

Armadillo and Eigen: A Tale of Two Linear Algebra Libraries

Mauricio Vargas Sepúlveda (ORCID 0000-0003-1017-7574)
Department of Political Science, University of Toronto
Munk School of Global Affairs and Public Policy, University of Toronto

Corresponding author: m.sepulveda@mail.utoronto.ca

Last updated: 2024-08-31 17:51

Contents

1	Abstract	1
2	Introduction	1
3	Syntax and benchmarks	2
4	Comparison with R packages	6
5	Cases where Armadillo and Eigen stand out	7
6	Conclusion	9
	References	10

1 Abstract

This article introduces `cpp11eigen`, a new R package that integrates the powerful Eigen C++ library for linear algebra into the R programming environment. This article provides a detailed comparison between Armadillo and Eigen speed and syntax. The `cpp11eigen` package simplifies a part of the process of using C++ within R by offering additional ease of integration for those who require high-performance linear algebra operations in their R workflows. This work aims to discuss the tradeoff between computational efficiency and accessibility.

2 Introduction

R is widely used by non-programmers (Wickham et al. 2019), and this article aims to introduce benchmarks in a non-technical yet formal manner for social scientists. Our goal is to provide a fair comparison between Eigen and Armadillo, being both highly efficient linear algebra libraries written in C++. We do it by using `cpp11eigen` and `cpp11armadillo`.

`Eigen` emphasizes flexibility and speed, while `Armadillo` focuses on a balance between speed and easy of use.

‘`RcppEigen`’, introduced in 2011, integrates Eigen with R through the `Rcpp` package, enabling the use of C++ for performance-critical parts of R code. ‘`RcppArmadillo`’ has a similar goal (Sanderson and Curtin 2016; Eddelbuettel and Sanderson 2014). At the time of writing this article, 247 CRAN packages depend on ‘`RcppEigen`’, 755 on ‘`RcppArmadillo`’ (Lee 2024), and therefore these are highly successful packages.

`cpp11eigen` is an independent project that aims to simplify the integration of R and C++ by using ‘`cpp11`’, an R package that eases using C++ functions from R. A distinctive characteristics of `cpp11eigen` is the vendoring capability, meaning that it allows to copy its code into a project, making it a one-time dependency with a fixed and stable code until it is updated, and it is useful in restricted environments such as servers and clusters (Wickham et al. 2019; Vaughan, Hester, and François 2023).

`cpp11armadillo` offers similar features and both libraries are useful in cases where vector-

ization (e.g., applying an operation to a vector or matrix as a whole instead of looping over each element) is not possible or challenging. A detailed discussion and examples about why and when (and when not) rewriting R code in C++ is useful can be found in Burns (2011) and Vargas Sepúlveda (2023).

We followed four design principles when developing `cpp11eigen`, same as `cpp11armadillo` (Vargas Sepúlveda and Schneider Malamud 2024): column oriented, package oriented, header-only, and vendoring capable.

3 Syntax and benchmarks

One possibility is to start by creating minimal R packages with the provided templates.

```
remotes::install_github("pachadotdev/cpp11armadillo")
remotes::install_github("pachadotdev/cpp11eigen")

cpp11eigen::create_package("armadillobenchmark")
cpp11eigen::create_package("eigenbenchmark")
```

Comparing numerical libraries requires to write equivalent codes. For instance, in R we can use `apply()` functions while in C++ we need to write a `for` loop, and this allows a fair comparison between the two libraries. However, R has heavily optimized functions that also verify the input data, such as `lm()` and `glm()`, that do not have a direct equivalent in Armadillo or Eigen, and for a fair comparison the options are to write a simplified function for the linear model in R or to write a more complex function in C++.

The ATT benchmark, is a set of functions that can be rewritten using Armadillo and Eigen with relative ease, and test has the advantage of being well-known and widely used in the R community.

The first test in the ATT benchmark is the creation, transposition and deformation of an $N \times N$ matrix ($2,500 \times 2,500$ in the original test). The R code comparable to the Armadillo code is:

```
matrix_calculation_01_r <- function(n) {
  a <- matrix(rnorm(n * n) / 10, ncol = n, nrow = n)
  b <- t(a)
  dim(b) <- c(n / 2, n * 2)
  a <- t(b)
  return(OL)
}
```

The Armadillo code is very similar:

```
#include <cpp11.hpp>
#include <cpp11armadillo.hpp>

using namespace arma;
using namespace cpp11;

[[cpp11::register]] int matrix_calculation_01_arma_(const int& n) {
  mat a = randn<mat>(n,n) / 10;
  mat b = a.t();
  b.reshape(n/2, n*2);
  a = b.t();
  return 0;
}
```

