Speeding up R with Rcpp

Luke KLEIN

9/24/2019

References

This presentation is adapted from portions of:

- Hadley Wickham's book
  - Wickham, Hadley. Advanced r. Chapman and Hall/CRC, 2014.
- Dirk Eddelbuttel's book
  - Eddelbuettel, Dirk, et al. "Rcpp: Seamless R and C++ integration." Journal of Statistical Software 40.8 (2011): 1-18.
- Dirk Eddelbuttel's presentation
  - Eddelbuettel, Dirk. "Introduction Rcpp Workshop", 12/06/2012, http://dirk.eddelbuettel.com/papers/rcpp_workshop_introduction_user2012.pdf

R is powerful

R is both a powerful **interactive** environment for data analysis, visualization, and modeling and an **expressive programming language** designed and built to support these tasks.

- The interactive nature of working with data – through data displays, summaries, model estimation, simulation, and numerous other tasks – is a key strength of the R environment.

The R programming language permits many applications – from interactive explorations to small scripts and all the way to complete implementations of new functionality.

R can be slow

Loops are slower in R than in C++ because R is an interpreted language (not compiled).

In most cases, R does not modify variables in place, but rather modifies a **copy**.

- The repeated allocation of new memory for these tasks is a large source of performance reduction.

R can be slow: An illustration

```r
x <- 1:1e6
y <- c(x,x,x)
z <- list(x, x, x)

object_size(x)
object_size(y)
object_size(z)
object_size(x,y)
object_size(x,z)
```

```
## 4 MB
## 12 MB
## 4 MB
## 16 MB
## 4 MB
```

Introduction: A first example

Let's illustrate the speed gains possible by using Rcpp

First we'll repeatedly evaluate $\frac{1}{1+x}$ using R, and use the rbenchmark package to compare

```
f1 <- function(n, x=1) for (i in 1:n) x=1/(1+x)
f2 <- function(n, x=1) for (i in 1:n) x=(1/(1+x))
f3 <- function(n, x=1) for (i in 1:n) x=(1+x)^(-1)
f4 <- function(n, x=1) for (i in 1:n) x={1/{1+x}}
f5 <- function(n, x=1) for (i in 1:n) x=1/{1+x}
```

Introduction: A first example

Now, let's run the benchmark

```
N <- 1e5
benchmark(f1(N,1), f2(N,1), f3(N,1), f4(N,1), f5(N,1),
          columns=c("test", "replications", "elapsed",
                    "relative"),
          order="relative", replications=10)
```

```
##       test replications elapsed relative
## 1 f1(N, 1)           10   0.061    1.000
## 5 f5(N, 1)           10   0.063    1.033
## 4 f4(N, 1)           10   0.065    1.066
## 2 f2(N, 1)           10   0.067    1.098
## 3 f3(N, 1)           10   0.110    1.803
```

Introduction: A first example

Let's bring Rcpp into the mix and see what kind of performance enhancements we can get.

cppFunction() allows you to write C++ functions in R, simply.

```
library(Rcpp)

cppFunction('int fCpp(int n, double x) {
  for (int i=0; i<n; i++) x=1/(1+x);
  return 0;
}')
```

When you run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function.

Introduction: A first example

Now let's run the benchmark again!!!

```
N <- 1e5
benchmark(f1(N,1.0), f2(N,1), f3(N,1), f4(N,1), f5(N,1),
          fCpp(N,1.0),
          columns=c("test", "replications", "elapsed",
                    "relative"),
          order="relative", replications=10)
```

```
##          test replications elapsed relative
## 6 fCpp(N, 1)           10   0.003    1.000
## 1   f1(N, 1)           10   0.055   18.333
## 4   f4(N, 1)           10   0.057   19.000
## 2   f2(N, 1)           10   0.059   19.667
## 5   f5(N, 1)           10   0.068   22.667
## 3   f3(N, 1)           10   0.112   37.333
```

A second example

The standard definition of the Fibonacci sequence is
$F_n = F_{(n-1)} + F_{(n-2)}$ with initial values $F_0 = 0$ and $F_1 = 1$.

