Rcpp: A Primer

Quang Nguyen

November 5, 2025

qntkhvn.github.io/Rcpp-primer/slides.pdf

Full disclosure

- I'm an **R** user, with some working knowledge of **C++** (thanks to **Rcpp**)
- This is not a **C++** tutorial
- This is a primer of **Rcpp** and its aspects that are useful for computing (coming from an **R** user perspective)

Credits and links

- Dirk Eddelbuettel's website
- Advanced R by Hadley Wickham

Good **C++** references

- `learncpp.com`
- `cppreference.com`
- Effective C++ series by Scott Meyers

About **Rcpp**

- The initial versions of **Rcpp** were written by Dominick Samperi in 2005
- Dirk Eddelbuettel became maintainer in 2008, and **Rcpp** continues to be under active development until now
- **Rcpp** has one goal: making it easier to interface **C++** and **R** code

Why **Rcpp**?

- *speed*: For many tasks, **Rcpp** excels immensely where **R** struggles (e.g., loops)
- *easy to use* (for **R** users); easier than you think!
    - **Rcpp** syntactic sugar: vectorized **C++** expressions
    - seamless access to all **R** objects: vector, matrix, list,. . .
- many *extensions* (e.g., **RcppArmadillo**, **RcppEigen**, . . .)

Rcpp is popular

- Over 100 million total downloads
- As of today, there are 3126 **CRAN** packages using **Rcpp** (corresponding to 13.6% of all packages, and 61.7% of packages containing compiled code)
- Packages depending on **Rcpp**: **glmnet**, **lme4**, **rstan**, **brms**, **ranger**, **mice**, **collapse**, **rayshader**,. . . (full list here)

C++ vs **R** (non-exhaustive list)

- Don't forget the `;`
- Assignment operator: `=` (not `<-`)
- Single-line comment starts with `//`
- **C++** is zero-indexed (not one-indexed)
- **C++** is a compiled, statically typed language (**R**: interpreted, dynamically typed)
  - Code is translated into machine code that computers can execute
  - Each variable must be given a specific type (e.g. `int x = 10;` vs `x <- 10`)
  - Each function must be declared with the types of its arguments and of its return value

Defining a **C++** function

Return Type
Data type of the result returned by the function

Function Name
Actual name of the function that can be called e.g. `is_odd_cpp()`

Parameters
Variables that receive a specific data type that can be used in the function's body

Default Values
The initial values used if the parameters are not supplied on function call

```
bool is_odd_cpp(int n = 10) {
    bool v = (n % 2 == 1);
    return v;
}
```

Body
Statements in between {} that are run when the function is called

Return Value
Result made available from running body statements that matches the return type

Basic **Rcpp** usage

- When we use **Rcpp**, it compiles the **C++** code and constructs an **R** function that connects to the compiled **C++** function
    - Behind the scenes **Rcpp** creates a wrapper
    - **Rcpp** then compiles, links, and loads the wrapper
    - The function is available in **R** under what we define in **C++**
- This allows us to take advantage of the speed and efficiency of **C++** while still using **R** for the overall programming workflow

Basic **Rcpp** usage

- When we use **Rcpp**, it compiles the **C++** code and constructs an **R** function that connects to the compiled **C++** function
  - Behind the scenes **Rcpp** creates a wrapper
  - **Rcpp** then compiles, links, and loads the wrapper
  - The function is available in **R** under what we define in **C++**
- This allows us to take advantage of the speed and efficiency of **C++** while still using **R** for the overall programming workflow
- 3 core **Rcpp** functions
  - `evalCpp()`
  - `cppFunction()`
  - `sourceCpp()`

evalCpp()

- Evaluates a single **C++** expression

```
library(Rcpp)
# evalCpp("1 + 1", verbose = TRUE, rebuild = TRUE)
evalCpp("1 + 1")
```

```
## [1] 2
```

```
evalCpp("std::numeric_limits<double>::max()")
```

```
## [1] 1.797693e+308
```

cppFunction()

- Defines an **R** function from an inline **C++** function

```
cppFunction("int add(int x, int y, int z) {
  int s = x + y + z;
  return s;
}")

add(1, 2, 3)
```

