type, value, and other attributes used by the R interpreter.

The R application programming interface (API) provides a limited set of macros and C routines for manipulating `SEXP`s and calling R functions.[1] The level of abstraction in the R API is low. Even simple tasks may require writing lengthy boilerplate code. Using the R API from C++ is especially uncomfortable, because it doesn't take advantage of any of C++'s features.

Rcpp[2] is an R package that makes it easier to interface R and C++ code. Rcpp does this by providing a set of C++ wrapper classes for common R data types, as well as tools for automating the process of compiling and loading C++ routines for R.

Installation

Installation is the same as for any other package, although Windows users must first install Rtools.[3]

```
install.packages("Rcpp")
```

Example: Hello, World!

We'll verify that Rcpp is installed and working correctly with a "Hello, world!" program. Create a blank text file and enter the code:

```
#include <Rcpp.h>

// [[Rcpp::export]]
void hello()
{
  Rprintf("Hello, world!\n");
}
```

Save the file as `hello.cpp`.

Let's look at the unfamiliar parts of this code. The header file `Rcpp.h` contains definitions used by Rcpp. The comment `[[Rcpp::export]]` is called an *Rcpp attribute*, and tells Rcpp to make `hello` available for use from R. Finally, the routine `Rprintf`, which is part of the R API, prints to the R console. The syntax is the same as `printf`.

[1] https://cran.r-project.org/doc/manuals/r-release/R-exts.html#The-R-API
[2] http://www.rcpp.org
[3] https://cran.r-project.org/bin/windows/Rtools

Time to test the code! Start R and enter the commands:

```
library(Rcpp)
sourceCpp("hello.cpp")
hello()
```

You should see "Hello, world!" printed on the R console.

The `sourceCpp` function compiles and dynamically loads the specified C++ file. It's very useful for quickly testing out your code.

The Rcpp Interface

Data Structures

Most of Rcpp's functionality is provided through a set of C++ classes that wrap R data structures. A few of them are:

- `IntegerVector`, `NumericVector`, `LogicalVector`, `CharacterVector`
- `List`, `DataFrame`
- `Named`, `Dimension`
- `IntegerMatrix`, `NumericMatrix`
- `Function`
- `Environment`

Memory management is handled automatically by the class constructors and destructors. These classes also have methods that mimic various R functions. A few of the most useful methods are:

- `isNULL`
- `attributeNames`, `hasAttribute`, `attr`
- `length`, `nrow`, `ncol`

The vector and list classes have constructors that accept the number of elements as a parameter, similar to their counterparts in R. The helper class `Dimension` can be used to create a multidimensional vector:

```
// Create a 2-by-3-by-4 vector.
NumericVector a = NumericVector( Dimension(2, 3, 4) );
```

They also have a static `create` method, for specifying the elements of the new vector. The helper class `Named` represents named vector elements. For instance,

```
IntegerVector q1_days = IntegerVector::create(
  Named("January") = 31,
  Named("February") = 28,
  Named("March") = 31
);
```

creates an integer vector with 3 named elements.

Rcpp Sugar

Rcpp sugar adds "syntactic sugar" for performing vectorized operations on the Rcpp classes. In other words, Rcpp sugar is a collection of C++ routines that mirror R's most useful vectorized functions. A few of these include:

- `+`, `-`, `*`, `/`
- `<`, `>`, `<=`, `>=`, `==`, `!=`

- `any`, `all`
- `is_na`, `is_true`, `is_false`
- `seq`, `seq_along`, `seq_len`
- `ifelse`
- `setdiff`, `union`, `intersect`, `unique`
- math functions `log`, `sin`, `cos`, ...
- distribution functions `dnorm`, `qnorm`, ...

They follow the same syntax as in R. A complete list of Rcpp sugar routines can be found in the package documentation.

Other Details

Rcpp converts R objects to and from C++ objects with the templated routines `as` and `wrap`, respectively. It's rarely necessary to call these routines explicitly, but since Rcpp makes frequent implicit use of them, it's important to know what they do.

The `clone` routine makes a copy of an Rcpp object. Since C++ uses reference semantics, you must explicitly call `clone` when you want to make a copy.

Missing values can be specified with the constants `NA_INTEGER`, `NA_REAL`, `NA_LOGICAL`, and `NA_STRING`. The special values `NaN`, `Inf`, and `-Inf` can be specified with the constants `R_NaN`, `R_PosInf`, and `R_NegInf`. These constants all come from the R API rather than Rcpp.