This leads this intuitive (but slow) R implementation:

```
## basic R function
fibR <- function(n) {
  if (n == 0) return(0)
  if (n == 1) return(1)
  return (fibR(n - 1) + fibR(n - 2))
}
```

A second example

We can write an easy (and very fast) C++ version:

```
cppFunction('int fibCpp(int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return (fibCpp(x - 1)) + fibCpp(x - 2);
}')
```

A second example

Now, let's compare the two:

```
N <- 35L
benchmark(fibCpp(N), fibR(N),
 columns=c("test", "replications", "elapsed",
 "relative"),
 order="relative", replications=1)
```

```
##        test replications elapsed relative
## 1 fibCpp(N)            1   0.076    1.000
## 2   fibR(N)            1  15.284  201.105
```

So a two-hundred fold increase for no real effort or setup cost.

sourceCpp

So far, we've used inline C++ with `cppFunction()`. This makes presentation simpler, but for real problems, it's usually easier to use stand-alone C++ files and then source them into R using `sourceCpp()`.

This lets you take advantage of text editor support for C++ files (e.g., syntax highlighting) as well as making it easier to identify the line numbers in compilation errors.

sourceCpp

Your stand-alone C++ file should have extension .cpp, and needs to start with:

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

And for each function that you want available within R, you need to prefix it with:

```
// [[Rcpp::export]]
```

Note that the **space** is mandatory.

sourceCpp

You can embed R code in special C++ comment blocks. This is really convenient if you want to run some test code:

```
/*** R
# This is R code
*/
```

The R code is run with `source(echo = TRUE)` so you don't need to explicitly print output.

To compile the C++ code, use
`sourceCpp("path/to/file.cpp")`

- This will create the matching R functions and add them to your current session.

Note: that these functions can not be saved in a `.Rdata` file and reloaded in a later session; they must be recreated each time you restart R.

sourceCpp

For example, running `sourceCpp()` on the following file implements mean in C++ and then compares it to the built-in `mean()`:

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

// [[Rcpp::export]]
double meanC(NumericVector x) {
  int n = x.size();
  double total = 0;

  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total / n;
}

/*** R
x <- runif(1e5)
microbenchmark(
  mean(x),
  meanC(x)
)
*/
```

sourceCpp

The output of the R code in the lower comment will appear like this

```
> library(microbenchmark)

> x <- runif(1e5)

> microbenchmark(
+   mean(x),
+   meanC(x)
+ )
Unit: microseconds
     expr     min      lq     mean   median       uq      max neval
  mean(x) 192.890 194.034 229.6489 206.0235 227.4670  775.640   100
 meanC(x)  96.163  97.063 142.6581 100.6380 111.1115 1946.385   100
```

Differences between R and C++

- In C++, Programs need to be **compiled first**. This may require access to header files defining interfaces to other projects.
- After compiling code into an object, the object is linked into an executable, possibly together with other libraries.
- Pointers and memory management are handled very differently, but many issues common with C can be avoided via STL, i.e. the C++ standard template library (which is something Rcpp promotes as well).

Differences between R and C++

- R is dynamically typed:
  - `x <- 3.14; x <- "foo"` is valid.
- In C++, each variable must be **declared** before first use. Common types are `int` and `long` (possibly with `unsigned`), `float` and `double`, `bool`, as well as `char`.
- No standard string type, though `std::string` comes close.

Differences between R and C++

Note that all of these C++ variable types are **scalars** which is fundamentally different from `R` where everything is a **vector** (possibly of *length one*).

Differences between R and C++

Classes

C++ has many basic vector classes

- `IntegerVector`, `NumericVector`, `LogicalVector`, `CharacterVector`

and their scalar and matrix equivalents

- `int`, `double`, `bool`, `String`
- `IntegerMatrix`, `NumericMatrix`, `LogicalMatrix`, `CharacterMatrix`

Differences between R and C++

Attributes

All R objects have arbitrary additional attributes, used to store **metadata** about the object.

Attributes can be thought of as a named list (with unique names). Attributes can be accessed individually with attr() or all at once (as a list) with attributes().

- See Hadley's **Advanced R** for more details.

In C++, attributes can be queried and modified with `.attr()`. Rcpp also provides `.names()` as an alias for the name attribute. The following code snippet illustrates these methods.

Differences between R and C++

Attributes

Note the use of `::create()`, a *class* method. This allows you to create an R vector from C++ scalar values:

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

// [[Rcpp::export]]
NumericVector attribs() {
  NumericVector out = NumericVector::create(1, 2, 3);

  out.names() = CharacterVector::create("a", "b", "c");
  out.attr("my-attr") = "my-value";
  out.attr("class") = "my-class";

  return out;
}
```

Differences between R and C++

Missing values

If you're working with missing values in C++, you need to know **two main things**:

- how R's missing values behave in C++'s scalars (e.g., double).
- how to get and set missing values in vectors (e.g., NumericVector).

