UNIVERSITY OF COPENHAGEN

Rcpp

Computational Statistics

Johan Larsson

Department of Mathematical Sciences, University of Copenhagen

March 31, 2026

Last Time: Stochastic Gradient Descent

A version of gradient descent that samples observations (functions) and converges in expectation.

Algorithm 1: Stochastic gradient descent (with mini-batches)

Data: Initial step size $t_0 > 0$
for $k = 1, 2, \ldots$ **do**
  $A_k \leftarrow$ batch of data;
  $t_k \leftarrow$ new, smaller, step size;
  $x \leftarrow x - t_k \sum_{i \in A_k} \nabla f(x_i)$;
  if *converged* **then**
    return x

Slow convergence, but cheap iterations.

We mentioned that R is ill-suited for this type of algorithm. Today, we will learn how to fix this through C++.

Today

C++

What is **compiled** code?

Why do we want to use C++?

Rcpp

We learn how to connect R and C++ using the Rcpp package.

Why R?

Why are we even using R?

Because R lets us

- work interactively,
- explore and visualize data via a rich toolset,
- easily retrieve or generate data,
- summarize and report (via RMarkdown/Quarto), and
- make use of a comprehensive ecosystem of packages.

Why *Not* R?

R is often slow because it is an **interpreted** language.

Interpreted Languages

Execution of code happens line by line.

Many functions are written in C, but you still have to call them from R.

Examples: R, Python, Matlab, Javascript.

Compiled Languages

Code is translated into machine code before execution.

Examples: C, C++, Fortran, Java

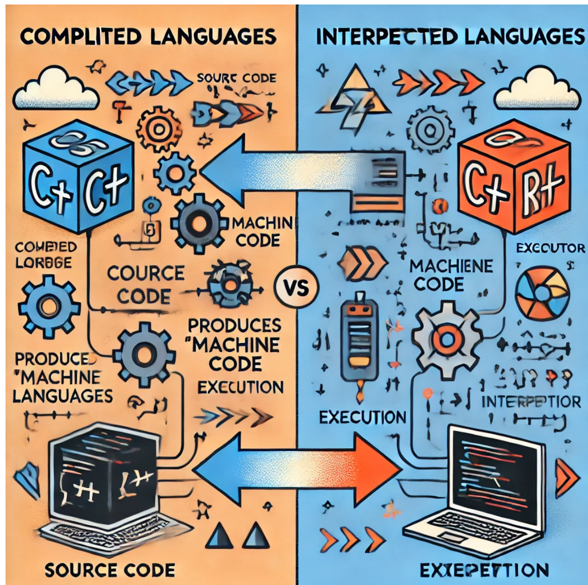


Figure 1: Compiled vs interpreted languages according to DALL-E 2 in 2024.

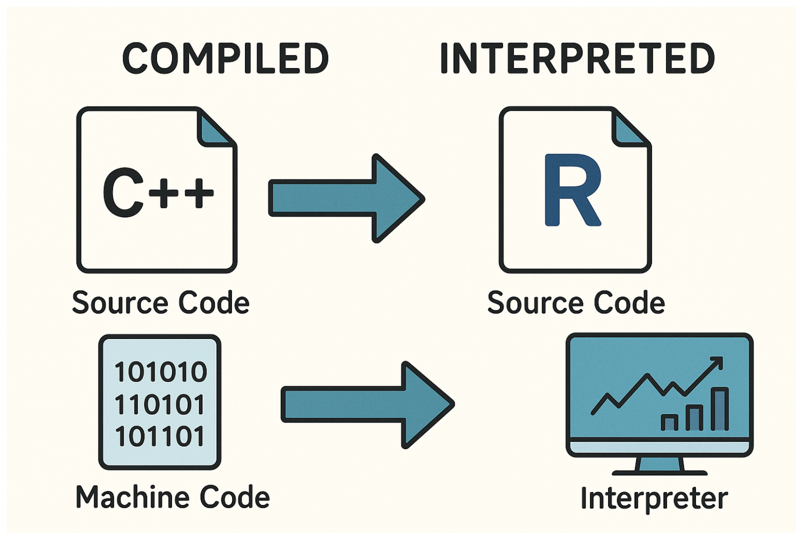


Figure 2: Compiled vs interpreted languages according to ChatGPT in 2025.

