armadillo: An R Package to Use the Armadillo C++ Library

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

Jonathan Schneider Malamud
Department of Electrical and Computer Engineering, University of Toronto

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

Last updated: 2024-08-18 16:45

Contents

1 Abstract

This article introduces 'armadillo', a new R package that integrates the powerful Armadillo C++ library for linear algebra into the R programming environment. Targeted primarily at social scientists and other non-programmers, this article explains the computational benefits of moving code to C++ in terms of speed and syntax. We provide a comprehensive overview of Armadillo's capabilities, highlighting its user-friendly syntax akin to MATLAB and its efficiency for computationally intensive tasks. The 'armadillo' 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 bridge the gap between computational efficiency and accessibility, making advanced linear algebra operations more approachable for R users without extensive programming backgrounds.

2 Introduction

R is widely used by non-programmers (Wickham et al. 2019), and this article aims to introduce computational concepts in a non-technical yet formal manner for social scientists. Our goal is to explain when and why moving code to C++ is beneficial in terms of speed or syntax and how to do it using 'armadillo', our novel Armadillo and R integration for linear algebra.

Armadillo is a C++ library designed for linear algebra, emphasizing a balance between performance and ease of use. C++ is highly efficient for computationally intensive tasks but lacks built-in data structures and functions for linear algebra operations. Armadillo fills this gap by providing an intuitive syntax similar to MATLAB (Sanderson and Curtin 2016).

'RcppArmadillo', introduced in 2010, integrates Armadillo with R through the Rcpp package, enabling the use of C++ for performance-critical parts of R code (Eddelbuettel and Sanderson 2014). 'RcppArmadillo' is a widely successful project, and at the time of writing this article, there are around 1,100 packages on CRAN that depend on it.

'armadillo' 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, and it is aligned

with the tidyverse philosophy of simplicity and user-centric design (Wickham et al. 2019; Vaughan, Hester, and François 2023).

'armadillo' is useful in cases where vectorization (e.g., applying an operation to a vector or matrix as a whole rather than looping over each element) is not possible or challenging, and it can help to solve some bottlenecks as it simplifies the task of rewriting R code that involves linear algebra as C++ code.

'armadillo' can be orders of magnitude faster, computing operations in parallel, which is especially useful for large objects. When vectorization is possible, using R's built-in functions is more efficient than writing loops in R, and the time of writing the same in C++ justifies to continue to use vectorized operations in R.

For cases where vectorization is applicable, Burns (2011) provides a good introduction. 'armadillo' is relevant in a project where the same operation is repeated many times and, at the same time, the computation time saved by using C++ is greater than the time spent writing and fixing C++ code.

We followed four design principles when developing 'armadillo':

1. **Column-oriented**: It follows the column-major order for its internals and all the documentation and examples, consistent with Hansen (2022), where vectors are column vectors.
2. **Package oriented**: It is designed to be used in an R package, which is the recommended way to organize code in medium and large scale projects.
3. **Header-only**: No separate actions are required; it only requires to include 'armadillo' as a dependency.
4. **Vendoring-capable**: It provides a dedicated function to copy the entire codebase into R packages, providing the option to make it a one-time dependency. This feature allows to run code in restricted environments (e.g., where installing packages from CRAN or GitHub is blocked by a firewall or available for administrators only).

3 Interpreted and compiled languages

R is an interpreted language, meaning that the code is executed line by line when you run a part or the totality of a script. One advantage of interpreted languages is that they are easier to debug because the code can be run by parts to isolate errors. Another advantage is that, assuming that all dependencies are solved, the code can be run in any computer without additional configurations. Other interpreted languages are Python, MATLAB, and Wolfram.