Differences between R and C++

Missing values: Scalars

The following code explores what happens when you take one of R's missing values, coerce it into a scalar, and then coerce back to an R vector.

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

// [[Rcpp::export]]
List scalar_missings() {
  int int_s = NA_INTEGER;
  String chr_s = NA_STRING;
  bool lgl_s = NA_LOGICAL;
  double num_s = NA_REAL;

  return List::create(int_s, chr_s, lgl_s, num_s);
}
```

Differences between R and C++

Let's call the `scalar_missings` function.

```r
str(scalar_missings())
```

```
## List of 4
##  $ : int NA
##  $ : chr NA
##  $ : logi TRUE
##  $ : num NA
```

With the exception of bool, things look pretty good here: all of the missing values have been preserved. However, as we'll see in the following sections, things are not quite as straightforward as they seem.

Differences between R and C++

Missing values: Boolean

While C++'s bool has two possible values - `true` or `false`

A logical vector in `R` has three possible values

- `TRUE`, `FALSE`, and `NA`

So, if you coerce a length 1 logical vector from `C++` to `R`, make sure it doesn't contain any missing values otherwise they will be converted to TRUE.

Differences between R and C++

Missing values: Integers

With integers in C++, missing values are **stored as the smallest possible integer**. If you don't do anything to them when passing them to R, **their values will be preserved**.

So, since `R` doesn't know that the smallest integer has this special meaning in C++, if you do anything to it you're likely to get an incorrect value: for example,

```
evalCpp('NA_INTEGER + 1')
```

```
## [1] -2147483647
```

Thus, if you want to work with missing integer values, either use a length one `IntegerVector` in C++ or be *very* careful with your code.

Differences between R and C++

Missing values: Doubles

With doubles, you may be able to get away with ignoring missing values and working with `NAN`s (not a number). This is because R's NA is a special type of IEEE 754 floating point number NaN. So any logical expression that involves a NaN (or in C++, NAN) always evaluates as FALSE:

```r
evalCpp("NAN == 1")
evalCpp("NAN < 1")
evalCpp("NAN > 1")
evalCpp("NAN == NAN")
```

```
## [1] FALSE
## [1] FALSE
## [1] FALSE
## [1] FALSE
```

Differences between R and C++

Missing values: Doubles

However, in numeric contexts NaNs will correctly propagate NAs:

```
evalCpp("NAN + 1")
evalCpp("NAN - 1")
evalCpp("NAN / 1")
evalCpp("NAN * 1")
```

```
## [1] NaN
## [1] NaN
## [1] NaN
## [1] NaN
```

Differences between R and C++

Missing values: Vectors

With vectors, you need to use a missing value specific to the type of vector, `NA_REAL`, `NA_INTEGER`, `NA_LOGICAL`, `NA_STRING`:

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

// [[Rcpp::export]]
List missing_sampler() {
  return List::create(
    NumericVector::create(NA_REAL),
    IntegerVector::create(NA_INTEGER),
    LogicalVector::create(NA_LOGICAL),
    CharacterVector::create(NA_STRING));
}
```

Differences between R and C++

Missing values: Vectors

```r
str(missing_sampler())
```

```
## List of 4
##  $ : num NA
##  $ : int NA
##  $ : logi NA
##  $ : chr NA
```

Differences between R and C++

Missing values: Vectors

To check if a value in a vector is missing, use the class method `::is_na()`:

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

// [[Rcpp::export]]
LogicalVector is_naC(NumericVector x) {
  int n = x.size();
  LogicalVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = NumericVector::is_na(x[i]);
  }
  return out;
}
```

Differences between R and C++

Missing values: Vectors

Running the code yields the following:

```
is_naC(c(NA, 5.4, 3.2, NA))
```

```
## [1]  TRUE FALSE FALSE  TRUE
```

Differences between R and C++

Missing values: Vectors

Another alternative is the sugar function `is_na()`, which takes a vector and returns a logical vector.

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

// [[Rcpp::export]]
LogicalVector is_naC2(NumericVector x) {
  return is_na(x);
}
```

```
is_naC2(c(NA, 5.4, 3.2, NA))
```

```
## [1]  TRUE FALSE FALSE  TRUE
```

Rcpp sugar

In computer science, **syntactic sugar** is syntax within a programming language that is designed to make things easier to read or to express.

