Rewriting R code in C++

R Programming - HAX815X

Jean-Michel Marin

February 2026

Faculty of Sciences, University of Montpellier

Introduction

- Sometimes R code just isn't fast enough
- We will learn how to improve performance by rewriting key functions in C++
- This magic comes by way of the Rcpp package
- Rcpp makes it very simple to connect C++ to R

Introduction

- Rcpp provides a clean, approachable API that lets you write high-performance code, insulated from R's complex C API

- Typical bottlenecks that C++ can address include:
  - Loops that can't be easily vectorised because subsequent iterations depend on previous ones
  - Recursive functions, or problems which involve calling functions millions of times
  - Problems that require advanced data structures and algorithms that R doesn't provide.

Introduction

- The aim of this course is to discuss only those aspects of C++ and Rcpp that are absolutely necessary to help you eliminate bottlenecks in your code
- We won't spend much time on advanced features like object-oriented programming or templates because the focus is on writing small, self-contained functions, not big programs
- A working knowledge of C++ is helpful, but not essential

Prerequisites

We will use Rcpp to call C++ from R

```r
library(Rcpp)
```

- You'll also need a working C++ compiler. To get it:
  - On Windows, install Rtools
  - On Mac, install Xcode from the app store
  - On Linux, sudo apt-get install r-base-dev or similar

Getting started with C++

`cppFunction()` allows you to write C++ functions in R

```
cppFunction('int add(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}')
add(1, 2, 3)
```

```
## [1] 6
```

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

No inputs, scalar output

Let's start with a very simple function

It has no arguments and always returns the integer 1

```r
one <- function() 1
```

The equivalent C++ function is

```cpp
int one() {
  return 1;
}
```

No inputs, scalar output

We can compile and use this from R with cppFunction()

```r
cppFunction('int one() {
  return 1;
}')
```

No inputs, scalar output

This small function illustrates a number of important differences between R and C++

- The syntax to create a function looks like the syntax to call a function
- You must declare the type of output the function returns. This function returns an int (a scalar integer). The classes for the most common types of R vectors are: NumericVector, IntegerVector, CharacterVector, and LogicalVector
- Scalars and vectors are different. The scalar equivalents of numeric, integer, character, and logical vectors are: double, int, String, and bool
- You must use an explicit return statement to return a value from a function
- Every statement is terminated by a `;`

Scalar input, scalar output

The next example function implements a scalar version of the sign() function which returns 1 if the input is positive, and -1 if it's negative

```
signR <- function(x) {
  if (x > 0) { 1
  } else if (x == 0) { 0
  } else { -1
  }
}
signR(-5)
```

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

Scalar input, scalar output

```
cppFunction('int signC(int x) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return -1;
  }
}')
signC(-5)
```

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

Scalar input, scalar output

In the C++ version

- We declare the type of each input in the same way we declare the type of the output
- The if syntax is identical — while there are some big differences between R and C++, there are also lots of similarities! C++ also has a while statement that works the same way as R's

Vector input, scalar output

One big difference between R and C++ is that the cost of loops is much lower in C++ ; for example, we could implement the sum function in R using a loop

```
sumR <- function(x) {
  total <- 0
  for (i in seq_along(x)) {
    total <- total + x[i]
  }
  total
}
sumR(c(5,6,7))
```

```
## [1] 18
```

Vector input, scalar output

In C++, loops have very little overhead, so it's fine to use them

```
cppFunction('double sumC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total;
}')
sumC(c(5,6,7))
```

```
## [1] 18
```

Vector input, scalar output

The C++ version is similar, but

- To find the length of the vector, we use the .size() method, which returns an integer C++ methods are called with . (i.e., a full stop)
- The for statement has a different syntax: `for(init; check; increment)` this loop is initialised by creating a new variable called i with value 0
  - before each iteration we check that `i < n`, and terminate the loop if it's not
  - after each iteration, we increment the value of `i` by one, using the special prefix operator `++` which increases the value of `i` by `1`

Vector input, scalar output

In C++, vector indices start at 0, which means that the last element is at position n-1

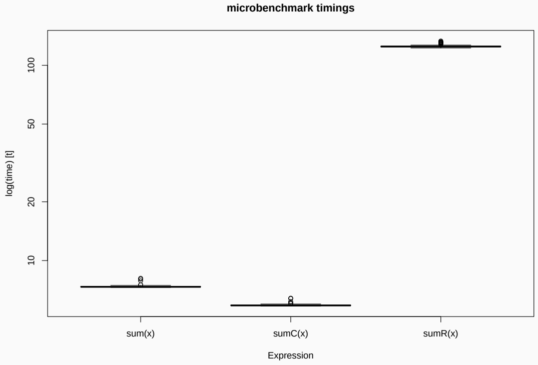
Use `=` for assignment, not `<-`

C++ provides operators that modify in-place: `total += x[i]` is equivalent to `total = total + x[i]`

Vector input, scalar output

This is a good example of where C++ is much more efficient than R

```r
library(microbenchmark)
x <- runif(1e7)
boxplot(microbenchmark(sum(x),sumC(x),sumR(x)))
```



Vector input, vector output

Next we'll create a function that computes the Euclidean distance between a value and a vector of values

```
pdistR <- function(x, ys) {
  sqrt((x - ys) ^ 2)
}
```

In R, it's not obvious that we want x to be a scalar from the function definition, and we'd need to make that clear in the documentation

Vector input, vector output

That's not a problem in the C++ version because we have to be explicit about types

```
cppFunction('NumericVector pdistC(double x, NumericVector ys) {
  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;
}')
```