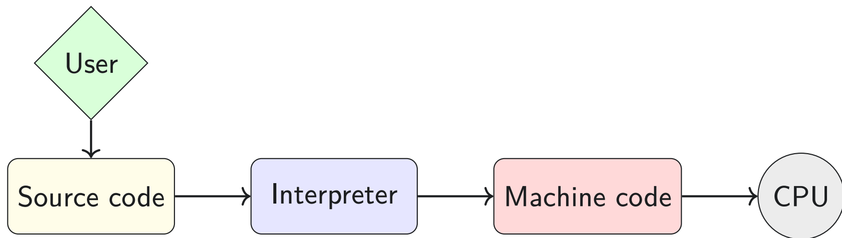


Figure 3: Interpreted languages (R, Python)

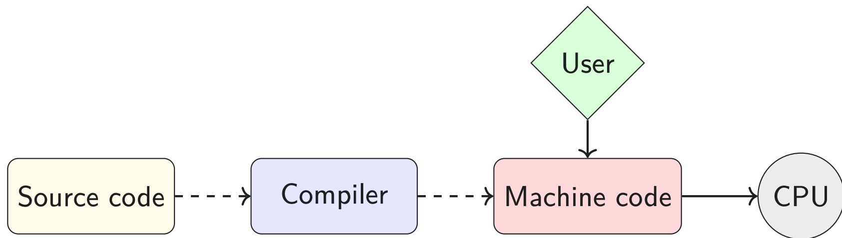


Figure 4: Compiled Languages (C++, Fortran, Rust)

C++ Code

Here's a small program in C++.

```cpp
int
add(int a, int b)
{
  return a + b;
}

int
main()
{
  std::cout << add(2, 3) << std::endl;
  return 0;
}
```

Assembly Representation

In assembly (human-readable machine) code, the code on the previous slide might look like this:

```
0000000000401130 <main>:
  401130:       55                              push   %rbp
  401131:       48 89 e5                        mov    %rsp,%rbp
  401134:       48 83 ec 10                     sub    $0x10,%rsp
  ...           ...                             ...    ...
  401149:       8b 45 f8                        mov    -0x8(%rbp),%eax
  401165:       c9                              leaveq
  401166:       c3                              retq
```

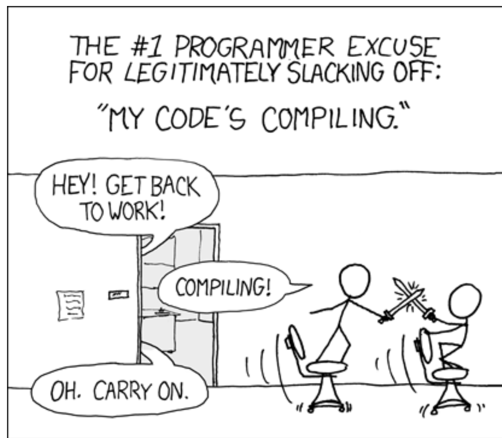
Binary Representation

In the end, this will be converted in binary (0s and 1s).

Why C++?

C++ generates performant machine code: you no longer have to be scared of for loops.

Compilation catches bugs early.



Why *Not* C++?

Compiling adds overhead and complicates debugging.

Distributing code is more involved.

R is actually built partially in compiled languages (C, Fortran).

But, writing your own C/C++ code and connecting it to R has historically been painful.

This has changed with the Rcpp package.

The Sum Function in C++

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

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

  return total;
}
```

As you can see, it looks quite similar to R code. We will explore the similarities and differences in the rest of this lecture.

What Rcpp Brings

A Link to C++
Rcpp provides an easy way to link functions written in C++ to R.

No need to manually compile and link code. Rcpp takes care of this for you!

Classes
Rcpp defines a number of classes that wraps R objects, including

- `Rcpp::NumericVector`
- `Rcpp::DataFrame`
- `Rcpp::List`

Functions
Rcpp offers many functions (sugar) that offer counterparts to R functions, including

- `Rcpp::sum()`
- `Rcpp::runif()`

Installing Rcpp

Linux

Depends on distro, but on Ubuntu call this:

```
sudo apt install r-base-dev
```

OSX

You need Xcode. Open up a terminal and call this:

```
xcode-select --install
```

Windows

Install Rtools.

Side-By-Side Comparison

R

```r
library(Rcpp)

sum_r <- function(x) {
  total <- 0

  for (i in seq_along(x)) {
    total <- total + x[i]
  }

  total
}
```

C++

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

// [[Rcpp::export]]
double
sum_cpp(NumericVector x)
{
  double total = 0;

  for (int i = 0; i < x.size(); ++i)
    total += x[i];

  return total;
}
```

Writing Code in C++ (Rcpp)

Return and Argument Types

C++ is **strongly typed**: you need to specify types for return values and arguments.

```
double   // Return type: double
f(int x) // Argument type: int
{
  return x / 3;
}
```

If your function returns nothing, use `void`:

```
void
f()
{
  Rcpp::cout << "Hello world" << std::endl;
}
```

Syntax

Assignment

You assign values using = (not <-, as in R).

