FYS3150
Project 1 - Atom Heart Matrix

Hugounet, Antoine & Villeneuve, Ethel

September 2017
University of Oslo
`https://github.com/kryzar/Alcyonide.git`

Table of contents

A short glimpse to this project

This project uses the pretext of a differential equations to study the implementation of Linear Algebra algorithms with modern languages. It aims to make one realize that many of the scientific problems can be solved with simple Linear Algebra tricks which involve matrices, vectors, diagonalization, etc. However, if those algorithm remain rather simple for a human being, they require quite a lot of memory and power for a standard laptop.
Calculating the inverse of a matrix using its adjugate matrix and its determinant is a huge operation for sizes $n > 10^3$. The gaussian elimination appears to be less direct but way more efficient regarding the operation time. It's a progress but the LU decomposition is even better than the gaussian elimination, and its brute force approach does not work for matrices which have any zero diagonal term. It even produces a result (which is obviously wrong) for non-singular matrices ! Efficient but restrictive.

In the end, the results we obtain are approximations of exact mathematical results, due to loss of numerical precision and the fact that computers are built on discrete values. The problematic is then to evaluate carefully our objectives. Stability ? Precision ? Speed ? Memory ? It is the job of the programmer to find its own balance and to code in consequence. The first rule of computational science is that no program is fully reliable, no program fully is impartial.

Chapitre 1

Resolution of the equation Ax = w with the method of gaussian elimination

The files for this chapter are in the folder "General case" on GitHub. You will find the program in C++ and some data from the algorithm.

The gaussian elimination is an algorithm used for solving systems of linear equations [1]. This algorithm is composed of two parts : a forward substitution to make the n sized matrix an upper triangular matrix and a backward substitution to solve the equations.
The gaussian elimination is a good example of a easily computerizable algorithm, but which can be subtle to code. For large n, it is more efficient than the classical method of matrix inversion which needs a lot of calculations.

1.1 Transformation of A into a upper triangular matrix

Gaussian algorithm

1: **for** row $= \overline{1, n}$ **do**
2: ratio $\leftarrow A[\text{row}, \text{row} - 1]/A[\text{row} - 1, \text{row} - 1]$
3: **for** $i = \overline{\text{row}, n}$ **do**
4: $A[i, \text{row} - 1] \leftarrow 0$
5: **end for**
6: **for** col $= \overline{\text{row}, n}$ **do**
7: $A[\text{row}, \text{col}] \leftarrow A[\text{row}, \text{col}] - \text{ratio} * A[\text{row} - 1, \text{col}]$
8: **end for**
9: $w[\text{row}] \leftarrow w[\text{row}] - \text{ratio} * w[\text{row} - 1]$
10: **if then**$A[\text{row}, \text{row}] = 0$
11: ERROR. Division by 0 !
12: **end if**
13: **end for**

1. It is also called "row reduction".

Remarque 1.1. *We have directly changed to* 0 *the first element of each line and all the elements below. It spares the computer some useless calculations and avoids us numerical approximations which might not end to* 0.

The program begins with a set of tests to set aside the possible errors. This algorithm is quite efficient but is no match at all to Armadillo. Please build and run the main.cpp file to test the efficiency and limits of the algorithm [2]

1.2 Comparison with other methods

There are two other methods : the brute force approach of matrix inversion and the LU decomposition.

— The classical approach for the matrix inversion needs a lot of calculations and will not be efficient enough compared to the two others.

— The LU decomposition is also a process of gaussian elimination. It consist of separate the matrix A into two matrices : a lower triangular (L) and an upper triangular (U) matrix. The determinant is the sum of U's diagonal elements. We will use the LU function of *Armadillo* to compare the efficiency of the methods. In the following array, you will see the results regarding the execution time and relative error for our algorithm and the Armadillo function.

n	Operation time (s)			relative error
	Our algorithm	LU armadillo	difference between them	
2	$9 * 10^-6$	$8 * 10^-6$	$1 * 10^-6$	-0.903089987
3	$7 * 10^-6$	$1 * 10^-6$	$6 * 10^-6$	0.7781512504
10^5	5.68913	$2 * 10^-6$	5.689128	6.454015709

We clearly see here that the LU decomposition function of Armadillo is more efficient than our gaussian elimination algorithm. For large n, our algorithm is way less efficient than Armadillo.

Number of FLOPs	
General case	LU decomposition
$3n^3 + 2n^2 + 2n - 5$	$\frac{2}{3}n^3$

The gaussian elimination is still a well-balanced algorithm. It is easy to imagine and to code, its execution-time is fair, for not very very large n, it works for most of cases (see next section). It certainly do not have the best precision but it is balanced between the important criteria of programming.