It makes the language **“sweeter”** for human use:

- Things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

Rcpp provides a lot of syntactic “sugar” *to ensure that C++ functions work very similarly to their R equivalents*. In fact, Rcpp sugar makes it possible to write efficient C++ code that looks almost identical to its R equivalent!

If there’s a sugar version of the function you’re interested in, you should use it: it’ll be both expressive and well tested. Sugar functions aren’t always faster than a handwritten equivalent, but they will get faster in the future as more time is spent on optimising Rcpp.

Rcpp sugar

Sugar functions can be roughly broken down into

- arithmetic and logical operators
- logical summary functions
- vector views

First, there's a grab bag of sugar functions that mimic frequently used R functions:

- **Math functions**: `abs()`, `acos()`, `asin()`, `atan()`, `beta()`, `ceil()`, `ceiling()`, `choose()`, `cos()`, `cosh()`, `digamma()`, `exp()`, `expm1()`, `factorial()`, `floor()`, `gamma()`, `lbeta()`, `lchoose()`, `lfactorial()`, `lgamma()`, `log()`, `log10()`, `log1p()`, `pentagamma()`, `psigamma()`, `round()`, `signif()`, `sin()`, `sinh()`, `sqrt()`, `tan()`, `tanh()`, `tetragamma()`, `trigamma()`, `trunc()`.

Rcpp sugar

Also:

- **Scalar summaries**: `mean()`, `min()`, `max()`, `sum()`, `sd()`, and `var()`.
- **Vector summaries**: `cumsum()`, `diff()`, `pmin()`, and `pmax()`.
- **Finding values**: `match()`, `self_match()`, `which_max()`, `which_min()`.
- **Dealing with duplicates**: `duplicated()`, `unique()`.
- `d/q/p/r` for **all standard distributions**.

Rcpp sugar: Arithmetic and logical operators

All the basic arithmetic and logical operators are **vectorised**: `-` `+`, `*`, `-`, `/`, `pow`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `!`.

For example, we could use sugar to considerably simplify the implementation of `pdistC()`.

Rcpp sugar: Arithmetic and logical operators

```
cppFunction('NumericVector pdistC(double x, NumericVector y
  int n = ys.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  return out;
}')
```

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

// [[Rcpp::export]]
NumericVector pdistC2(double x, NumericVector ys) {
  return sqrt(pow((x - ys), 2));
}
```

Rcpp sugar: Logical summary functions

The sugar function `any()` and `all()` are fully *lazy* so that `any(x == 0)`, for example, **might only need to evaluate one element of a vector**, and return a special type that can be converted into a bool in C++ using `.is_true()`, `.is_false()`, or `.is_na()`.

We could also use this sugar to write an efficient function to determine whether or not a numeric vector contains any missing values.

To do this in R, we could use `any(is.na(x))`:

```r
any_naR <- function(x) any(is.na(x))
```

However, in R this will do the same amount of work regardless of the location of the missing value because the base R implementation is NOT lazy.

Rcpp sugar: Logical summary functions

Here's the C++ implementation using `Rcpp` sugar:

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

// [[Rcpp::export]]
bool any_naC(NumericVector x) {
  return is_true(any(is_na(x)));
}
```

Rcpp sugar: Logical summary functions

Now let's compare them using three use-cases.

```
x0 <- runif(1e5)
x1 <- c(x0, NA)
x2 <- c(NA, x0)

benchmark(any_naR(x0), any_naC(x0), any_naR(x1), any_naC(x1
          columns=c("test", "replications", "elapsed",
                    "relative", "user.self"),
          order=NULL, replications=100)
```

```
##            test replications elapsed relative user.self
## 1 any_naR(x0)          100   0.054       54     0.040
## 2 any_naC(x0)          100   0.030       30     0.030
## 3 any_naR(x1)          100   0.049       49     0.037
## 4 any_naC(x1)          100   0.030       30     0.031
## 5 any_naR(x2)          100   0.027       27     0.017
## 6 any_naC(x2)          100   0.001        1     0.001
```

Rcpp sugar: Vector views

A number of helpful functions provide a “view” of a vector: `head()`, `tail()`, `rep_each()`, `rep_len()`, `rev()`, `seq_along()`, and `seq_len()`.