C++ is a compiled language, meaning that to run the code, it must be converted to an executable file containing instructions that the processor can understand. This allows the compiler to optimize the code for the specific hardware it is running on, making it faster than interpreted languages. The main disadvantage of compiled languages is that they are harder to debug because it is not possible to run the code by parts as when running R code blocks on-the-fly. C++ code requires a compiler (e.g. 'gcc' or 'clang') to produce an executable file, which is a software separate from the editor (e.g. 'RStudio' or 'VS Code') that translates the code to machine code, and it is not possible to use an executable produced on Windows with UNIX and vice versa. Other compiled languages are C, FORTRAN, and Java.

R internals consist in functions written in C and FORTRAN that the end user has ready-made to run scripts. These functions, while available in R source code, are usually not of interest for the end user as these already have a proven stability, even in corner cases, and those are written with memory and speed efficiency at the expense of syntax and flexibility. C inherent steep learning curve motivated C++ creation, and C++ is a superset of C with additional features that make it easier to use and that provide flexibility. C++ is a high-level language that can be used for a wide range of purposes, including parts of operating systems (e.g., Windows), internet browsers (e.g., Firefox) and streaming platforms (e.g., YouTube), and it is particularly useful for computationally intensive tasks. Transitioning from R to C++ involves adapting to several differences in conventions and flexibility regarding data types and operations.

FORTRAN was released in 1957, C in 1972, C++ in 1985 and R in 1993. All of these languages have updated releases and standards in the last decade. The time argument stresses

that our examples are mostly instructional, we are not trying to reinvent the wheel, these examples emerge from the idea of highlighting the differences that emerge when transitioning from R to C++.

There are contemporary versions of the C++ standard, such as C++14, C++17, and C++20, but C++11 is still widely used. The "11" part in 'cpp11' refers to the fact that it follows the C++11 standard, which was released in 2011 and introduced several features that make C++ relatively easier to use and more flexible, despite the fact that it is possible to use it with codes written using newer C++ standards as long as a compatible compiler is installed.

4 Linear algebra libraries

Linear algebra libraries are essential for scientific computing, and they provide functions for matrix operations, such as matrix multiplication, inversion, and decomposition. These libraries are written in C or C++ and are optimized for speed without compromising stability. Some of these libraries are the Linear Algebra Package (LAPACK) and the Basic Linear Algebra Subprograms (BLAS). As C and C++, these libraries have a long history and time-tested correctness, and the first LAPACK release was in 1992 and the first BLAS release was in 1979.

When you run `sessionInfo()` in R, it shows lines similar to:

```
Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.10.0
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0
```

Or similar to:

```
Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.20.so;
LAPACK version 3.10.0
```

This information reveals that R is using the BLAS and LAPACK libraries for linear algebra

operations. BLAS and LAPACK are used internally when executing functions such as `lm()`, `solve()` or `%*%`.

Armadillo also calls BLAS and LAPACK for linear algebra operations, and this can be verified in its source code that contains the lines:

```
#include "armadillo/def_blas.hpp"
#include "armadillo/def_lapack.hpp"
```

While it is possible to use BLAS or LAPACK directly in C++ code, Armadillo provides efficient routines that largely simplify the syntax and the time involved to write useable code, and it combines operations to reduce intermediate steps and temporary objects when possible. Armadillo is not just about speed, in some cases it is also about feasibility, as it allows to use the available memory more efficiently and to run tasks that would be impossible with the available memory if done in a naive way that involves creating temporary objects and copies.

Just as an elemental example, without additional details, the following code computes the dot product of two vectors using BLAS:

```
// 2x2 matrix written as a long array
double a[4] = {1.0, 2.0, 3.0, 4.0};

// matrix to write the transposed result to
double at[4];

// dimensions
int r = 2, c = 2;

// transpose
cblas_dgeam(CblasRowMajor, CblasTrans, CblasTrans, rows, cols,
  1.0, matrix, cols, 0.0, nullptr, cols, transposed_matrix, rows);
```

The same result can be obtained with R with the following code:

```
A <- matrix(c(1, 2, 3, 4), nrow = 2)
At <- t(A)
```

The same result can be obtained with Armadillo with the following code:

```
Mat<double> A = {{1.0, 2.0}, {3.0, 4.0}};
Mat<double> At = A.t();
```

Writing code directly in BLAS or LAPACK can be challenging, and it is very challenging to beat BLAS or LAPACK performance. This is why R and Armadillo use them for the internal computation and provide functions with a simplified syntax for the end user. The interested readers can explore Zhang and Kroeker (2024) to read about OpenBLAS, an even faster version of BLAS.