Statements

Statements end with a semicolon (`;`). A new line is not enough!

Explicit Return

You need to use `return` to return a value.

```r
f <- function(x) {
  y <- x + 1
  y
}
```

```c
int
f(int x)
{
  int y = x + 1;
  return y;
}
```

Zero-Based Indexing

In C++, vector and array indices start at 0 (not 1)!

Don't do this!

```
void
ooops(Rcpp::NumericVector x)
{
  for (int i = 0; i <= x.size(); ++i) {
    x[i] = x[i] / 5;
  }
}
```

Will look at one element past the end of the vector!

Declaration Order Matters

This will not compile:[1]

```
double
f(double x)
{
  return g(x) + 1;
}

double
g(double x)
{
  return x * 2;
}
```

You need to declare `g()` before `f()`.

[1] But would work just fine in R.

In C++, you can explicitly pass arguments by reference using &.

```
void
f(double& x)
{
  x = x + 1;
}
```

```
double
g()
{
  double x = 2;
  f(x);
  return x; // returns 3
}
```

Pointers

You can also use pointers (∗).

```
void
f(double x)
{
  double* p = &x;
  *p = *p + 1;

  return;
}
```

But you don't need them with Rcpp and they are often a source of bugs.

In-Place Modification

Unlike R, C++ functions can modify their arguments in place.

```
void
f(int& x)
{
  x += 1; // identical to x = x + 1;
}

int
g()
{
  int x = 2;
  f(x);
  return x; // returns 3
}
```

Classes and Objects

C++ is an object-oriented programming language.

```
class Point
{
public: // can also use private
  double x;
  double y;
}
```

```
Point
create_point(double x, double y)
{
  Point p;
  p.x = x;
  p.y = y;

  return p;
}
```

Methods in C++

In C++, methods belong to classes.

```
class Point
{
public:
  double x;
  double y;
  double distance_from_origin() { return std::sqrt(x * x + y * y); }
};
```

You call methods using the . operator (or -> for references).

```
double
distance(Point p)
{
  return p.distance_from_origin();
}
```

C++ is Not a Functional Programming Language

But it has elements of it.

You can, for instance, pass functions as arguments.

```
int
add(int a, int b)
{
  return a + b;
}

int
compute(int x, int y, int (*func)(int, int))
{
  return func(x, y);
}
```

Function Arguments

In R, the same function can take different types of arguments.

This works differently in C++:

```cpp
double
f(int x)
{
  return x + 1;
}

void
main()
{
  f(3);   // OK (an integer, result is 4)
  f(2.5); // Look out! (Compiles, but `x` -> 2, result is 3)
  f("2"); // Error! (Cannot convert string to int.)
}
```

Function Overloading

Instead, you can define multiple functions with the same name but different argument types:

```
double
f(double x)
{
  return x + 1;
}

int
f(int x)
{
  return x + 1;
}
```

Constants

In C++, we can use `const` to declare constants.

```cpp
const double pi = 3.14159;
pi = 3.14; // Error!
```

After declaration, the variable cannot be modified.

Marginal performance improvements, but signals intent and can prevent bugs.

```r
library(Rcpp)
```

The C++ code can be passed directly to the `cppFunction()` as a text string for compilation and setting up an R interface.

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

And then the function can be called from R.

```r
one()
```

```
[1] 1
```

Source Files

But it's better to use `.cpp` files (for instance in the `src/` folder) and call `sourceCpp()` (or RStudio's UI tool).

