cpp11armadillo: an R package to use the Armadillo C++ library

Mauricio Vargas Sepulveda[a], Jonathan Schneider Malamud[b]

[a] *University of Toronto, Munk School of Global Affairs and Public Policy and Department of Political Science, m.sepulveda@mail.utoronto.ca*
[b] *University of Toronto, Department of Electrical and Computer Engineering*

Abstract

This article introduces 'cpp11armadillo', an R package that integrates the highly efficient Armadillo C++ linear algebra library with R through the 'cpp11' interface. Designed to offer significant performance improvements for computationally intensive tasks, 'cpp11armadillo' simplifies the process of integrating C++ code into R. This package is particularly suited for R users requiring efficient matrix operations, especially in cases where vectorization is not possible. Our benchmarks demonstrate substantial speed gains over native R functions and Rcpp-based setups.

Keywords: R, C++, Armadillo, linear algebra, benchmarking

Metadata

Nr.	Code metadata description	Metadata
C1	Current code version	v0.4.0
C2	Permanent link to code/repository used for this code version	`https://github.com/pachadotdev/cpp11armadillo`
C4	Legal Code License	Apache License 2.0.
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	C++, R.
C7	Compilation requirements, operating environments & dependencies	C++11 and R 4.0.0 at least.
C8	If available Link to developer documentation/manual	`https://pacha.dev/cpp11armadillo/`
C9	Support email for questions	m.sepulveda@mail.utoronto.ca

Table 1: Code metadata (mandatory)

1. Motivation and significance

As R continues to grow in popularity for statistical computing and data analysis, it can create bottlenecks when working with large datasets. The need to interface with lower-level languages like C++ to bypass these bottlenecks has led to the development of tools like 'Rcpp' [1], and more recently, 'cpp11' [2].

'cpp11armadillo' is a novel package that builds upon the existing Armadillo C++ library [3] to provide high-level access to efficient linear algebra operations directly from R. The existing 'RcppArmadillo' [4] already offers this, and 'cpp11armadillo' offers an alternative with modern C++ features, where we highlight vendoring, which can improve performance and simplify the its usage.

This package is not designed for the average R user. Instead, it is intended for users in academia or industry who require intensive linear algebra computations. 'cpp11armadillo' provides tools to bridge the gap between ease of use and the computational efficiency demanded by such workloads.

'RcppArmadillo' is a popular package for this purpose, it has existed for nearly a decade, 'cpp11armadillo' offers a different approach with similar performance. Rewriting code from 'RcppArmadillo' to 'cpp11armadillo' or vice versa only requires changes to the function's arguments and return types, while the main code actual computation uses the Armadillo library syntax.

Armadillo provides ready-made functions and data structures for linear algebra, bringing some of the ease of use from R or Python to C++ while leveraging its additional speed. Using 'cpp11armadillo' does not require al-

tering the operating system or R configurations, it uses the system's default numerical libraries, and it is compatible with OpenBLAS and Intel MKL that offer additional performance improvements.
Armadillo-based implementations, while they offer significant performance improvements, have the downside of requiring the user to write C++ code and learn a new syntax. While Armadillo largely simplifies C++ writing, it does not completely remove C++ usage barriers for some users, and the learning curve is only justified if the performance gains are substantial for repeated tasks.
'cpp11armadillo' simplifies C++ integration by leveraging the 'cpp11' package [2], which is designed to reduce the complexity of including C++ code in R. Its core features include:

- High-Performance Matrix Operations: By using Armadillo, 'cpp11armadillo' allows users to perform matrix multiplications, inversions, and decompositions with minimal overhead. These operations are, for some cases, several times faster than their R counterparts.

- Simplified Syntax: Armadillo offers a user-friendly syntax similar to that of 'MATLAB' [3], making it accessible to R users without requiring an extensive background in C++.

- Modern C++ Support: 'cpp11armadillo' supports C++11 and newer standards, enabling efficient memory management, parallel computations, and advanced templates, all integrated into an R workflow.

- Integration with R: The package allows direct manipulation of R data structures (e.g., vectors, matrices) in C++, providing a seamless workflow between the two languages without requiring manual data conversion (e.g., such as exporting data to CSV files and continously reading them in R to use the C++ output or vice versa).