5 R vectorization and loops

Some of R's vectorized functions include `sum()`, `mean()`, `apply()` and its variants, and `map()` and its variants in the purrr package.

Instead of using the `mean()` function, the mean of a vector can be computed with a loop in R:

```
x <- c(1, 2, 3, 4, 5)

numerator <- 0
denominator <- length(x)

for (i in 1:denominator) {
  y <- y + x[i]
}

numerator / denominator
```

The previous loop is inefficient because it involves writing more code and it is slower because it goes through each element one by one. In R or any other interpreted language (e.g. Python), loops are slower than vectorized operations.

An example of efficient vectorization is the `pmax()` function to obtain the element-wise maximum for vectors or matrices. The same can be done for two input matrices with a loop in R that has the advantage of being explicit in terms of the operations performed, but it is slower than the vectorized function:

```r
A <- matrix(c(1, 2, 3, 4), nrow = 2)
B <- matrix(c(4, 3, 2, 1), nrow = 2)

C <- matrix(0, nrow = 2, ncol = 2)

for (i in 1:2) {
  for (j in 1:2) {
    C[i, j] <- max(A[i, j], B[i, j])
  }
}
```

The problem with this loop is that it is particularly slow, for two 1000x1000 matrices filled with `rnorm()`, it takes takes 422 miliseconds to run while the `pmax()` function takes 7 miliseconds, meaning that the implemented loop is twenty times slower.

Loops should be used in computations where the one step depends on the previous steps. One example of this is the Gram-Schmidt method to obtain an orthogonal matrix from a square matrix X, in which case the N vector depends on the previous $N-1$ computed vectors, and it consists of the following algorithm (Strang 1988):

- Step 1: Construct a matrix X of dimension $M \times N$ with the vectors to be orthonormalized as column vectors.
- Step 2: Construct a matrix U of the same dimension as X filled with zeroes to store the orthonormal basis later.
- Step 3: Replace the first column of U with the vector $u_1 = x_1/\|x_1\|$, where x_1 is the first column of X and $\|x_1\|$ is the euclidian norm that is the square root of the sum of the squared m coordinates x_{1i} given by $\sqrt{\sum_{i=1}^{m} x_{1i}^2} = \sqrt{x_1^t x_1}$.
- Step 4: For the remaining $M-1$ vectors $x_{j>1}$, calculate the projection of the vector x_j onto the vector u_j and subtracts it from x_j, this is $x_j = x_j - \sum_{i=1}^{j-1}(u_i^t x_j/u_i^t u_i)u_i$.
- Step 5: Normalize each $x_{j>1}$ to unit length as $x_j = x_j/\|x_j\|$ and replace it in the remaining columns of U.

In R this can be written as:

```r
X <- matrix(c(3,4,4,4), nrow = 2)
U <- matrix(0, nrow = 2, ncol = 2)

N <- ncol(X)

U[, 1] <- X[, 1] / sqrt(sum(X[, 1]^2))

for (j in 2:N) {
  v <- X[, j]
  for (i in 1:(j - 1)) {
    u <- U[, i]
    v <- v - (crossprod(u, v) / crossprod(u, u)) * u
  }
  U[, j] <- v / sqrt(sum(v^2))
}
```

This result is correct according to Wolfram Alpha, that returns the column vectors $c_1 = (3/5, 4/5)$ and $c_2 = (4/5, -3/5)$.