In `R/my_r_experiment.R` (or similar), call

```r
sourceCpp(here::here("src", "my_cpp_functions.cpp"))
```

```
my_project/
├── scripts/
│   └── my_r_experiment.R     # R script
├── src/
│   └── my_cpp_functions.cpp # Source for C++ functions
└── my_rproject.Rproj         # RStudio project file
```

Rcpp Syntax

Each file should start with

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

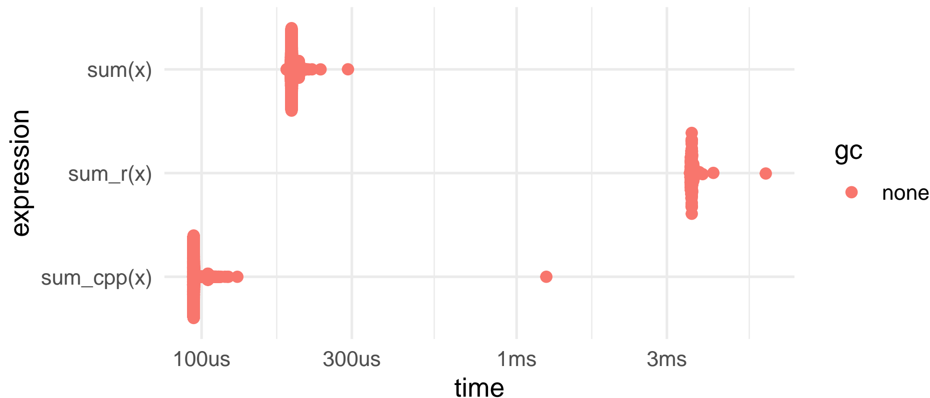
Before each function you want to export to R, add the comment

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

Sum Functions

Let's see if we can beat R's own `sum()` (See here for the implementation.)

```r
x <- runif(1e5)
bench::mark(sum(x), sum_cpp(x), sum_r(x)) |>
  plot()
```



References in Rcpp

You might think that this would create a copy of the R input vector.

```
// [[Rcpp::export]]
double
sum_cpp_ref(Rcpp::NumericVector x)
{
  return Rcpp::sum(x);
}
```

But no, `Rcpp::NumericVector` is a reference to the R object.

Pitfall: Modifying R Objects

```cpp
// [[Rcpp::export]]
void
uh_oh(Rcpp::NumericVector x)
{
  x = x + 1.0;
}
```

```r
x <- 1.5:4.5
uh_oh(x)
x
```

```
[1] 2.5 3.5 4.5 5.5
```

The input vector `x` has been modified in place! You can use `Rcpp::clone()` to avoid this.

Const References

If you don't want to modify the input, you can pass by const reference.

```
// [[Rcpp::export]]
double
sum_cpp_const_ref(const Rcpp::NumericVector& x) {
  double total = 0.0;
  const int n = x.size();

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

  return total;
}
```

In C++, this can make a big difference, but with Rcpp, it makes no difference.

Printing to the R console can be done using `Rcpp::Rcout`.

```cpp
// [[Rcpp::export]]
void
print_cpp(const Rcpp::NumericVector& x) {
  Rcpp::Rcout << "x = " << x << std::endl;
}
```

```r
print_cpp(1:5)
```

```
x = 1 2 3 4 5
```

Errors and Warnings

You can signal errors and warnings using `Rcpp::stop()` and `Rcpp::warning()`.

```
// [[Rcpp::export]]
double
safe_log(double x) {
  if (x <= 0) {
    Rcpp::stop("x must be positive");
  }

  return std::log(x);
}
```

Premature optimization is especially useless with C++ since the compiler will often optimize the code for you anyway.

What you see in the source code is **not** what you get.

Copies and temporaries are often optimized away.

Example: Von Mises Rejection Sampling

R Version

```r
rvonmises <- function(n, kappa) {
  y <- numeric(n)
  for (i in 1:n) {
    reject <- TRUE
    while (reject) {
      y0 <- runif(1, -pi, pi)
      u <- runif(1)
      reject <- u > exp(kappa * (cos(y0) - 1))
    }
    y[i] <- y0
  }
  y
}
```

Rcpp Version

```cpp
// [[Rcpp::export]]
Rcpp::NumericVector
rvonmises_cpp(const int n, const double kappa)
{
  Rcpp::NumericVector y(n);
  int i = 0;

  while (i < n) {
    double y0 = R::runif(-M_PI, M_PI);
    bool accept = R::runif(0, 1) <= exp(kappa * (cos(y0) - 1));
    if (accept) {
      y[i] = y0;
      i++;
    }
  }
  return y;
}
```