After installation, linear algebra functions in R remain unaltered, the end user will be able to write C++ code and benefit from more flexible data structures and faster operations. Besides detecting the number of cores available in the system, 'cpp11armadillo' does not require any additional configuration, and it will automatically use the operating system default numerical library, which the user can change if needed.
The R programming language was not designed for high-performance computing, it was created to offer ready-made statistical functions for the end-user, and its interpreted nature can lead to slow execution times for computationally intensive tasks [5,6]. The same applies to Python and other high-level languages.

Compiled languages like C++ offer significant performance improvements due to their direct access to hardware resources and efficient memory management. The counterpart is that C++ has a steep learning curve and it is less user-friendly than R. C++ goals are different, it is designed to be fast and efficient, and it is not focused on statistical computing but on general-purpose programming [7].
However, bottlenecks in R can be solved by using vectorized operations instead of writing loops, which treats data objects as a whole instead of element-wise [8]. There are cases where vectorization is not possible or challenging to implement [8], and this is where 'cpp11armadillo' comes in and it offers data structures and functions that are not available in R that facilitate the implementation of loops and other operations [3,6].
Even in spite of these difficulties to work with large datasets or computationally intensive tasks, R is a growing popularity language for statistical computing and data analysis, and its community has created tools to integrate it with SQL databases for memory-efficient data access [9]. There are multiple ways to measure a language popularity, and one of them is the number of R questions in Stack Overflow, which has been increasing over the years as shown in the following plot [10].

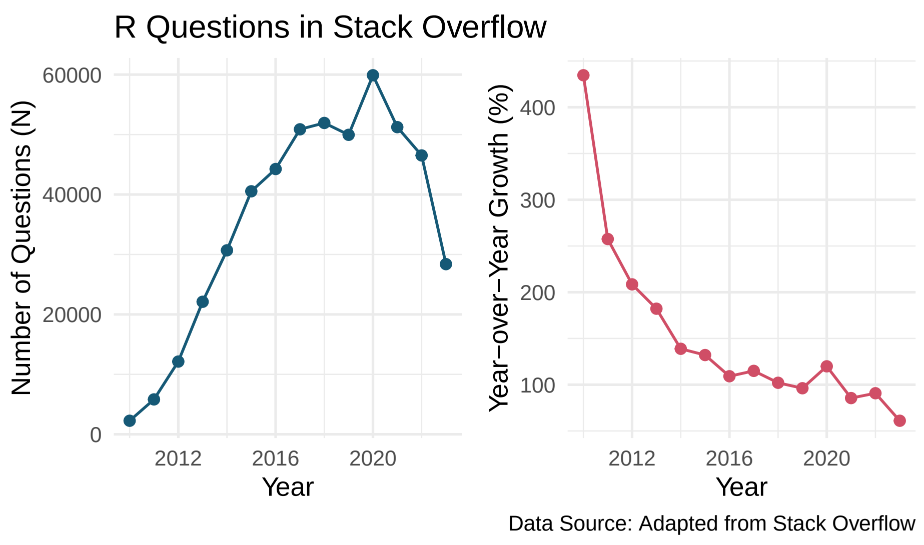

Figure 1: R Questions in Stack Overflow increase from 2,260 in 2010 to 28,385 in 2023 with a peak of 59,895 in 2020.

Large Language Model tools, including ChatGPT, explain a decline in the popularity and the number of R questions in Stack Overflow and other coding forums [11]. Another explanation in the decline in last years in the plot is that already answered questions are not asked again, and the community has access to existing answers through search engines.