6 Common pitfalls when transitioning from R to C++

6.1 Syntax and defaults

Semicolons are mandatory in C++. C++ defaults to int for numbers, while R defaults to double. In C++, you must declare the variable type.

The following R code treats x as a double (e.g. a decimal number) unless otherwise specified:

```r
# double
x <- 200
function(x) {
  x + 100
}

# integer
x <- 200L
function(x) {
  x + 100L # L = integer
}
```

C++ needs to declare the data type of the variable even if the number does not have a decimal point:

```
// integer
int x = 200;
double function(double y) {
  return y + 100;
}

// double
double x = 200.0; // x = 200 also works
double function(double y) {
  return y + 100.0; // y + 100 also works
}
```

In R, variable names can be recycled in any function without issues. In C++ the example, the function has an argument `y` instead of `x` because `x` was previously declared in the global scope, and the code would not compile if the function had an argument `x`.

6.2 Lack of a terminal Shell

C++ lacks a dedicated terminal shell and cannot be used as a scientific calculator like R. C++ code must be compiled and executed. An analogy for this is that R is like ready-made Eggo waffles, while C++ is like making waffles from scratch by mixing the ingredients.

6.3 Data types

C++ requires explicit library inclusion for strings, vectors, matrices, lists, and data frames, which are not natively available. R has built-in data structures for these types. 'cpp11' provides wrappers for these data structures, which facilitate R and C++ integration. This is similar to R's tibble, a data structure not natively available in base R but that is provided by the 'tibble' package, and that enhances the data frame structure (Müller and Wickham 2023).

In addition to 'cpp11' vectors and matrices, Armadillo provides its own data structures for linear algebra operations. Armadillo data structures are more flexible and allow for a highly

readable and concise syntax, this is why 'armadillo' exists, because R cannot directly use Armadillo data structures unless there is a package that translates them to R data structures. This is similar to R's SQL integration, where the 'RPostgres' package contains C++ code capable of translating a SQL query to a tibble (Wickham, Ooms, and Müller 2023).

6.4 Operations and indexing

C++ has useful operators that do not exist in R (e.g., ++, +=, and *=). C++ is zero-indexed, whereas R is one-indexed.

The following R code sums the numbers in the sequence 5 7 4 4 2

```
x <- c(5, 7, 4, 4, 2)

for (i in 1:5) {
  x[i] <- x[i] + i
}
```

An equivalent C++ code is:

```
int x[5] = {5, 7, 4, 4, 2};

for (int i = 0; i < 5; ++i) {
  x[i] = x[i] + (i + 1);
}
```

7 Computational complexity

Computational complexity refers to the number of steps required to solve a problem. It is expressed in terms of the size of the input data, n, and the number of operations required to solve the problem.

The same algorithm implemented in different programming languages will retain its computational complexity. Rewriting an R code in C++ may reduce the chronological time to run a function, but the complexity will remain the same. The only possibility to reduce the computational complexity is to write an equivalent algorithm that implement different steps

to do the same, which is why reduced forms are important in the field of Econometrics and others.

C++ is faster for loops because the time (e.g., seconds) it takes for each iteration of the loop is usually lower than in R, but for equivalent loops in C++ and R the total number of operations is the same. One of the notations, the big-O notation is expressed as $O(f(n))$, where $f(n)$ is a function that describes the upper bound of the number of operations required to solve the problem.

For the previous loop to compute the mean of a vector of n coordinates (or elements), the computational complexity is $O(n)$ because there are n elements to sum to create the numerator plus one division by the denominator given by the number of elements, and this results in involves $n+1$ operations which is still in the order of $O(n)$. Functions that require n, $n+10$ or $n-2$ operations are still in the order of $O(n)$.