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

`sourceCpp()`

- Compiles and links a stand-alone **C++** source file and exports tagged functions into **R**
- Preferred way of working with **C++**, well supported by **RStudio**

Source file structure

- Include the **Rcpp** header

```
#include <Rcpp.h>
```

- Include the namespace (optional, else call `Rcpp::` everywhere)

```
using namespace Rcpp;
```

- Prefix any functions that will be sourced into **R** with

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

- Code testing using an **R** code block

```
/*** R
# INSERT R CODE
*/
```

Data types

- All of the basic types **in R** are vectors by default.
  In **C++**: scalars
- So it is necessary to have one more level of abstraction to translate between the two
- **Rcpp** provides this with several built-in classes

R type (typeof)	**C++** type (scalar)
`integer`	`int`
`numeric`	`double`
`logical`	`bool`
`character`	`std::string`
`raw`	`char`
`complex`	`std::complex<double>`

Rcpp classes

Rcpp class	**R** typeof
`Integer`{`Vector`,`Matrix`}	`integer`
`Numeric`{`Vector`,`Matrix`}	`numeric`
`Logical`{`Vector`,`Matrix`}	`logical`
`Character`{`Vector`,`Matrix`}	`character`
`Raw`{`Vector`,`Matrix`}	`raw`
`Complex`{`Vector`,`Matrix`}	`complex`
`List`	`list` (generic vectors)
`Expression`{`Vector`,`Matrix`}	`expression`
`Environment`	`environment`
`Function`	`function`
`XPtr`	`externalptr`
`Language`	`language`
`S4`	`S4`
`...`	`...`

Vector operations (styled after **STL** operations)

- access elements via `()` (throws an error if out of bounds) or `[]` (no bounds checking)
- `length()` also aliased to `size()`
- `fill(v)` fills vector with value of v
- `begin()` pointer to beginning of vector, for iterators
- `end()` pointer to one past end of vector
- `push_back(x)` insert x at end
- `push_front(x)` insert x at beginning
- `insert(i, x)` insert x at position i
- `erase(i)` remove element at position i

Standard Template Library (**STL**)

- The standard template library (**STL**) provides a set of extremely useful data structures and algorithms
- If you need an algorithm or data structure that isn't implemented in **STL**, check out **boost**
- Main components of **STL**:
  - containers: store elements for processing (e.g., vector)
  - iterators: allow access to elements for processing
  - algorithms: perform actual processing (e.g., search, sort)

Containers

- sequence containers (ordered collections): `vector`, `deque`, `list`
- associative containers (unordered collections): `set`, `multiset`, `map`, `multimap`, `hash_set`, `hash_map`, `hash_multiset`, `hash_multimap`
- container adapters: `queue`, `priority_queue`, `stack`

Iterators

- Iterators are used to access and iterate through elements of containers (data structures)
- Iterators typically have 3 fundamental operations:
    - dereference (`*`)
    - comparison (== and `!=`)
    - increment (++)

Iterators example

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

// [[Rcpp::export]]
double my_sum(NumericVector x) {
  double total = 0;
  NumericVector::iterator i;
  for(i = x.begin(); i != x.end(); ++i) {
    total += *i;
  }
  return total;
}
```

- `.begin()` & `.end()`: iterators pointing to first and past-the-end elements
- `++`: advance
- `*`: dereferencing

Example: implement `mean()` with **Rcpp**

Example: implement mean() with **Rcpp**

R

```r
mean_r <- function(x) {
  n <- length(x)
  total <- 0
  for(i in 1:n) {
    total <- total + x[i]
  }
  return(total / n)
}
```

Example: implement mean() with **Rcpp**

R

```r
mean_r <- function(x) {
  n <- length(x)
  total <- 0
  for(i in 1:n) {
    total <- total + x[i]
  }
  return(total / n)
}
```

Rcpp

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

// [[Rcpp::export]]
double mean_rcpp(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total / n;
}
```