Programming Strategy

Generally speaking, you should write most of your code in R, to take advantage of its high level of abstraction. Then you can profile your code to identify bottlenecks where R is unacceptably slow, and replace those sections with C++ code for a performance boost. The most straightforward way to do this is to rewrite an entire function. As long as your C++ routine has the same call signature as the R function it replaces, the change should be invisible to the rest of your application.

Example: Row Maximums

Suppose we want to compute the maximum element of each row in a matrix. To achieve this, we loop over each row of the matrix and use the sugar routine `max`:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector row_max(NumericMatrix m) {
  int nrow = m.nrow();
  NumericVector max(nrow);

  for (int i = 0; i < nrow; i++)
    // Get row i with m(i, _).
    max[i] = Rcpp::max( m(i, _) );

  return max;
}
```

Notice that the matrix classes in Rcpp use parentheses `( )` as the subset operator rather than square brackets `[ ]`. This is due to limitations in C++.

Example: Box Packing

Suppose we want to simulate a discrete box-packing Markov chain. At each time step, an item with weight randomly distributed in $\{1, \ldots, w\}$ arrives for packing. Items are placed in the same box so long as the box weight does not exceed w. If an item would make the current box's weight exceed w, a new box is started with that item. We might be interested in the weight of the current box at each time step, as well as which times a new box is started.[4]

A simulation of the box-packing chain can be implemented in R, but suppose we want to run the simulation for a large number of time steps in order to estimate long-run statistics. In that case, the simulation might be unacceptably slow. We can use C++ and Rcpp to write a much faster version.

Implementation

Create a blank text file and enter the code skeleton:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List pack_boxes(int n, NumericVector p) {

  // ...

}
```

The `pack_boxes` routine will contain our simulation. It needs to sample item weights, add each item weight to the previous time step's box weight, and then check whether the box is too heavy, starting a new box when necessary. The routine has parameters `n`, the number of steps to simulate, and `p`, the probabilities of the item weights. We don't need to make w a parameter, since w can be inferred from the length of `p`. The routine has return type `List`. Rcpp implicitly converts between `SEXP` and these input/output types.

If we were implementing the simulation in R, we could sample the item weights with the `sample` function. The R API doesn't have a corresponding C routine. Fortunately, Rcpp's `Function` class makes calling R functions from C++ simple. The constructor takes the name of the desired function as parameter. After creating a `Function` object for `sample`, we can call it with the same parameters as the original R function. A word of caution: calling R functions from C++ code is at least as slow as calling them from R itself, so use them sparingly.

For the rest of the simulation, we need a vector `weight` of length `n` to hold the weight of the box at each time step, and another vector, `first`, to hold the first item times. We also need a variable `n_boxes` to keep track of how many boxes have been packed.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List pack_boxes(int n, NumericVector p) {
  Function sample = Environment("package:base")["sample"];
```

[4] Thanks to Norm Matloff for this example.

```
  // Sample item weights.
  int w = p.size();
  IntegerVector item = sample(w, n, true, p);

  // Initialize loop variables.
  IntegerVector weight(n);
  weight[0] = item[0];

  IntegerVector first(n);
  first[0] = 1;
  int n_boxes = 1;

  // ...
```

We don't know how long `first` needs to be, but we can ensure it's long enough by making it length `n`, as above. Alternatively, if we were concerned about memory usage, we could've used a data structure from C++'s standard template library and converted to a correctly-sized `IntegerVector` at the end of the simulation with Rcpp's `wrap` routine.

The core of our simulation is a `for` loop. Unlike R, where `for` loops be avoided in favor of vectorized code, there's no penalty for using `for` loops in C++.

```
  for (int i = 1; i < n; i++) {
    int new_weight = weight[i - 1] + item[i];

    if (new_weight <= w) {
      // Continue with current box.
      weight[i] = new_weight;
    } else {
      // Start a new box.
      weight[i] = item[i];
      first[n_boxes++] = i + 1;
    }
  }

  // ...
```

The body of the loop doesn't use any features of Rcpp we haven't seen before. Note that 1 has to be added to the index stored in `first`, because elements are indexed starting from 0 in C++, but from 1 in R.

Finally, `item`, `weight`, and `first` need to be placed in a list to be returned to R. We can create named list elements with the syntax `_[ ... ]`, which is a shortcut for the `Named` helper class' constructor. Also, recall that `first` may be slightly too long, so the extraneous entries need to be removed. With Rcpp sugar, we can subset using a vector of indexes, just like in R.