Making the loop to obtain the mean worse in terms of efficiency is useful to clarify the computational complexity. Consider the following loop in R:

```
x <- c(1, 2, 3, 4, 5)

numerator <- 0
denominator <- length(x)

for (i in 1:denominator) {
  y <- (y + x[i]) / denominator
}
```

This loop still has a complexity of $O(n)$, but it is less efficient because it involves $n > 1$ divisions, leading to kn total operations with $k > 1$. For the big-O notation, $2n$, $3n$ or $200n + 100$ are also in the order of $O(n)$.

Regardless of the type of operation, big-O counts the number of operations. For example, the geometric mean is computationally more expensive because multiplication (and division) are slower than sums, but its complexity is still $O(n)$.

Consider the following loop in C++:

```
int sum = 0;
int n = 9;
for (int i = 0; i < n; i++) {
  sum++;
}
```

The total number of operations is $3n + 1$, because there is one operation to set the initial value of `sum`, one to set the initial value of `n`, one to set the initial value of `i`, n verifications that $i < n$, n increments of `i`, and n increases of `sum` by a value of one. The complexity is still in the order of $O(n)$.

Other operations can be more expensive, such as matrix multiplication, which has a complexity of $O(n^3)$ for two $n \times n$ matrices, and finding the inverse of a matrix, which has a complexity of $O(n^3)$ for a $n \times n$ matrix.

8 Reduced forms

Two function can reach the same result but with a different number of operations, and therefore different complexity. Adapting from Emara (2024), here are two function written by Dante and Virgilio to compute 2^n using recursion:

```
int Dante(int n) {
  if (n == 0) {
    return 1;
  } else {
    return (Dante(n-1) + Dante(n-1));
  }
}

int Virgilio(int n) {
  if (n == 0) {
    return 1;
  } else {
    return (2 * Virgilio(n-1));
  }
}
```

The two functions are correct and equivalent, but the number of operations is different.

`Dante()` uses two recursive calls for each step, creating `2^n` total calls, and therefore its complexity is `O(2^n)`. `Virgilio()` uses one recursive call for each step, creating `n` total calls, and therefore its complexity is `O(n)`.

The second function is a reduced form of the first, and in practical terms it is the same as simplifying $x^2 + 2x + 1$ to $(x + 1)^2$ to reduce the number of operations required to obtain the result. R and Armadillo internals make an extensive use of reduced forms to optimize the code, and in general it is not simple to write a reduced form in any language, especially for complex operations such as regression models where the reductions can introduce a wide range of issues (e.g., unintended divisions by zero in corner cases).

9 Gauss-Jordan C++ implementation

Consider the following system of linear equations from Vargas Sepúlveda (2023):

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6.50 \\ 7.50 \\ 8.50 \end{bmatrix}$$

The system can be solved with row operations to obtain the inverse matrix:

$$\overset{\text{row 2}-\text{row 1}}{\rightarrow} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right] \overset{\text{row 3}-\text{row 2}}{\rightarrow} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & 1 & -1 & 1 \end{array}\right]$$

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 6.50 \\ 7.50 \\ 8.50 \end{bmatrix} = \begin{bmatrix} 6.50 \\ 1.00 \\ 7.50 \end{bmatrix}$$