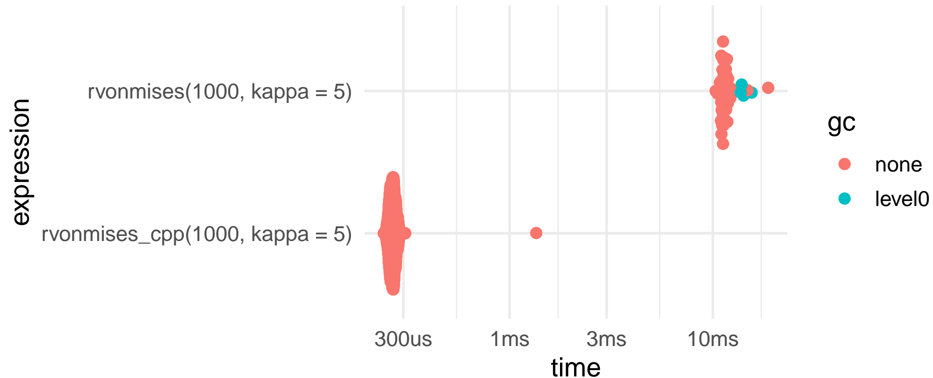


Figure 5: Benchmark of R and Rcpp implementations of the von Mises rejection sampler.

The Standard Template Library (STL)

A library of generic algorithms, data structures, and iterators for C++.

It's like R's base package, but for C++.

Accessed via the `std` namespace.

Provides a number of useful functions for working with vectors and other containers.

Efficiently Growing Vectors

Unlike R and Rcpp vector, `std::vector()` grows efficiently.

```
std::vector<int> x;

for (int i = 0; i < 10; ++i) {
  x.emplace_back(i);
}
```

Armadillo is a user-friendly C++ library for linear algebra.

```
#include <RcppArmadillo.h>

// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
arma::mat
armaMatMul(const arma::mat& A, const arma::mat& B)
{
  return A * B;
}
```

Intuitive syntax, depends on BLAS and LAPACK.

Eigen is another popular C++ library for linear algebra.

```
#include <RcppEigen.h>

// [[Rcpp::depends(RcppEigen)]]
// [[Rcpp::export]]
Eigen::MatrixXd
eigenMatMul(const Eigen::MatrixXd& A, const Eigen::MatrixXd& B)
{
  return A * B;
}
```

More low-level than Armadillo, but does not depend on BLAS/LAPACK.

Parallelization

Parallelization in R comes with overhead. The situation in C++ is better.

Two main ways to parallelize C++ code in R are OpenMP and the RcppParallel package.

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

// [[Rcpp::depends(RcppParallel)]]
// [[Rcpp::export]]
double
parallel_function(const Rcpp::NumericVector& x)
{
  // Your parallel code here
}
```

We won't cover the details here, but expect much better leverage than using R's parallelization tools.

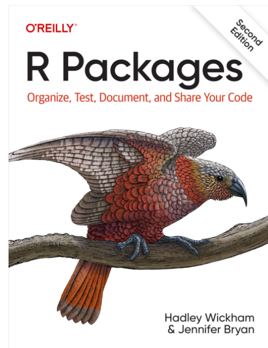
Like tests, Rcpp code works best in packages.

We will cover this in our final lecture.

Not as complicated as you might think!

R Packages

Great book to get started; free and available online at https://r-pkgs.org.



The cpp11 Package

Nowadays, you can also use the cpp11 package, which works similarly to Rcpp but is more lightweight.

Won't cover it here, but has some advantages and disadvantages compared to Rcpp.

Advantages

- Compiles faster
- Grows vectors efficiently
- Enforces copy-on-write semantics (like R)

Disadvantages

- Smaller ecosystem
- Fewer built-in functions (sugar)
- No (stable) Armadillo or Eigen bindings

Getting Help

- The many official vignettes
- Rcpp For Everyone by Masaki E. Tsuda
- Rcpp Gallery
- The unofficial Rcpp API documentation

1. Implement a function using Rcpp that computes the standard deviation of a (numeric) vector.[2]
2. Test your function against R's `sd()` function to make sure it works correctly.
3. Benchmark your function against R's `sd()` function to see if it is faster.
4. Handle cases when the input vector is of length 0 or 1.

[2] Don't use 'Rcpp::sd()', please!

Summary

R is an interpreted language, which makes it easy to use but often slow.

You can use C++ to speed up your code, and it is not as difficult as you might think.

But there are pitfalls, so be careful!

Debugging C++ code is tricky and failures can crash R.

Course Evaluations

The course evaluations are open!

Please take a few minutes to fill them out. Your feedback is valuable!

Next Two Lectures

Variations on Stochastic Gradient Descent

We expand on the stochastic gradient descent algorithm, looking at momentum and adaptive step sizes.

R Packages and Wrap-up (Final Lecture)

We learn how to set up R packages, which is the best way to distribute R and Rcpp code.