Quick check

```r
sourceCpp("mean.cpp")
x <- rnorm(1000)
```

```r
mean(x)
```

```
## [1] -0.02980887
```

```r
mean_r(x)
```

```
## [1] -0.02980887
```

```r
mean_rcpp(x)
```

```
## [1] -0.02980887
```

Benchmark

```r
bench::mark(
  mean(x),
  mean_r(x),
  mean_rcpp(x)
)
```

```
## # A tibble: 3 x 6
##   expression        min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 mean(x)        3.03us   3.24us   302573.        0B        0
## 2 mean_r(x)     17.88us     18us    55032.        0B        0
## 3 mean_rcpp(x)   1.23us   1.31us   757825.    5.17KB        0
```

Rcpp syntactic sugar

Rcpp syntactic sugar

From Wikipedia:

> 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.

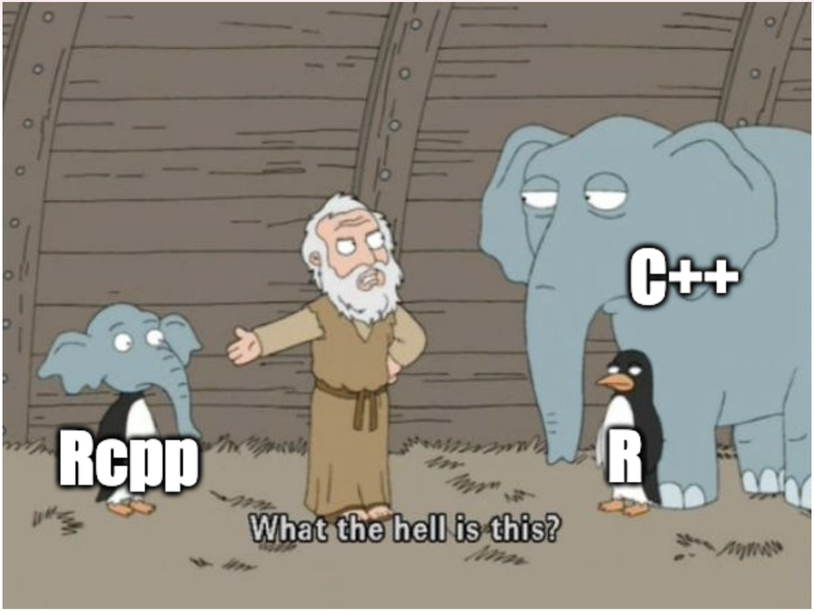
Rcpp syntactic sugar

From Wikipedia:

> 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 **R**-like “syntactic sugar” for operating on vectors in a concise way

Rcpp sugar defines functions that closely match the behavior of **R** functions of the same name.


C++
Rcpp
R
What the hell is this?

Rcpp syntactic sugar

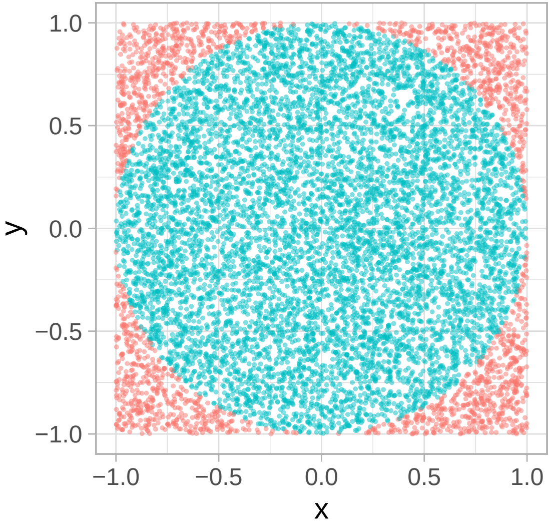
- logical operations (`ifelse()`, `all()`, `any()`,. . .)
- arithmetic (`sign()`, `sqrt()`, `exp()`,. . .)
- vector summaries (`cumsum()`, `diff()`, `pmax()`,. . .)
- scalar summaries (`mean()`, `min()`, `max()`, `sum()`,. . .)
- matrix operations (`outer()`, `diag()`, . . .)
- statistical distribution (d/p/q/r)
- other functions on vectors (`sapply()`, `rep()`, `seq_len()`, `head()`,. . .)

