Implementation of Gaussian elimination in C++

January 1, 2024

Šimon Brandner
brandsi1@cvut.cz
Faculty of Electrical Engineering,
Czech Technical University

Abstract

This semestral work demonstrates a C++ implementation of Gaussian elimination. We show how it can be used for various tasks like solving systems of linear equations, matrix equations, computing determinants, computing inverses etc. We also implement a parallelized version of this algorithm leveraging the concurrency features of C++ and show how it performs compared to the regular version. Lastly, we give an example of the algorithm's shortcomings.

1 Introduction

The goal of this semestral work was to implement Gaussian elimination using C++, implement a parallelized version and measure the algorithm's performance as well as accuracy. Along with Gaussian elimination several related algorithms such as determinant computation, matrix multiplication etc. were implemented to aid testing.

2 Gaussian elimination

Gaussian elimination is a well-known algorithm for solving systems of linear algebraic equations. It leverages the linearity of the equations to transform the original augmented matrix to a form which makes it "easy" to see the system's solution. This is done using a sequence of isomorphisms which leave the set of all solutions unchanged (1).

In practice, for square matrices, this means performing row elimination in order to get an upper triangular matrix. Gaussian elimination works for non-square matrices in theory but in real-world scenarios, we cannot easily determine if a given vector is a zero vector making it impossible to determine the rank of a matrix or to find its kernel (null space). For these reasons, we focus on regular matrices while keeping the code flexible enough to handle other cases in the future (1).

2.1 Implementation

Our implementation of Gaussian elimination is available on GitHub at SimonBrandner/gem-cpp along with instructions on how to run the code.

We store a matrix in an `std::vector<T> data` encapsulated by template class `Matrix<T>`, where `T` is the floating point number type to be used. `Matrix<T>` has several member functions allowing us to interact with the `data` safely and efficiently. This class also provides methods for generating several types of matrices (see Section 3) and computing inverses, determinants etc.

Most of the elimination logic lies in the class `EliminableMatrix<T>` which inherits from `Matrix<T>`.

The first notable member function is `void Matrix<T>::pivot(size_t column)` which picks a row with the best pivot – a pivot with the highest absolute value[1] – and moves it so that it is in the right place to get the upper triangular matrix.

The `void Matrix<T>::perform_gem()` method calls `void Matrix<T>::pivot(size_t column)` for each row/column and eliminates the rows below the row with the pivot using the row with the pivot. Holistically, it transforms the matrix into an upper triangular shape.

The `void Matrix<T>::perform_jem()` member function then transforms the Matrix into a diagonal shape. The `void Matrix::normalize_rows_based_on_diagonal(bool parallel)` method follows making sure we have an identity matrix. This makes the Jordan step complete.

These member functions are called by the template function `Matrix<T>`

[1]Picking a pivot with the highest absolute value improves the numerical stability of Gaussian elimination.

`solve_system_of_equations(Matrix<T> map, Matrix<T> right_side)` which also extracts the solution out of the augmented matrix.

2.2 Parallelization

While Gaussian elimination itself is not parallelizable, parts of it are. Namely, once a pivot is picked we can eliminate the rows in parallel. Similarly, the Jordan step of elimination may be parallelized. All of this can be done using the `std::thread` features of C++.

While this can drastically improve performance for large matrices it can harm performance for small matrices due to the overhead it introduces (see Section 3).

3 Findings

This section presents the results of running the Gaussian elimination algorithm on the Hilbert matrix H. which is a square matrix with elements

$$\boldsymbol{H}_{\text{ij}} = \frac{1}{i+j-1}.$$

We then choose a solution

$$\boldsymbol{x}_{\text{exact}} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

based on which we compute the right hand side

$$\boldsymbol{b} = \boldsymbol{H}\boldsymbol{x}_{\text{exact}}.$$

3.1 Time complexity

Firstly, we look at how fast each algorithm is for matrices of different size n.

3.1.1 System of linear equations

As we can see in Figure 1, the solver's performance matches the expected $O(n^3)$. The parallelized version performs quite a bit better, as we expected.

3.1.2 Determinant

As a side experiment, we also look at how our implementation of Gaussian implementation compares to computing determinants from the definition. Again, as expected we can see that computing determinants from the definition is extremely slow (actually $O(n!)$) which is easily outperformed by our systems solver. Since going through all 10! permutations of set $\{1, ..., 10\}$ would take a lot of time, we only show results up to a matrix with $n = 9$.

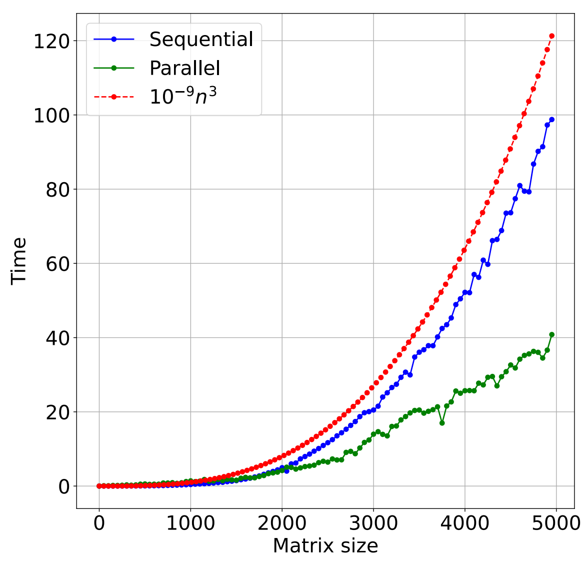

Figure 1: Time complexity of solving a system of linear equations

In this example, we can also see that parallelization carries quite a bit of overhead which results in worse performance when performing Gaussian elimination on small matrices.