The same can be done with a naive implementation of the Gauss-Jordan algorithm that has complexity $O(n^3)$ (Strang 1988). It should serve as a starting point to understand the syntax and the data structures.

```cpp
#include <cpp11.hpp>

using namespace cpp11;

[[cpp11::register]] doubles_matrix<> invert_matrix_(doubles_matrix<> a) {
  // Check dimensions
  int n = a.nrow(), m = a.ncol();
  if (n != m) {
    stop("X must be a square matrix");
  }

  // Copy the matrix
  writable::doubles_matrix<> acopy(n, n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      acopy(i, j) = a(i, j);
    }
  }

  // Create the identity matrix as a starting point for Gauss-Jordan
  writable::doubles_matrix<> ainv(n, n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      ainv(i, j) = (i == j) ? 1.0 : 0.0;
    }
  }

  // Overwrite Ainv by steps with the inverse of A
  // (find the echelon form of A)
  for (int i = 0; i < m; i++) {
    double aij = acopy(i, i);

    // Divide the row by the diagonal element
    for (int j = 0; j < m; j++) {
      acopy(i, j) /= aij;
      ainv(i, j) /= aij;
    }

    // Subtract the row from the other rows
    for (int j = 0; j < m; j++) {
      if (i != j) {
        aij = acopy(j, i);
        for (int k = 0; k < m; k++) {
          acopy(j, k) -= acopy(i, k) * aij;
```

```
          ainv(j, k) -= ainv(i, k) * aij;
        }
      }
    }
  }

  return ainv;
}

[[cpp11::register]] doubles_matrix<> multiply_inverse_(doubles_matrix<> a,
                                                       doubles_matrix<> b) {
  // Check dimensions
  int n1 = a.nrow(), m1 = a.ncol(), n2 = b.nrow(), m2 = b.ncol();
  if (n1 != m1) {
    stop("a must be a square matrix");
  }
  if (n1 != n2) {
    stop("b must have the same number of rows as a");
  }
  if (m2 != 1) {
    stop("b must be a column vector");
  }

  // Obtain the inverse
  doubles_matrix<> ainv = invert_matrix_(a);

  // Multiply ainv by b
  writable::doubles_matrix<> x(n1, 1);
  for (int i = 0; i < n1; i++) {
    x(i, 0) = 0.0;
    for (int j = 0; j < n1; j++) {
      x(i, 0) += ainv(i, j) * b(j, 0);
    }
  }

  return x;
}
```

The code above includes the 'cpp11' library (`#include <cpp11.hpp>`) and loads the corresponding namespace (`using namespace cpp11`) to simplify the notation (e.g., typing `doubles_matrix<>` instead of `cpp11::doubles_matrix<>`). It declares two functions, `invert_matrix_` reads a matrix from R in a direct way (by making a copy) and returns

its inverse and `multiply_inverse_` that reads from R (`a` and `b`) and C++ (`ainv`), and solves a system of linear equations.

These functions use `doubles_matrix<>` and `writable::doubles_matrix<>`. The `writable::` prefix must be added every time the object will be modified later or the code will not compile. The code was written in a modular way, organized in two dedicated functions, and resulting object is assigned to a `doubles_matrix<>` that goes from C++ to R. In the functions, `stop()`, `nrow()` and `ncol()` are not a part of stock C++, these are provided by 'cpp11'.

10 Gauss-Jordan Armadillo implementation

The previous Gauss-Jordan implementation can be largely simplified by using the Armadillo library.

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

using namespace arma;
using namespace cpp11;

[[cpp11::register]]
doubles_matrix<> invert_matrix_(const doubles_matrix<>& a) {
  Mat<double> Acopy = as_Mat(a);
  Mat<double> Ainv = inv(Acopy);
  return as_doubles_matrix(Ainv);
}

[[cpp11::register]]
doubles_matrix<> multiply_inverse_(const doubles_matrix<>& a,
                                   const doubles_matrix<>& b) {
  Mat<double> Acopy = as_Mat(A);
  Mat<double> Bcopy = as_Mat(b);
  Mat<double> X = inv(Acopy) * Bcopy;
  return as_doubles_matrix(X);
}
```

The `inv()` function and the `*` operator verify the dimensions of the matrices. This example shows that Armadillo largely simplifies the code. Its enhanced speed is an extra feature to

its readability and conciseness.

11 Linear models in Armadillo

One possibility is to start by creating a minimal package with the provided templates.