See here for more

Example: Estimating π using Monte Carlo simulation

- Simulate N random (x, y) points with domain as a square of side $2r$ units centered at the origin
- Consider a circle inside the same domain with radius r and inscribed into the square
- Calculate the ratio of number points inside the circle and total number of generated points

Example: Estimating π using Monte Carlo simulation



Example: Estimating π using Monte Carlo simulation

R

```r
pi_r <- function(N) {
  x <- runif(N, -1, 1)
  y <- runif(N, -1, 1)
  d <- sqrt(x^2 + y^2)
  return(4 * mean(d <= 1.0))
}
```

```r
set.seed(99)
c(pi_r(1000), pi_r(10000))
```

```
## [1] 3.1280 3.1432
```

Example: Estimating π using Monte Carlo simulation

R

```r
pi_r <- function(N) {
  x <- runif(N, -1, 1)
  y <- runif(N, -1, 1)
  d <- sqrt(x^2 + y^2)
  return(4 * mean(d <= 1.0))
}
```

```r
set.seed(99)
c(pi_r(1000), pi_r(10000))
```

```
## [1] 3.1280 3.1432
```

Rcpp

```r
cppFunction('
  double pi_rcpp(const int N) {
    NumericVector x = runif(N, -1, 1);
    NumericVector y = runif(N, -1, 1);
    NumericVector d = sqrt(x*x + y*y);
    return 4.0 * mean(d <= 1.0);
  }
')
```

```r
set.seed(99)
c(pi_rcpp(1000), pi_rcpp(10000))
```

```
## [1] 3.1280 3.1432
```

Rcpp-related packages

- **RcppArmadillo**: fast matrix computations (extends **Armadillo**)
- **RcppEigen**: eigenvalue problems (extends **Eigen**)
- **RcppGSL**: numerical computations (extends **GSL**)
- **RcppParallel**: parallel programming
- **RcppRoll**: rolling/windowed operations
- And many more. . .

Armadillo and **RcppArmadillo**

- **Armadillo** is an efficient library for linear algebra in **C++**
- Provides high-level syntax and functionality deliberately similar to **Matlab**
- Supports vectors, matrices, and cubes in dense or sparse format
- Great documentation
- **RcppArmadillo**: **Rcpp** integration for **Armadillo**, enables the use of **Rcpp** attributes and related tools
- Include the following header

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

RcppArmadillo example: linear regression

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

// [[Rcpp::export]]
Rcpp::List lm_rcpp(const arma::mat& X, const arma::colvec& y) {

  int n = X.n_rows;
  int p = X.n_cols;

  arma::colvec coef = arma::solve(X, y);
  arma::colvec e = y - X * coef;
  double s2 = std::inner_product(e.begin(), e.end(), e.begin(), 0.0)/(n - p);
  arma::colvec se = arma::sqrt(s2 * arma::diagvec(arma::pinv(arma::trans(X) * X)));

  return Rcpp::List::create(
    Rcpp::Named("coef") = coef,
    Rcpp::Named("se") = se,
  );
}
```

RcppArmadillo example: linear regression

```r
n <- 1000000
x1 <- rnorm(n)
x2 <- rnorm(n)
x3 <- rnorm(n)
x4 <- rnorm(n)
y <- 4 + 6*x1 - x2 + 10*x3 - 17*x4 - rnorm(n)
X <- cbind(1, x1, x2, x3, x4)
```

RcppArmadillo example: linear regression