The Eigen code requires to create a function to draw random numbers from a normal distribution but it has a built-in function for the uniform distribution:

```
#include <cpp11.hpp>
#include <cpp11eigen.hpp>
#include <random>

using namespace Eigen;
using namespace cpp11;

std::mt19937& random_normal() {
  static std::random_device rd;
  static std::mt19937 gen(rd());
  return gen;
}

[[cpp11::register]] int matrix_calculation_01_eigen_(const int& n) {
  std::normal_distribution<double> d(0, 1);
```

```

MatrixXd a = MatrixXd::NullaryExpr(n, n, [&]() {
    return d(random_normal());
}) / 10;

// for the uniform distribution this is simpler
// MatrixXd a = MatrixXd::Random(n, n) / 10;

MatrixXd b = a.transpose();
b.resize(n / 2, n * 2);
return 0;
}

```

The functions do not move data between R and C++, and this is intentional to focus on the performance of the linear algebra libraries and not adding overhead from data transfer in the benchmarks. Each function creates a matrix and conducts equivalent operations on it. The returned value is zero in R and C++ in case that the functions run without errors. The benchmarks were conducted on a ThinkPad X1 Carbon Gen 9 with the following specifications:

- Processor: Intel Core i7-1185G7 with eight cores
- Memory: 16 GB LPDDR4X-4266
- Operating System: Pop!_OS 22.04 based on Ubuntu 22.04
- R Version: 4.4.1
- BLAS Library: OpenBLAS 0.3.20

The median times for the adapted and comparable implementations of the ATT benchmarks are as follows:

Table 1: Matrix calculation

Operation	Time (s)	Rank
$2,400 \times 2,400$ matrix ^{1,000} - Armadillo	0.188	1
$2,400 \times 2,400$ matrix ^{1,000} - Eigen	0.301	2
$2,400 \times 2,400$ matrix ^{1,000} - R	0.325	3
$2,800 \times 2,800$ cross-product matrix - Armadillo	0.398	1
$2,800 \times 2,800$ cross-product matrix - R	0.444	2

Operation	Time (s)	Rank
$2,800 \times 2,800$ cross-product matrix - Eigen	1.151	3
Creation and modification of a $2,500 \times 2,500$ matrix - Armadillo	0.204	1
Creation and modification of a $2,500 \times 2,500$ matrix - Eigen	0.232	2
Creation and modification of a $2,500 \times 2,500$ matrix - R	0.294	3
Linear regression over a $3,000 \times 3,000$ matrix - Armadillo	0.459	1
Linear regression over a $3,000 \times 3,000$ matrix - R	5.303	2
Linear regression over a $3,000 \times 3,000$ matrix - Eigen	8.809	3
Sorting of 7,000,000 values - Armadillo	0.663	1
Sorting of 7,000,000 values - Eigen	0.691	2
Sorting of 7,000,000 values - R	0.759	3

Table 2: Matrix functions

Operation	Time (s)	Rank
Cholesky decomposition of a $3,000 \times 3,000$ matrix - Armadillo	0.608	1
Cholesky decomposition of a $3,000 \times 3,000$ matrix - R	0.709	2
Cholesky decomposition of a $3,000 \times 3,000$ matrix - Eigen	2.902	3
Determinant of a $2,500 \times 2,500$ matrix - Armadillo	0.293	1
Determinant of a $2,500 \times 2,500$ matrix - R	0.303	2
Determinant of a $2,500 \times 2,500$ matrix - Eigen	0.562	3
Eigenvalues of a 640×640 matrix - Armadillo	0.367	1
Eigenvalues of a 640×640 matrix - R	0.369	2
Eigenvalues of a 640×640 matrix - Eigen	1.629	3
Fast Fourier Transform over 2,400,000 values - Eigen	0.14	1
Fast Fourier Transform over 2,400,000 values - R	0.23	2
Fast Fourier Transform over 2,400,000 values - Armadillo	0.294	3
Inverse of a $1,600 \times 1,600$ matrix - Armadillo	0.312	1
Inverse of a $1,600 \times 1,600$ matrix - R	0.324	2
Inverse of a $1,600 \times 1,600$ matrix - Eigen	0.758	3

Table 3: Programmation

Operation	Time (s)	Rank
3,500,000 Fibonacci numbers calculation - Eigen	1.4×10^{-1}	1
3,500,000 Fibonacci numbers calculation - Armadillo	1.7×10^{-1}	2
3,500,000 Fibonacci numbers calculation - R	1.7×10^{-1}	3
Creation of a $3,000 \times 3,000$ Hilbert matrix - Eigen	4.6×10^{-6}	1
Creation of a $3,000 \times 3,000$ Hilbert matrix - Armadillo	5.9×10^{-2}	2