2. We put the algorithm in an external cpp file to make it easier to export use with other problems.

It is the programmer's responsibility to determine precisely what is important for his program. In our case, the resolution of a set of linear equation in an easy and quick way, the criteria are satisfied. But we wouldn't use this algorithm to launch a rocket...

1.3 Limits of the gaussian elimination law

The gaussian elimination has many advantages. But the main drawback is that it works only for regular matrices. For a singular matrix, the algorithm returns a *wrong* result (see the example 3 in the results file).

Regular matrices are associated with bijective mapping. There has to be only one result for this equation. The error does not indeed come from the program but from the mathematical algorithm itself. Knowing this, we tried to cover as many cases as possible using the exit() function from C++, but it is fairly impossible to predict all possibilities in a relatively simple way.

To make the program work correctly, the diagonal terms of the matrix **A** must not be zeros. Indeed, each substitution needs a division by a different non-zero diagonal coefficient. However, we can fix that by swapping rows but it makes the algorithm more complicated. This goes against the method's peculiar simplicity. Moreover, it may use up memory unnecessarily because of the storage of some matrix rows. Even by doing that, the algorithm will not be perfect, the precision will not be improved. For those cases, we may think of a very optimized inverse or LU algorithm.

Chapitre 2

Special case : tridiagonal matrix

The files for this chapter are in the folder "Tridiagonal case" on GitHub. You will find the program in C++ and some data from the algorithm.

Here, we attempt to specialize the general algorithm for tridiagonal matrices. The reasoning itself will be the same, but the declaration of the matrix elements will be different. The goal is to solve the one-dimensional Poisson equation using the gaussian elimination.

2.1 The one-dimensional Poisson equation

We can write the Poisson equation as $-u''(x) = f(x)$, with $u''(x)$ the second derivative of $u(x)$ and u the electrostatic potential. The Poisson equation is mainly used in electromagnetism to describe potential field caused by a charge. This is a simple example (because of its simplification $-u''(x) = f(x)$) of a linear second-order differential equation as we often find them in physics.

2.1.1 Special use of Poisson equation

Before trying to solve this equation, let us be more specific on the conditions : we use here the Dirichlet boundary conditions.

$$-u''(x) = f(x), x \quad x \in (0,1), \quad u(0) = u(1) = 0$$

Then, we approximate u with discretized values v_i with grid points $x_i = ih, \quad i \in (0, n+1)$ and $x_0 = 0$ and $x_{n+1} = 1$. We define also the step length as $h = 1/(n+1)$, and the boundary conditions $v_0 = v_{n+1} = 0$.
We have

$$-u''(x) = f(x) \tag{2.1}$$

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = \overline{(1,n)} \tag{2.2}$$

2.1.2 Rewriting of the equation as a set of linear equations

$$-\frac{v_{i+1}+v_{i-1}-2v_i}{h^2} = f_i \quad \text{for } i = \overline{(1,n)} \tag{2.3}$$

$$\Rightarrow -(v_{i+1}+v_{i-1}-2v_i) = h^2 f_i \tag{2.4}$$

$$\Rightarrow -v_{i-1}+2v_i-v_{i+1} = h^2 f_i \tag{2.5}$$

So, if we express this equation in matrix form we have

$$\begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{i-1} \\ v_i \\ v_{i+1} \\ \dots \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} f_1 \\ f_2 \\ \dots \\ f_{i-1} \\ f_i \\ f_{i+1} \\ \dots \\ f_n \end{bmatrix}$$

So we can rewrite the equation as

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}},$$

with

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{bmatrix} \quad \text{and} \quad \tilde{\mathbf{b}} = h^2\mathbf{f}$$

Now we have the Poisson equation written as a set of linear equation. We can use the gaussian elimination to solve it.

2.2 Special algorithm

2.2.1 Specialization of the general algorithm

In the special case program, we did not use a matrix as we did for the general case but three dynamic arrays to set the three diagonals. The vector $\tilde{\mathbf{b}}$ corresponds to g in the program for more simplicity ; also be careful with the index i in the program : i in the program corresponds to $i+1$ in reality, that is because of arrays which must begin with $i = 0$. The process is quite the same as the general case for the gaussian elimination.