```
bench::mark(
  lm(y ~ x1 + x2 + x3 + x4),
  lm.fit(X, y),
  lm_rcpp(X, y),
  check = FALSE
)
```

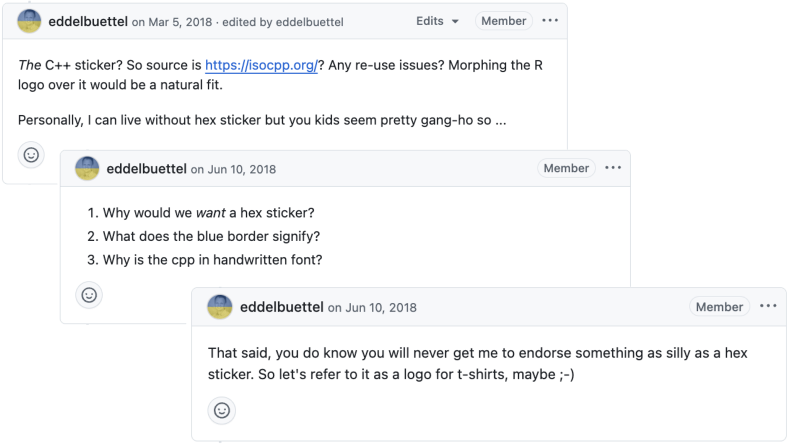
```
## # A tibble: 3 x 6
##   expression                     min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 lm(y ~ x1 + x2 + x3 + x4)  159.6ms  163.4ms      6.14     306MB    16.9
## 2 lm.fit(X, y)                67.7ms   70.9ms     12.9     76.3MB     9.19
## 3 lm_rcpp(X, y)                 63ms   63.4ms     15.7         0B     0
```

Developing packages that depend on **Rcpp**

- **Rcpp** provides a function `Rcpp.package.skeleton()` for automating the creation of a skeleton package (modeled after `package.skeleton()` in base **R**)
- Similarly, check out `RcppArmadillo::RcppArmadillo.package.skeleton()`, `RcppEigen::RcppEigen.package.skeleton()`, etc.
- Nicer documentation: `usethis::use_rcpp()` + **roxygen2** documentation

Cheers.

Fun fact: there's no such thing as an official **Rcpp** hex sticker



eddelbuettel on Mar 5, 2018 · edited by eddelbuettel Edits Member

The C++ sticker? So source is https://isocpp.org/? Any re-use issues? Morphing the R logo over it would be a natural fit.

Personally, I can live without hex sticker but you kids seem pretty gang-ho so ...

eddelbuettel on Jun 10, 2018 Member

1. Why would we *want* a hex sticker?
2. What does the blue border signify?
3. Why is the cpp in handwritten font?

eddelbuettel on Jun 10, 2018 Member

That said, you do know you will never get me to endorse something as silly as a hex sticker. So let's refer to it as a logo for t-shirts, maybe ;-)

Source: `github.com/RcppCore/Rcpp/issues/827`

Another example: Euclidean distance matrix

- Given a matrix X of dimension $n \times p$, with rows $x_i = (x_{i1}, \ldots, x_{ip})$
- Compute Euclidean distance matrix D of dimension $n \times n$ with entries

$$d_{ii'} = \sqrt{\sum_{j=1}^{p} (x_{ij} - x_{i'j})^2}\,, \qquad i, i' \in \{1, \ldots, n\}$$

R version

```r
dist_r <- function(X) {
  n <- nrow(X)
  D <- matrix(0, n, n)
  for (i in 1:n) {
    for (k in 1:i) {
      D[i,k] <- D[k,i] <- sqrt(sum((X[i,] - X[k,])^2))
    }
  }
  return(D)
}
```

Rcpp version

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

// [[Rcpp::export]]
mat dist_rcpp(const mat& X) {
  int n = X.n_rows;
  mat D(n, n, fill::zeros);
  for (int i = 0; i < n; i++) {
    for(int k = 0; k < i; k++) {
      D(i, k) = sqrt(sum(pow(X.row(i) - X.row(k), 2)));
      D(k, i) = D(i, k);
    }
  }
  return D;
}
```

Benchmark

```r
m <- as.matrix(USArrests)
bench::mark(
  dist_rcpp(m),
  dist_r(m),
  dist(m),
  check = FALSE
)
```

```
## # A tibble: 3 x 6
##   expression        min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 dist_rcpp(m)   4.63us   5.78us   162340.    19.6KB    16.2 
## 2 dist_r(m)      1.35ms   1.39ms      711.   104.1KB    22.0 
## 3 dist(m)       10.66us  12.18us    80453.    29.7KB     8.05
```