Vector input, vector output

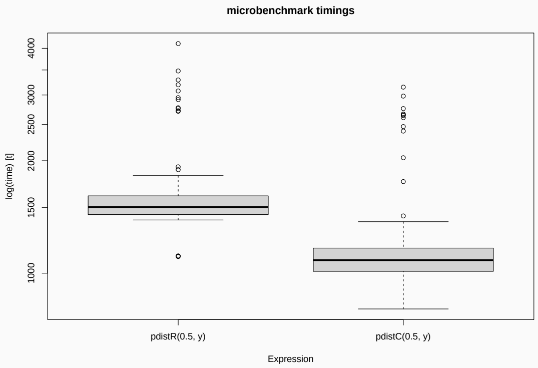
This function introduces only a few new concepts

- We create a new numeric vector of length n with a constructor: `NumericVector out(n)`
- C++ uses `pow()`, not `^`, for exponentiation

Vector input, vector output

Note that because the R version is fully vectorised, it's already going to be fast

```
y <- runif(1e6)
boxplot(microbenchmark(pdistR(0.5, y),pdistC(0.5, y)))
```



Vector input, vector output

- The reason why the C++ function is faster is subtle, and relates to memory management
- The R version needs to create an intermediate vector the same length as y (x - ys), and allocating memory is an expensive operation
- The C++ function avoids this overhead because it uses an intermediate scalar

Using sourceCpp

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

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

Using sourceCpp

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

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

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
*/
```

Using sourceCpp

- 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

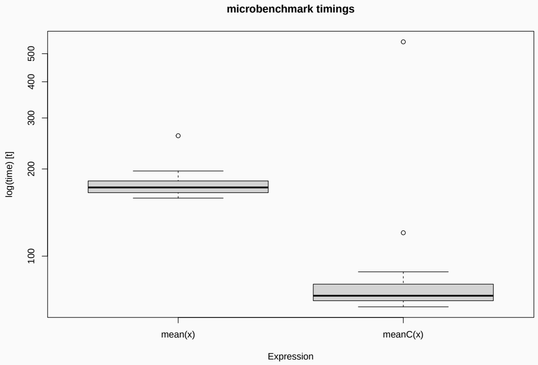
Using sourceCpp

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

```cpp
#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;
}
```

Using sourceCpp

```r
x <- runif(1e5)
boxplot(microbenchmark(mean(x),meanC(x)))
```



R vectorisation versus C++ vectorisation

```r
vacc1a <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * if (female) 1.25 else 0.75
  p <- max(0, p)
  p <- min(1, p)
  p
}
```

R vectorisation versus C++ vectorisation

We want to be able to apply this function to many inputs, so we might write a vector-input version using a for loop.

```
vacc1 <- function(age, female, ily) {
  n <- length(age)
  out <- numeric(n)
  for (i in seq_len(n)) {
    out[i] <- vacc1a(age[i], female[i], ily[i])
  }
  out
}
```

R vectorisation versus C++ vectorisation

If you're familiar with R, you'll have a gut feeling that this will be slow, and indeed it is

There are two ways we could attack this problem

- If you have a good R vocabulary, you might immediately see how to vectorise the function (using ifelse(), pmin(), and pmax()).
- Alternatively, we could rewrite vacc1a() and vacc1() in C++, using our knowledge that loops and function calls have much lower overhead in C++

R vectorisation versus C++ vectorisation

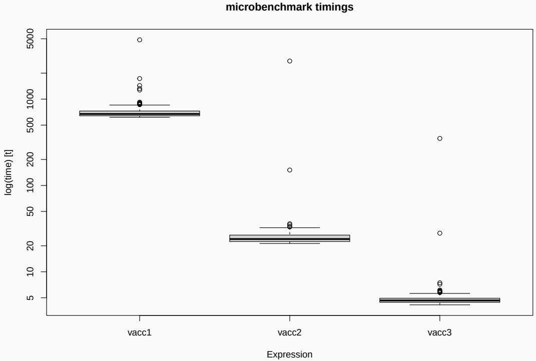
```r
vacc2 <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * ifelse(female, 1.25, 0.75)
  p <- pmax(0, p)
  p <- pmin(1, p)
  p
}
```

R vectorisation versus C++ vectorisation

```
#include <Rcpp.h>
using namespace Rcpp;
double vacc3a(double age, bool female, bool ily){
  double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
  p = p * (female ? 1.25 : 0.75);
  p = std::max(p, 0.0);
  p = std::min(p, 1.0);
  return p; }
// [[Rcpp::export]]
NumericVector vacc3(NumericVector age, LogicalVector female,
                    LogicalVector ily) {
  int n = age.size();
  NumericVector out(n);
  for(int i = 0; i < n; ++i) {
    out[i] = vacc3a(age[i], female[i], ily[i]); }
  return out; }
```

R vectorisation versus C++ vectorisation

```r
n <- 1000
age <- rnorm(n, mean = 50, sd = 10)
female <- sample(c(T, F), n, rep = TRUE)
ily <- sample(c(T, F), n, prob = c(0.8, 0.2), rep = TRUE)
boxplot(microbenchmark(vacc1 = vacc1(age, female, ily),
  vacc2 = vacc2(age, female, ily),
  vacc3 = vacc3(age, female, ily)))
```



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 create or modify the following configuration files:

```
In DESCRIPTION add

LinkingTo: Rcpp
Imports: Rcpp
```

Make sure your NAMESPACE includes:

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

Using Rcpp in a package

We need to import something (anything) from Rcpp so that internal Rcpp code is properly loaded

The easiest way to set this up automatically is to call `usethis::use_rcpp()`

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