Operation	Time (s)	Rank
Creation of a $3,000 \times 3,000$ Hilbert matrix - R	1.5×10^{-1}	3
Creation of a 500×500 Toeplitz matrix - Eigen	7.9×10^{-7}	1
Creation of a 500×500 Toeplitz matrix - Armadillo	4×10^{-4}	2
Creation of a 500×500 Toeplitz matrix - R	2.6×10^{-3}	3
Escoufier's method on a 45×45 matrix - Armadillo	2.4×10^{-2}	1
Escoufier's method on a 45×45 matrix - Eigen	3.2×10^{-2}	2
Escoufier's method on a 45×45 matrix - R	1.4×10^{-1}	3
Grand common divisors of 400,000 pairs - Eigen	2.1×10^{-2}	1
Grand common divisors of 400,000 pairs - Armadillo	2.3×10^{-2}	2
Grand common divisors of 400,000 pairs - R	1.884	3

The results reveal that Armadillo leads in most of the benchmarks, but Eigen is particularly faster in some tests such as the Fast Fourier Transform. R is the second or third in all benchmarks, but it is important to note that R comes with an additional advantage in terms of simplified syntax and the ability to run the code without compiling it.

These tests are not exhaustive, and we must be cautious when interpreting the results. The ATT benchmark is a good starting point, but it does not cover mundane tasks such as data manipulation, and it is important to consider the tradeoff between computational efficiency and ease of use.

4 Comparison with R packages

The syntax and speed differences posit a similar case to the tradeoff between using `dplyr` and `data.table` (Wickham et al. 2019; Barrett et al. 2024), where `dplyr` is easier to use but `data.table` is faster. `dplyr` was not designed to be fast but `data.table` was not designed to be easy to use. For instance, the code to obtain the grouped means by number of cylinders in the `mtcars` dataset is:

```
# dplyr
mtcars %>%
  group_by(cyl) %>%
  summarise_all(mean)

# data.table
```

```
as.data.table(mtcars)[, lapply(.SD, mean), by = cyl]
```

The benchmark for the grouped means reveals that `dplyr` has a median time of 2.7 ms and `data.table` has a median time of 600 μ s, and this means that `dplyr` is four times slower than `data.table` at this task. The syntax of `dplyr` is easier to understand for non-programmers, but `data.table` can be equally expressive for users who are familiar with its syntax.

The tests for Armadillo and Eigen reveal that, for repeated and computationally intensive tasks, rewriting R code in C++ can lead to significant performance improvements, but it comes at the cost of learning a new syntax.

As with `dplyr` and `data.table`, the choice between Armadillo and Eigen As an example, the `economiccomplexity` package (Vargas Sepulveda 2020) uses base R depends on the user’s needs and preferences. For instance, Armadillo or Eigen can be ideal to work with a $1,000,000 \times 1,000,000$ matrix but R can be more suitable for a $1,000 \times 1,000$ matrix, and something similar applies to `dplyr` that is suitable for a 2-4 GB CSV files or SQL data but `data.table` is more suitable for 100 GB CSV datasets.

5 Cases where Armadillo and Eigen stand out

Vargas Sepulveda (2020) uses base R and the Matrix package to calculate the Balassa index and provides international trade data for 226 countries and 785 exported commodities.

Let $X \in \mathbb{R}^{C \times P}$ be a matrix with entries $x_{c,p}$ that represents the exports of country c in product p , from this matrix the Balassa indices matrix is calculated as:

$$B = ([X \oslash (X \vec{1}_{P \times 1})]^t \oslash [X^t \vec{1}_{C \times 1} \oslash (\vec{1}_{C \times 1}^t X \vec{1}_{P \times 1})])^t, \quad (1)$$

where \oslash denotes element-wise division and t denotes transposition.