2. Software description

‘cpp11armadillo’ uses ‘cpp11’ to leverage modern C++ features to provide a seamless interface between R and the Armadillo library, offering high-performance linear algebra operations with minimal overhead.
‘cpp11’ is a modern rewrite of the C++ interface for R, designed to improve safety, performance, and ease of use. It enforces copy-on-write semantics to prevent unintended modifications to data, ensuring that changes to objects do not affect other references. ‘cpp11’ provides safer access to R’s C API, reducing runtime errors in C++ code. It also supports ALTREP objects for efficient memory management and deferred computations, making it ideal for handling large datasets. By using UTF-8 strings throughout, ‘cpp11’ ensures robust handling of datasets created in different countries where encodings vary [2].
Built on C++11 features like smart pointers and lambdas, ‘cpp11’ offers a more straightforward and efficient implementation compared to ‘Rcpp’. Its header-only design eliminates ABI compatibility issues, making it easier to integrate and manage in projects. ‘cpp11’ also compiles faster, uses less memory, and grows vectors more efficiently, optimizing performance when dealing with large amounts of data. These improvements make ‘cpp11’ a powerful, streamlined tool for developers who need reliable, high-performance C++ bindings for R [2].
‘cpp11armadillo’ offers vendoring, something not available in ‘RcppArmadillo’. Vendoring is a well-known concept in the Go community, and it consists in copying the dependency code directly into a project’s source tree. This approach ensures that dependencies remain fixed and stable, preventing any external changes from inadvertently breaking the project. While vendoring offers stability, it comes with trade-offs. The primary advantage is that updates to the ‘cpp11armadillo’ library will not disrupt existing code, and it also copies ‘cpp11’ C++ headers. However, the drawbacks include an increase in package size and the loss of automatic updates, meaning that bug fixes and new features will only be available when manually updated.
Vendoring which can simplify the installation process and reduce dependency issues, especially when working in environments with restricted access to the Internet. For instance, the Niagara Supercomputer that we use at the University of Toronto has restricted access to the Internet, and vendoring can simplify the installation process. In other words, vendoring allows the

package creator to provide a dependency-free package that can be installed in any environment without requiring the end user to install ‘cpp11’ nor ‘cpp11armadillo’. This approach makes ‘cpp11armadillo’ a dependency for the developer but not for the end user.

3. Software functionalities

To use ‘cpp11armadillo’, users must first install the package from CRAN or GitHub. The package includes the Armadillo library, no additional installation is required. The following code shows how to install the package:

```r
install.packages("cpp11armadillo")

# or
remotes::install_github("pachadotdev/cpp11armadillo")
```

Once installed, users can use the provided package template function to create a new package that uses C++ code with Armadillo. The package template includes simple examples and all the necessary files to compile the code and install the new R package. The following code shows how to create a new package:

```r
# subdir + package name
# subdir can be "." to create the package in the current directory
cpp11armadillo::pkg_template("pkgtemplate", "myownpackage")
```

Then the user can open a new RStudio session to modify the package template, or type `setwd("pkgtemplate")` to open the package in the current R session. We highly recommend that users start by reading the readme file in the package template directory to understand the package structure and the necessary steps to add functions and build the package.
The package vignettes cover the directories organization and the necessary steps to build an R package with relatively low setup efforts.
It is important to note that ‘cpp11armadillo’ only work within an R package, and it is not possible to compile individual C++ scripts on the fly. This design choice ensures that the R and C++ code is organized following the organization described in [12].
The package creator needs to export the functions for the end user. For example, for the following C++ function:

```
double plus_one_(const double &x) {
  return x + 1;
}
```

The function must be exported and it can be documents in the R package as follows:

```
#' Add one to a number
#' @param x A number
#' @return The number plus one
#' @export
plus_one <- function(x) {
  plus_one_(x)
}
```

4. Illustrative examples

In order to evaluate the performance of ‘cpp11armadillo’, we compared it with native R functions and ‘RcppArmadillo’. To provide a point of reference, we also used ‘NumPy’, a popular numerical computing library for Python, which is known for its performance and ease of use. For all the benchmarks, we used a value of $n = 10,000$.
Compiling the benchmarking functions took 4 seconds using ‘cpp11armadillo’, and 9 seconds using RcppArmadillo. These functions are quite simple, but this comparison is consistent with the build times for general usage R packages, as shown in the following table:

Table 2: Benchmark comparison of ‘cpp11’ and ‘Rcpp’ [2].

Package	‘cpp11’ compile time	‘Rcpp’ compile time
haven	7.1s	17.4s
readr	81.0s	124.1s
roxygen2	4.2s	17.3s
tidyr	3.3s	14.3s

Table 1 reveals that R packages using ‘cpp11’ compile faster than using ‘Rcpp’. This is the result of a full refactor of the C++ parts of these packages to use ‘cpp11’ instead of ‘Rcpp’ to expose C++ functions to R.