```
  return List::create(
    _["item"] = item,
    _["weight"] = weight,
    _["first"] = first[seq(0, n_boxes - 1)]
  );
}
```

Now it's time to test the code. Save the file as `box.cpp`. Open the R interpreter and enter the commands

```
library(Rcpp)
sourceCpp("box.cpp")
```

```
set.seed(256)
pack_boxes(10, c(0, 0.7, 0.2, 0.1))
```

You should see the output:

```
$item
 [1] 2 2 2 2 2 3 2 2 2 2

$weight
 [1] 2 4 2 4 2 3 2 4 2 4

$first
[1] 1 3 5 6 7 9
```

This suggests the code works correctly, but also shows that Rcpp takes care of managing the state of the pseudo-random number generator (RNG). We didn't worry about this much while writing the code, but incorrectly handling RNG state from C or C++ code can invalidate a simulation.

Experiment with the arguments to `pack_boxes`. You might notice that it's easy to crash the routine by providing bad arguments. The routine could be made more robust by adding checks for pathological input, or providing error handling code. Rcpp supports forwarding errors to the R interpreter; see the vignettes for details.

Packaging and roxygen2 Integration

The box-packing simulation works, but we might want to offer it to other users as an R package rather than a standalone script. Rcpp includes the `Rcpp.package.skeleton` function, which creates the minimal directory structure and files needed for a package, similar to R's built-in `package.skeleton` function. It also supports adding existing files to the new package.

For instance, to create a package called BoxPack for the box-packing simulation, open the R interpreter and enter the commands

```
library(Rcpp)
Rcpp.package.skeleton("BoxPack", example_code = FALSE, cpp_files = "box.cpp")
```

The `example_code` parameter controls whether an example C++ file should be generated, while the `cpp_files` parameter controls which C++ files are included in the package. A few more steps are needed before the package is ready for distribution.

Rcpp automatically generates wrappers for C++ routines tagged with the `[[Rcpp::export]]` attribute. The wrappers are stored in `R/RcppExports.R` and `src/RcppExports.cpp`. Avoid editing these files manually. If you update your C++ code, call `compileAttributes()` from R to regenerate the wrappers.

First, the package needs documentation. You can write the documentation by hand, or you can use roxygen2. The roxygen2 package lets you document functions with comments in your code, and automatically converts those comments to R help files. Rcpp has full support for roxygen2. For instance, to document the `pack_boxes` routine, add the following comments above its definition in `box.cpp`.

```
//' Simulate box-packing.
//'
//' @export
// [[Rcpp::export]]
List pack_boxes(int n, NumericVector p) {
```

The roxygen2 `@export` tag says we want to *export* the `pack_boxes` routine so it's available to users of our package. This is different from the Rcpp tag `[[Rcpp::export]]`, which merely says an an R wrapper function should be generated for the `pack_boxes` routine.

After adding the roxygen2 comments, regenerate the wrapper functions and documentation by opening an R interpreter and entering the commands

```
library(Rcpp)
compileAttributes()

library(roxygen2)
roxygenize()
```

You'll still need to fill out the package's top-level help file by hand, although you can automate this as well by learning more about roxygen2. The full details are beyond the scope of this note, but it's an extremely useful package to learn.

Second, fill out the generated `DESCRIPTION` file with the package details.

Finally, build your package and run it through `R CMD check`. Depending on how you designed your package, you might see the message

```
* checking dependencies in R code ... NOTE
Namespace in Imports field not imported from: 'Rcpp'
  All declared Imports should be used.
```

This `NOTE` is harmless and can be safely ignored. However, make sure to correct any `ERROR` and `WARNING` messages before sharing your package.

Resources

The vignettes included with Rcpp are a good starting point for learning more. They can be accessed from the R console with the `vignette` function. For instance:

```
library(Rcpp)
vignette("Rcpp-attributes")
```

Call `vignette()` to see a list of all vignettes.

The Rcpp Gallery[5] has many short examples of Rcpp in a variety of contexts.

For a definitive reference on Rcpp, see Dirk Eddelbuettel's book, *Seamless R and C++ Integration with Rcpp*.[6] Dirk is one of the package's primary developers.

[5] http://gallery.rcpp.org/
[6] http://www.rcpp.org/book/