2.2.2 Solution to the Poisson equation

By running the program, we have values for the approximate formula of Poisson equation $-\frac{v_{i+1}+v_{i-1}-2v_i}{h^2} = f_i$. Let us compare these values to the closed-form solution of the equation

$u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. What we have done with that program is to approach the exact value. See below the comparative graph between what we should have and our results. On top of that, main.cpp requires arguments :

— argv[0] : name of the program
— argv[1] : size n
— argv[2] : coefficient of the first diagonal
— argv[3] : coefficient of the second diagonal
— argv[4] : coefficient of the third diagonal
— argv[5] : path of the file for the results
— argv[6] : path of the file for the data

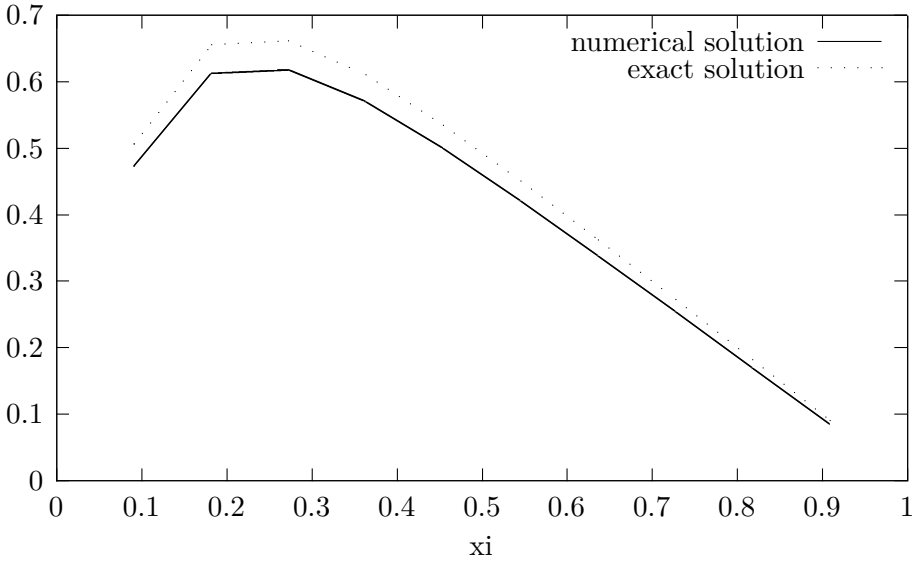

FIGURE 2.1 – Comparative graph between numerical and exact results for $n = 10$

We can see that for $n \geq 100$ (and even probably before that) the expected and the numerical results are similar.
The smaller the step length is the closer we are to the real plot and the more we have values the more we can be precise but we are still confronted with the numerical lack of precision for large values. (see examples in the results file)

2.3 Comparisons

The relative error is quite log-linear for log(n) until $n = 10^5$. In theory, if n tends to ∞, our precision gets to be perfect. However, since computers can only work with a finite number of numbers, the precision directly depends on the way we store numbers in the computer. For the float type, the computer uses 32 bits, for the double type, it uses 64 bits. This leads to a precision of 10^{-7}, which implies that the computer considered all numbers $x \leq 10^{-7}$ to be 0. This is called the loss of numerical precision and it has been a problem here for the relative