This is the same as the Balassa index for country c and product p :

$$B_{cp} = \frac{x_{cp}}{\sum_c x_{cp}} / \frac{\sum_p x_{cp}}{\sum_c \sum_p x_{cp}} \quad (2)$$

What is often used is to produce a zeroes and ones matrix S defined as:

$$s_{c,p} = \begin{cases} 1 & \text{if } b_{cp} > 1 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

(3) can be implemented in base R as:

```
balassa_r <- function(X) {  
  B <- t(t(X / rowSums(X)) / (colSums(X) / sum(X)))  
  B[B < 1] <- 0  
  B[B >= 1] <- 1  
  B  
}
```

The C++ code using `cpp11armadillo` is:

```
#include <cpp11.hpp>  
#include <cpp11armadillo.hpp>  
  
using namespace cpp11;  
using namespace arma;  
  
[[cpp11::register]] doubles_matrix<> balassa_arma_  
  (const doubles_matrix<>& x) {  
  mat X = as_Mat(x);  
  
  mat B = X.each_col() / sum(X, 1);  
  B = B.each_row() / (sum(X, 0) / accu(X));  
  B.elem(find(B < 1)).zeros();  
  B.elem(find(B >= 1)).ones();  
  
  return as_doubles_matrix(B);  
}
```

The C++ code using `cpp11eigen` is:

```
#include <cpp11.hpp>  
#include <cpp11eigen.hpp>  
  
using namespace cpp11;  
using namespace Eigen;
```

```

[[cpp11::register]] doubles_matrix<> balassa_eigen_(
  const doubles_matrix<>& x) {
  MatrixXd X = as_Matrix(x);

  MatrixXd B = X.array().rowwise() / X.rowwise().sum().array();
  B = B.array().colwise() / (X.colwise().sum().array() / X.sum());
  B = (B.array() < 1).select(0, B);
  B = (B.array() >= 1).select(1, B);

  return as_doubles_matrix(B);
}

```

If we use UN COMTRADE data for the year 2020 for 234 countries and 5,386 countries (United Nations 2023), we can observe that Armadillo and Eigen are around two times faster than base R at obtaining the Balassa matrix, and this includes the time to move the data between R and C++:

Table 4: Balassa indices

Operation	Time (s)	Rank
Balassa indices Eigen	0.013	1
Balassa indices Armadillo	0.014	2
Balassa indices R	0.026	3

The rest of the methods in Vargas Sepulveda (2020) involve recursion and eigenvalues computation, and these tasks were already covered in the ATT benchmark, meaning that the same speed gains can be expected as in the Balassa matrix.

6 Conclusion

Armadillo and Eigen can be highly expressive, these are flexible libraries once the user has learned the syntax, and these languages have data structures that do not exist in R that help to write efficient code. Eigen and `cpp11eigen` do not simplify the process of writing C++ code for R users but excels at computationally demanding applications. Armadillo and `cpp11armadillo`, on the other hand, provides a balance between speed and ease of use,

and it is a good choice for users who need to write C++ code that is easier to modify and maintain.

References

- Barrett, Tyson, Matt Dowle, Arun Srinivasan, Jan Gorecki, Michael Chirico, and Toby Hocking. 2024. *Data.table: Extension of ‘Data.frame’*. <https://CRAN.R-project.org/package=data.table>.
- Burns, Patrick. 2011. *The r Inferno*. Lulu.
- Eddelbuettel, Dirk, and Conrad Sanderson. 2014. “RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra.” *Computational Statistics & Data Analysis* 71 (March): 1054–63. <https://doi.org/10.1016/j.csda.2013.02.005>.
- Lee, Clement. 2024. *Crandep: Network Analysis of Dependencies of CRAN Packages*. <https://CRAN.R-project.org/package=crandep>.
- Sanderson, Conrad, and Ryan Curtin. 2016. “Armadillo: A Template-Based c++ Library for Linear Algebra.” *Journal of Open Source Software* 1 (2): 26. <https://doi.org/10.21105/joss.00026>.
- United Nations. 2023. “UN Comtrade.” <https://comtradeplus.un.org/>.
- Vargas Sepúlveda, Mauricio. 2020. “Economiccomplexity: Computational Methods for Economic Complexity.” *Journal of Open Source Software* 5 (46): 1866. <https://doi.org/10.21105/joss.01866>.
- Vargas Sepúlveda, Mauricio. 2023. *The Hitchhiker’s Guide to Linear Models*. Leanpub. <https://leanpub.com/linear-models-guide>.
- Vargas Sepúlveda, Mauricio, and Jonathan Schneider Malamud. 2024. “Cpp11armadillo: An R Package to Use the Armadillo C++ Library.” arXiv. <https://doi.org/10.48550/arXiv.2408.11074>.
- Vaughan, Davis, Jim Hester, and Romain François. 2023. *Cpp11: A c++11 Interface for r’s c Interface*. <https://CRAN.R-project.org/package=cpp11>.
- Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D’Agostino McGowan, Romain François, Garrett Grolemond, et al. 2019. “Welcome to the Tidy-

verse.” *Journal of Open Source Software* 4 (43): 1686. <https://doi.org/10.21105/joss.01686>.