- In R **these would all produce copies of the vector**, but in Rcpp they simply point to the existing vector and override the subsetting operator (`[`) to implement special behaviour.
- This makes them **very efficient**: for instance, `rep_len(x, 1e6)` does not have to make a million copies of `x`!

Example: Gibbs sampler

The following case study illustrates the conversion of a Gibbs sampler in R to C++.

The R and C++ code shown below is very similar (it only took a few minutes to convert the R version to the C++ version), but runs about 20 times faster on my computer.

Dirk Eddelbuettel's blog post also shows another way to make it even faster: - using the faster random number generator functions in **GSL** (easily accessible from R through the `RcppGSL` package) can make it another 2 - 3x faster.

Example: Gibbs sampler

The R code is as follows:

```r
gibbs_r <- function(N, thin) {
  mat <- matrix(nrow = N, ncol = 2)
  x <- y <- 0

  for (i in 1:N) {
    for (j in 1:thin) {
      x <- rgamma(1, 3, y * y + 4)
      y <- rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))
    }
    mat[i, ] <- c(x, y)
  }
  mat
}
```

Example: Gibbs sampler

This is straightforward to convert to C++. We:

- add type declarations to all variables
- use `(` instead of `[` to index into the matrix
- subscript the results of `rgamma` and `rnorm` to convert from a vector into a scalar

Example: Gibbs sampler

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

// [[Rcpp::export]]
NumericMatrix gibbs_cpp(int N, int thin) {
  NumericMatrix mat(N, 2);
  double x = 0, y = 0;

  for(int i = 0; i < N; i++) {
    for(int j = 0; j < thin; j++) {
      x = rgamma(1, 3, 1 / (y * y + 4))[0];
      y = rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))[0];
    }
    mat(i, 0) = x;
    mat(i, 1) = y;
  }

  return(mat);
```

Example: Gibbs sampler

Benchmarking the two implementations yields:

```r
benchmark(gibbs_r(100, 10), gibbs_cpp(100, 10),
          columns=c("test", "replications", "elapsed",
                    "relative"),
          order=NULL, replications=100)
```

```
##                  test replications elapsed relative
## 1   gibbs_r(100, 10)          100   0.830   21.842
## 2 gibbs_cpp(100, 10)          100   0.038    1.000
```

Using Rcpp in a package

The same C++ code that is used with `sourceCpp()` can also be bundled into a package. There are several benefits of moving code from a stand-alone C++ source file to a package:

- Your code can be made available to users without C++ development tools.
- Multiple source files and their dependencies are handled automatically by the R package build system.
- Packages provide additional infrastructure for testing, documentation, and consistency.

Using Rcpp in a package

To add `Rcpp` to an existing package, you put your C++ files in the `src/` directory and modify/create the following configuration files:

- In `DESCRIPTION`, add:

```
LinkingTo: Rcpp
Imports: Rcpp
```

We need to *import* Rcpp so that internal Rcpp code is properly loaded. This is a bug in R and hopefully will be fixed in the future.

- Make sure your `NAMESPACE` includes:

```
useDynLib(mypackage)
importFrom(Rcpp, sourceCpp)
```

Using Rcpp in a package

To generate a new Rcpp package that includes a simple `"hello world"` function you can use `Rcpp.package.skeleton()`:

```r
Rcpp.package.skeleton("NewPackage", attributes = TRUE)
```

To generate a package based on C++ files that you've been using with `sourceCpp()`, use the `cpp_files` parameter:

```r
Rcpp.package.skeleton("NewPackage", example_code = FALSE,
                      cpp_files = c("convolve.cpp"))
```

Using Rcpp in a package

Finally, before building the package, you'll need to run `Rcpp::compileAttributes()`.

- This function scans the C++ files for `Rcpp::export` attributes and generates the code required to make the functions available in R.
- Re-run compileAttributes() **whenever** functions are added, removed, or have their signatures changed.
  - This is done automatically by the `devtools` package and by Rstudio.

For more details see the `Rcpp` package vignette, `vignette("Rcpp-package")`.

End