4.1. Eigenvalues

We focused on a single test consisting in obtaining the eigenvalues for a dense (symmetric) matrix of size $n \times n$.
The R code for the computation is as follows:

```r
bench_r <- function(m) {
  eigen(m)$values
}
```

An equivalent C++ code is as follows:

```cpp
doubles bench_eig_cpp11armadillo_(const doubles_matrix<>& m) {
  mat M = as_Mat(m);
  colvec Y = eig_sym(M);
  return as_doubles(Y);
}
```

The Python code to have a referential computation time is as follows:

```python
def bench_eigenvalues_py(M):
    return np.linalg.eigvals(M)
```

'cpp11armadillo' has a speed-up relative to 'RcppArmadillo' of approximately 0.1 times and a speed-up relative to base R of approximately 2.4 times. The following table shows the median execution time for one hundred thousand runs of the three implementations compared to 'NumPy' in Python:

Table 3: Execution Time for the Eigenvalues Benchmark.

'cpp11armadillo'	'RcppArmadillo'	Base R	NumPy (Python)
21.86 s	22.1 s	52.42 s	109.56 s

Table 2 shows a tie between 'cpp11armadillo' and 'RcppArmadillo' in terms of speed, and both are faster than base R and 'NumPy'. For this particular operation, the ranking is: (1) C++, (2) R, and (3) Python.

4.2. Multi-operation Expression

We focused on a single test consisting in a multi-operation expression that computes $P^T Q^{-1} R$, where P is a vector of length n, Q is a diagonal square matrix of size $n \times n$ filled with random values, and R is a vector of length n. This test was adapted from the official Armadillo documentation [13].
The R code for the computation is as follows:

```r
bench_r <- function(p, q, r) {
  as.numeric(t(p) %*% solve(diag(q)) %*% r)
}
```

An equivalent C++ code is as follows:

```cpp
double bench_cpp11armadillo_(const doubles &p, const doubles &q,
                             const doubles &r) {
  colvec P = as_Col(p)
  colvec Q = as_Col(q)
  colvec R = as_Col(r)
  return as_scalar(trans(P) * inv(diagmat(Q)) * R);
}
```

The Python code to have a referential computation time is as follows:

```python
def bench_multi_py(p, q, r):
    return float(np.dot(np.dot(p.T, np.linalg.inv(np.diag(q))), r))
```

'cpp11armadillo' has a speed-up relative to 'RcppArmadillo' of approximately 1.7 times and a speed-up relative to base R of approximately 140,000 times. The following table shows the median execution time for one hundred thousand runs of the three implementations compared to NumPy in Python:

Table 4: Execution Time for the Multi-operation Expression Benchmark.

'cpp11armadillo'	'RcppArmadillo'	'NumPy' (Python)	Base R
26.1 us	45.44 us	2.60 s	3.75 s

Table 3, unlike Table 2, reveals a clear difference between C++ functions exposed to R using 'cpp11' and 'Rcpp'. The ranking is: (1) 'cpp11', (2) 'Rcpp', (3) R, and (4) Python.
The benchmarks were run on the Niagara Supercomputer at the University of Toronto using forty cores and R, Python, and the Armadillo library compiled with the Intel MKL library, and the benchmarks were run sequentially.
The benchmarks were intentionally run on the Niagara Supercomputer to ensure an accurate and controlled benchmarking environment, free from potential distortions caused by background processes like automatic updates,

antivirus scans, cloud synchronization, or internet browsing that are common on regular laptops or desktops [14]. This setup allows us to isolate the performance of the package itself without interference.
It is worth noting that these benchmarks can be run on a regular laptop because a dense matrix of double precision numbers of $10,000 \times 10,000$ uses 800 MB in RAM plus the extra memory for processing.
The reason to use the median execution time is to provide a robust measure of the execution time, and it is less sensitive to outliers and the stored results show there is variation in the execution time for repeated runs.
The benchmarks do not cover sparse matrices, which are covered with examples in 'cpp11armadillo' documentation and follow the same syntax as dense matrices. The Armadillo library can solve large-scale linear algebra problems efficiently. These functions can be used to solve problems in areas such as machine learning, optimization, and scientific computing that involve a matrix of, for example, $10^6 \times 10^6$ elements. However, sparse matrices should only be used when the matrix is mostly empty in the sense that most of its entries are zero. For sparse matrices, the Armadillo library provides functions to convert between dense and sparse matrices, and the library documentation warns that sparse data objects are lighter but use compression techniques that can slow down the computation [15].