```
install.packages("armadillo")
# or
# remotes::install_github("pachadotdev/armadillo")
armadillo::create_package("armadilloexample")
```

Given a design matrix X and outcome vector y, one naive function (available in the package template) to obtain the Ordinary Least Squares (OLS) estimator $\hat{\beta} = (X^tX)^{-1}(X^tY)$ (Hansen 2022) as a matrix (column vector) is:

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

using namespace arma;
using namespace cpp11;

[[cpp11::register]]
doubles_matrix<> ols_mat_(const doubles_matrix<>& y,
                          const doubles_matrix<>& x) {
  Mat<double> Y = as_Mat(y);
  Mat<double> X = as_Mat(x);

  // \beta = (X^tX)^{-1}X^tY
  Mat<double> b = inv(X.t() * X) * X.t() * Y;

  return as_doubles_matrix(b);
}
```

The previous code loads the corresponding namespaces (e.g., the `using namespace arma`) in order to simplify the notation (e.g., using `Mat` instead of `arma::Mat`), and then it declares the function `ols_mat()` that takes inputs from R, does the computation on C++ side, and it can be called from R. In this particular case, because the output has dimension $n \times 1$, it

is possible to use `doubles ols_mat_` and `as_doubles(beta)` to return an R vector instead of a matrix.

Unlike the first Gauss-Jordan example, it uses the `inv()` function from Armadillo instead of implementing the inverse. It also uses `X.t()` to transpose and `*` to multiply matrices, which saves writing a loop to transpose and three loops to multiply. Armadillo uses its own definition of the multiplication operator, and when it is used with two matrices it does the same as the `%*%` operator in R.

The use of `const` and `&` are specific to the C++ language and allow to pass data from R to C++ by reference, that avoid copying the data, and therefore save time and memory.

`as_Mat()` and `as_doubles_matrix()` are 'armadillo' bridge functions to pass data between R and the Armadillo library.

In order to use this function in R, it needs to be documented, and after loading the package it is possible to compare with the R computation:

```
devtools::document()
devtools::load_all()

x <- armadillo::mtcars_mat$x
x <- x[, c("wt", "cyl4", "cyl6", "cyl8")]
y <- armadillo::mtcars_mat$y

ols_mat(y, x)
```

This can be verified against the R code to verify that the solution is

$$\hat{\beta} = (-3.21, 33.99, 29.74, 27.92).$$

```
solve(t(x) %*% x) %*% t(x) %*% y
```

In R, the `lm()` function does not use a code similar to the previous implementation. Instead, in R uses the QR decomposition to solve the OLS problem, which is more stable and efficient than the direct computation of the inverse of X^tX.

A more robust OLS implementation is:

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

using namespace arma;
using namespace cpp11;

[[cpp11::register]] doubles_matrix<> ols_mat_qr_(const doubles_matrix<>& y,
                                                 const doubles_matrix<>& x)
                                                 {
  Mat<double> Y = as_Mat(y);
  Mat<double> X = as_Mat(x);

  // (X'X)^(-1)
  Mat<double> Q, R;
  bool computable = qr_econ(Q, R, X.t() * X);