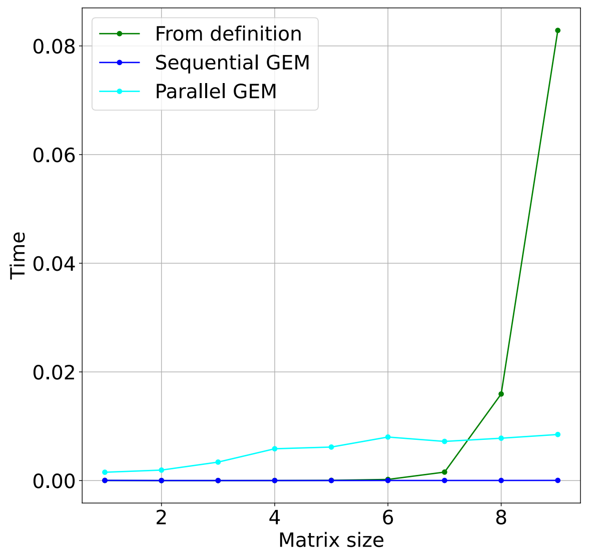

Figure 2: Time complexity of determinant computation

3.2 Error and residual

Aside from performance we also explored the accuracy of our Gaussian elimination implementation. This has been done using error e computed as follows:

$$e = \|\boldsymbol{x}_{\text{exact}} - \boldsymbol{x}_{\text{comp}}\|.$$

And using the residual r computed as follows:

$$r = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{\text{comp}}\|.$$

Where $\|\boldsymbol{x}\|$ denotes the Euclidean norm of the vector $\boldsymbol{x}$.

As is visible in Figure 3, the Hilbert matrix is a great example of how Gaussian elimination may perform very well in terms of residual but awfully in terms of error (2), (3).

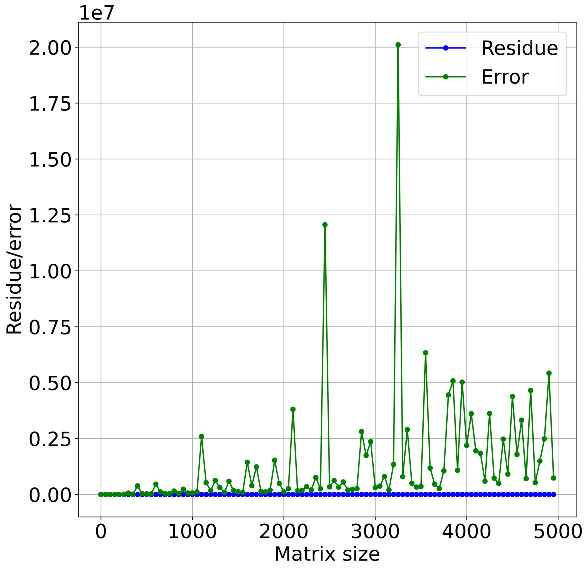

Figure 3: System stability

4 Further improvements

Several improvements could be made to the current implementation to improve its performance and functionality. We shall briefly explore these in this section.

The first possible improvement would be to avoid calling `T Matrix<T>::at(size_t row, size_t column)` many times when performing row elimination operations as each call effectively means having to perform an addition and a multiplication. That said, the compiler may be able to detect this and optimize it away.

Another plausible optimization is ignoring matrix elements whose value should be 0 during the Jordan step of row elimination.

As mentioned before (see Section 2), it is difficult to use Gaussian elimination for solving systems with non-regular matrices but there are tricks which can be used to do this. One example would be solving the following system

$$\left(\begin{array}{ccc|c} 1 & -4 & 7 & 10 \\ 0 & 1 & -2 & -4 \end{array}\right)$$

by adjusting it to the following form

$$\left(\begin{array}{ccc|cc} 1 & -4 & 7 & 10 & 0 \\ 0 & 1 & -2 & -4 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{array}\right).$$

We can now use the last column to compute the kernel of the matrix and therefore get the whole set of solutions. While this trick is very simple and elegant, it only works for underdetermined rectangular matrices whose rank we know beforehand. For other applications methods such as singular value decomposition have to be used.

5 Conclusion

The goal of this semestral work was to demonstrate an implementation of Gaussian elimination in the C++ programming language. We have shown that this can be done in a rather efficient manner. Our implementation has shown to perform as expected. We have also proven that while Gaussian elimination as a whole is not parallelizable, parts of it can be parallelized which has a great effect on performance.

We have shown that computing determinants using the definition is very impractical and is much better done using Gaussian elimination.

While the algorithm is effective, it has its shortcomings such as having high error values on Hilbert matrices of most sizes.

6 Bibliography

1. JIŘÍ VELEBIL. *Abstraktní a konkrétní lineární algebra.* Online. 2023. Available from: https://math.fel.cvut.cz/en/people/velebil/akla.html

2. J. STOER and R. BULIRSCH. *Introduction to Numerical Analysis.* Springer, 2002. ISBN 0-387-95452-X.

3. CLEVE MOLER. *Numerical Computing with MATLAB.* Online. 2004. Available from: https://www.mathworks.com/moler/chapters.html