5. Impact

Researchers can explore questions involving complex matrix operations or large datasets that introduce significant bottlenecks in R, and our alternative scales with relative ease, as we showed in our benchmarks where we were able to run the same code in a server with minimal setup efforts due to vendoring.
The improved computational efficiency facilitates simulation studies, and users can test and implement advanced linear algebra algorithms that export the results to R for posterior analysis and visualization, and the package can be used without needing a separate C++ compilation step.
Our benchmarks demonstrate substantial speed improvements compared to existing equivalent implementations, enabling researchers to test and modify computations with large datasets without a significant computational overhead.
The package structure reduces the complexity of extending R with C++, allowing users to focus on writing linear algebra instead of compiler configurations to write high-performance code.
The software supports a structured workflow for combining R scripts with efficient underlying C++ computations, enhancing reproducibility in computational research.

By design, ‘cpp11armadillo’ eliminates much of the boilerplate coding required for R and C++ integration, allowing users to focus on algorithm design. The simplified usage, besides scaling, facilitates collaboration between R users and C++ developers, promoting interdisciplinary work.
The package is ideal for scientists who rely on matrix-heavy computations. It may interest users from disciplines like biology, finance, and physics, where high-performance computation is highly desirable.
Companies working with R for data analysis and simulation could integrate the package into their pipelines, benefiting from faster computations, and this is compatible with the Apache license, which allows commercial use. By lowering barriers to high-performance R and C++ integrations, the package could inspire startups focused on domain-specific software solutions or consulting services based on efficient computational methods.

6. Conclusion

‘cpp11armadillo’ provides a simple and efficient way to integrate C++ code with R, leveraging the ‘cpp11’ package and the Armadillo library. It simplifies the process of writing C++ code for R users, allowing them to focus on the logic of the algorithm rather than the technical details of the integration. It can help to solve performance bottlenecks in ‘R’ code by using the efficient linear algebra operations provided by Armadillo in cases where vectorization is challenging.

7. Acknowledgments

We thank the reviewers for their valuable feedback and suggestions. We also thank Professor Salma Emara who taught us C++ and clearly cares about her students.

References

[1] D. Eddelbuettel, R. Francois, Journal of Statistical Software 40 (2011) 1–18.

[2] D. Vaughan, J. Hester, R. François, cpp11: A C++11 Interface for R’s C Interface, 2023.

[3] C. Sanderson, R. Curtin, Journal of Open Source Software 1 (2016) 26.

[4] D. Eddelbuettel, C. Sanderson, Computational Statistics & Data Analysis 71 (2014) 1054–1063.

[5] H. Wickham, M. Averick, J. Bryan, W. Chang, L.D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T.L. Pedersen, E. Miller, S.M. Bache, K. Müller, J. Ooms, D. Robinson, D.P. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, H. Yutani, Journal of Open Source Software 4 (2019) 1686.

[6] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, 2024.

[7] B. Stroustrup, (2012).

[8] P. Burns, The R Inferno, Lulu, 2011.

[9] H. Wickham, J. Ooms, K. Müller, rpostgres: C++ Interface to PostgreSQL, 2023.

[10] Stack Overflow, (2024).

[11] S. Kabir, D.N. Udo-Imeh, B. Kou, T. Zhang, in: Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, Association for Computing Machinery, New York, NY, USA, 2024, pp. 1–17.

[12] H. Wickham, J. Bryan, R Packages, O'Reilly, 2023.

[13] C. Sanderson, (2024).

[14] D. Beyer, S. Löwe, P. Wendler, International Journal on Software Tools for Technology Transfer 21 (2019) 1–29.

[15] C. Sanderson, R. Curtin, Mathematical and Computational Applications 24 (2019) 70.