  if (!computable) {
    stop("QR decomposition failed");
  } else {
    // backsolve R
    Mat<double> b = solve(R, Q.t() * X.t() * Y);
    return as_doubles_matrix(b);
  }
}
```

The previous example, instead of directly inverting X^tX, creates empty matrices Q and R, and a boolean (e.g., logical) value for `qr_econ()`, and then the QR function tries to decompose X^tX into an orthogonal matrix Q and an upper triangular matrix R such that $XtX = QR$. If the composition is sucessful, the function returns "true" or "false" otherwise. The `solve()` arguments come from the fact that R is upper triangular, and backsolving from the last equation results in $R\beta = QX^ty$.

11.1 Logistic regression

Armadillo also provides additional data structures, such as field and cube, which ressemble a list of scalars, vectors or matrices and that are particularly useful for loops.

Adapting from the OLS examples, it is possible to fit a logistic regression with a loop that repeats calls to a function that returns the OLS coefficients. To do this, the starting point

is to transform the data by the logistic link function (McCullagh and Nelder 1989; Vargas Sepúlveda 2023):

$$\mu = \frac{y + 1/2}{2}$$
$$\eta = \log\left(\frac{\mu}{1-\mu}\right)$$
$$z = \eta + \frac{y - \mu}{\mu}$$

From the transformed outcome z and a design matrix X, we can implement a Re-Weighted Least Squares (RWLS) algorithm to obtain the coefficients of the logistic regression (McCullagh and Nelder 1989). The following code is a naive implementation of the RWLS algorithm:

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

using namespace arma;
using namespace cpp11;

Mat<double> rwls_mat_coef_(const Mat<double>& Y, const Mat<double>& X,
                           const Mat<double>& W) {
  Mat<double> Wd = diagmat(W);

  // \beta = (X^tWX)^{-1}X^tWZ
  Mat<double> B = inv(X.t() * Wd * X) * X.t() * Wd * Y;
  return B;
}

[[cpp11::register]] doubles_matrix<> logistic_mat_coef_(
    const doubles_matrix<>& y, const doubles_matrix<>& x) {
  // v = original variables y and x
  field<Mat<double>> v = {as_Mat(y), as_Mat(x)};

  // nv = new variables mu, eta, and z
  field<Mat<double>> nv(3);
  nv(0) = (v(0) + 0.5) / 2;
  nv(1) = log(nv(0) / (1 - nv(0)));
  nv(2) = nv(1) + (v(0) - nv(0)) / nv(0);
```

```
  // s = scalars dif, rss1, rss2, and tol
  // initialized with 1, 1, 1, and 0.05 as a starting point
  Col<double> s = {1, 1, 1, 0.05};

  // res = residuals without and with transformation
  field<Mat<double>> res(2);

  // b = regression coefficients
  Mat<double> b;

  while (abs(s(0)) > s(3)) {
    b = rwls_mat_coef_(nv(2), v(1), nv(0));
    res(0) = nv(2) - v(1) * b;
    nv(1) = nv(2) - res(0);
    nv(0) = exp(nv(1)) / (1 + exp(nv(1)));
    nv(2) = nv(1) + (v(0) - nv(0)) / nv(0);
    res(1) = v(0) - v(1) * b;
    s(2) = accu(res(1) % res(1));
    s(0) = s(2) - s(1);
    s(1) = s(2);
  }

  b = rwls_mat_coef_(nv(2), v(1), nv(0));
  return as_doubles_matrix(b);
}
```

12 Conclusion

'armadillo' 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.

13 Acknowledgements

We would like to thank Professor Salma Emara who taught us C++ in the course ECE244 (Programming Fundamentals). 'armadillo' is a byproduct of the knowledge we acquired in that course.

References

ref-burns2011rBurns, 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.

Emara, Salma. 2024. *Khufu: Object-Oriented Programming Using C++*. Self-published.

Hansen, Bruce. 2022. *Econometrics*. Princeton University Press.

McCullagh, P., and J. A. Nelder. 1989. *Generalized Linear Models*. 2nd ed. New York: Routledge. https://doi.org/10.1201/9780203753736.

Müller, Kirill, and Hadley Wickham. 2023. *Tibble: Simple Data Frames*. https://CRAN.R-project.org/package=tibble.

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.

Strang, Gilbert. 1988. *Linear Algebra and Its Applications*. 3rd ed. –. San Diego: Harcourt, Brace, Jovanovich, Publishers.

Vargas Sepúlveda, Mauricio. 2023. *The Hitchhiker's Guide to Linear Models*. Leanpub. https://leanpub.com/linear-models-guide.

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 Grolemund, et al. 2019. "Welcome to the Tidy-

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

Wickham, Hadley, Jeroen Ooms, and Kirill Müller. 2023. *RPostgres: C++ Interface to Postgresql*. https://CRAN.R-project.org/package=RPostgres.

Zhang, Xianyi, and Martin Kroeker. 2024. "OpenBLAS : An Optimized BLAS Library." https://www.openblas.net/.