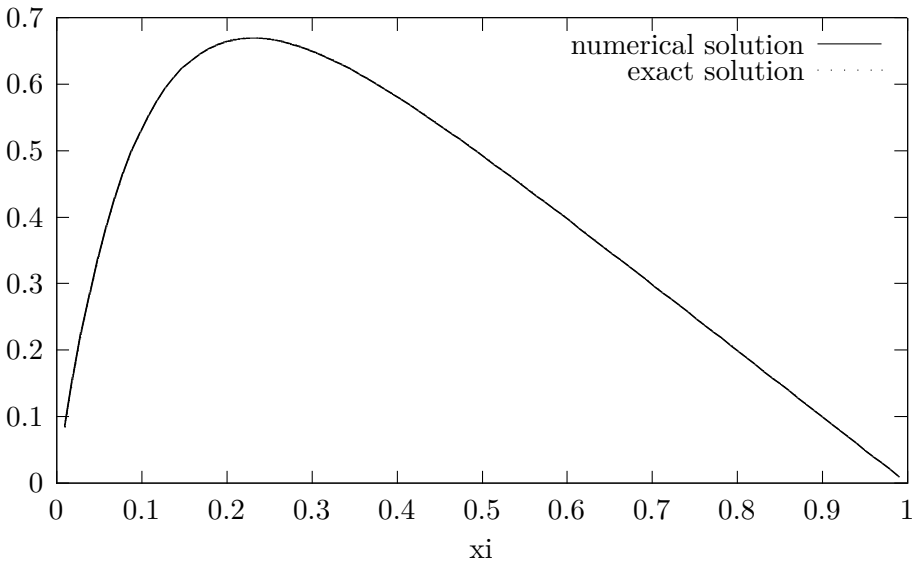

FIGURE 2.2 – Comparative graph between numerical and exact results for $n = 100$

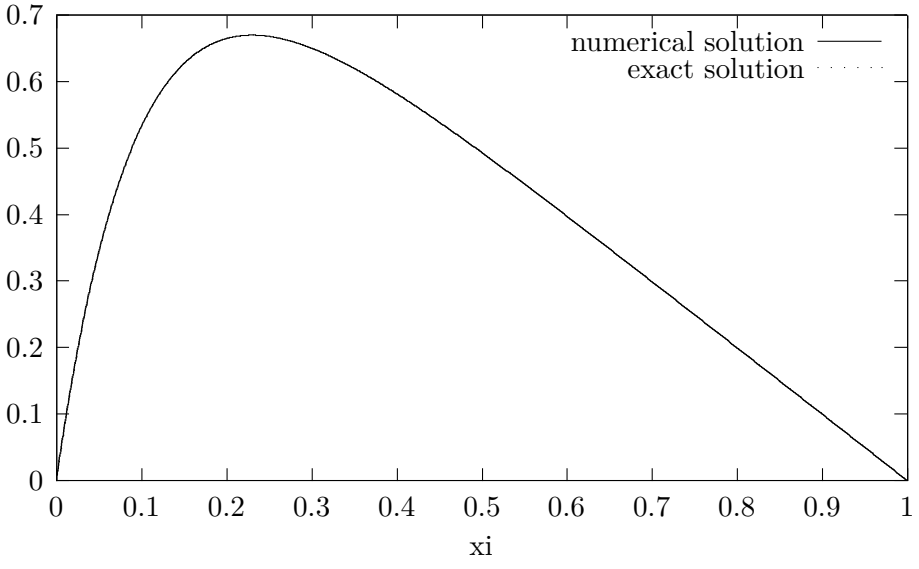

FIGURE 2.3 – Comparative graph between numerical and exact results for $n = 1000$

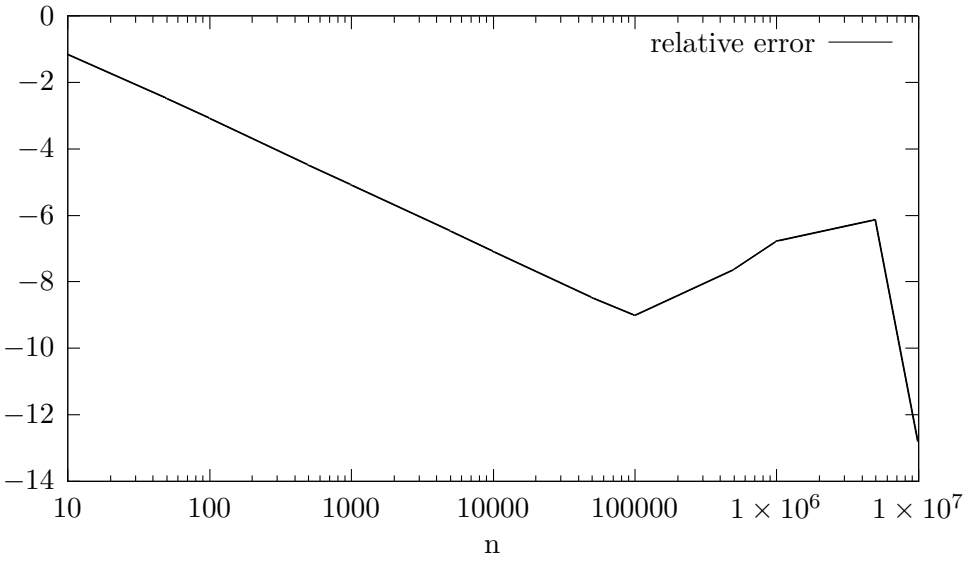

FIGURE 2.4 – Relative error

errors. In many cases, we asked the computer to compare two values, and the results may not be reliable [1].

Number of FLOPs	
General case	Tridiagonal case
$3n^3 + 2n^2 + 2n - 5$	$11n - 6$

1. See the data files.

Conclusion

This project may be small but it gathers many of the problems a programer has to deal with every day : loss of numerical precision, simplicity of the algorithm, operation time, the ability to use the same code for future projects. We have developed this c++ algorithm to be a good balance of all the criteria. We know that it is not the most precise or the quickest but it would be enough for most of the easy scientific problems. However we already realize that loss of numerical precision will be a crucial issue for other problems.