FYS4150 - Project 1

Kevin Cappa, Peder Hauge & William Eivik Olsen*
University of Oslo - Department of Physics

(Dated: September 10, 2020)

The main scope of this project is to look at a particular class of differential equations and find a numerical solution to these via the implementation of an algorithm developed in the C++ environment, as well as evaluating its effectiveness through error analysis. In addition to getting acquainted with dynamic memory allocation and array handling in this programming language, an alternative solution approach by using the Armadillo library is considered.

I. INTRODUCTION

The equation in consideration is a linear second order nonhomogeneous differential equation of following form:

$$\frac{\partial^2 y}{\partial x^2} + k^2(x)y = f(x), \tag{1}$$

where $f(x) \neq 0$ is the inhomogeneous term, or source term, and $k^2(x)$ is a real function. The principal focus will be on solving a particular case of the given equation known as the one dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations.

In the first part of this report the theoretical foundations of the project will be explained, in particular how the proposed algorithm will discretize the solution of the considered equation. Firstly a general function will be defined describing a generalized Thomas algorithm developed by taking into account the matrix form of the linear equation set. Given a source term $f(x)$, the obtained solution will be repeatedly plotted using Python with an increasing number of grid points in the interval given by the boundary conditions in order too look at the behaviour of the approximation as the step length between the plotted points decreases.

Thereafter this algorithm will be specialized to solve a particular case of a tridiagonal matrix approximation having identical elements immediately above and below the main diagonal. Then a brief analysis of the floating point operations (FLOPS) necessary for the execution of the main program, comparing the CPU time for the two different algorithms for matrices with up to $n = 10^7$ grid points will be presented. Further implementation of the program include a computation of the relative error setting up a given logarithmic function.

At last an alternative approach will be explored by using the Armadillo library and considering a different kind of matrix decomposition known as LU decomposition.

* Code repository: https://github.com/williameivikolsen/FYS4150

II. THEORY

A. Discretization of the problem

First let's look at some mathematical background on function discretization. Given a function $u(x)$ it can be discretized by letting the continuous independent variable x go to a discrete x_i defined in the following way:

$$x \to x_i + ih \tag{2}$$

where $i = 1, ..., n$ and h is a step size equivalent to: $h = (x_n - x_0)/n$. As a consequence of the following domain interval and boundary conditions

$$x \in (0, 1),\ u(0) = u(1) = 0. \tag{3}$$

we can identify $x_0 = 0$ and $x_{n+1} = 1$. Accordingly the continuous function $u(x)$ is now approximated by the discrete function $v(x_i) = u(x_i)$ and we can state the following identities: $v(x_i) = v_i$ and $v(x_i \pm h) = v_{i\pm1}$.

Now examining for instance at the three-dimensional Poisson equation used in electrodynamics

$$\boldsymbol{\nabla}^2\Phi = -4\pi\rho(\mathbf{r}) \tag{4}$$

in a few easy steps, assuming a spherically symmetric Φ and $\rho(r)$ and using the substitutions $\Phi \to \phi(r)/r$, $-4\pi\rho(r) \to f(x)$, $r \to x$ and $\phi \to u$, we can quite immediately deduce its one-dimensional variant

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\Phi}{dr}\right) = -4\pi\rho(r), \tag{5}$$

$$\frac{d^2\phi}{dr^2} = -4\pi r\rho(r). \tag{6}$$

$$-u''(x) = f(x) \tag{7}$$

Applying our discretization algorithm to this last equation Eq. (7) yields the following expression:

$$\left.\frac{\partial^2 v}{\partial x^2}\right|_{x_i} = \frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} + \mathcal{O}(h^2) \tag{8}$$

The error term $\mathcal{O}(h^2)$ will be analyzed in a later section and is therefore negligible in the implementation of our algorithm. Now substituting Eq. (8) into Eq. (7) and multiplying both sides by h^2 one gets:

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i, \tag{9}$$

where $f_i = f(x_i)$. For the running values of i this expression can be rewritten in vector form in the following manner:

$$\begin{bmatrix} 2v_1 - v_2 \\ -v_1 + 2v_2 - v_3 \\ -v_2 + 2v_3 - v_4 \\ \vdots \\ -v_n - 1 + 2v_n \end{bmatrix} = h^2 \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{bmatrix} \tag{10}$$

Which, by defining $\tilde{b}_i = h^2 f_i$, is equivalent to the matrix form giving rise to the set of linear equations:

$$\begin{bmatrix} 2 & -1 & 0 & & \dots & 0 \\ -1 & 2 & -1 & 0 & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & \\ & 0 & -1 & 2 & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n+1} \end{bmatrix}$$

$$\implies \mathbf{Av} = \mathbf{b}, \tag{11}$$

Where A is a $n \times n$ matrix defined as follows:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \ddots & \vdots \\ 0 & -1 & 2 & -1 & \ddots & 0 \\ 0 & 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & 0 & 0 & -1 & 2 \end{bmatrix} \tag{12}$$

For a more general purpose aimed at the first part of the code implementation this matrix **A** will be generalized to the subsequent form:

$$\mathbf{A} = \begin{bmatrix} d_0 & c_0 & 0 & & \dots & 0 \\ a_0 & d_1 & c_1 & \ddots & & \vdots \\ 0 & a_1 & d_2 & \ddots & \ddots & \\ & \ddots & \ddots & \ddots & c_{n-3} & 0 \\ \vdots & & \ddots & a_{n-3} & d_{n-2} & c_{n-2} \\ 0 & \dots & & 0 & a_{n-2} & d_{n-1} \end{bmatrix} \tag{13}$$

B. Analytical solution

In order to solve equation Eq. (7) and to plot its graph, a source term $f(x) = 100e^{-10x}$ is given. Once this function is known, finding the solution for u is quite straightforward:

$$\begin{aligned} u''(x) &= -100e^{-10x} \\ \int \frac{du'(x)}{dx}dx &= \int -100e^{-10x}dx \\ u'(x) &= 10e^{-10x} + C \\ \int \frac{du(x)}{dx}dx &= \int \left(10e^{-10x} + C\right)dx \\ u(x) &= -e^{-10x} + Cx + D \end{aligned} \tag{14}$$

Taking into account the Dirichlet boundary conditions Eq. (3) and inserting them into the above expression one arrives at values for the integration constants:

$$\begin{aligned} u(0) &= -e^{-10\times 0} + C \times 0 + D = 0 \\ &\implies D = 1 \\ u(1) &= -e^{-10\times 1} + C \times 1 + 1 = 0 \\ &\implies C = e^{-10} - 1 \end{aligned}$$

The closed-form solution is thus obtained:

$$u(x) = -e^{-10x} + (e^{-10} - 1)x + 1 \tag{15}$$

C. Gaussian elimination

This set of linear equation shown in Eq. (11) solved by Gaussian elimination. There are two stages to this method, forward substitution and backward substitution. Looking first at general case of Eq. (13), in the forward substitution bit the following algorithm is used to reduce the augmented matrix $(A|\mathbf{b})$ to an augmented upper triangular matrix:

$$row(i) = row(i) - \frac{a_{i-1}}{d_i} \times row(i-1) \tag{16}$$

The new matrix elements are relabelled as $d_i \to \tilde{d}_i$ and $b_i \to \tilde{b}_i$:

$$\tilde{d}_i = d_i - \frac{a_{i-1} \times c_{i-1}}{\tilde{d}_{i-1}}; \tag{17}$$

$$\tilde{b}_i = b_i - \frac{a_{i-1} \times \tilde{b}_{i-1}}{\tilde{d}_{i-1}}; \tag{18}$$

$$\left(\begin{array}{ccccccc|c} \tilde{d}_1 & c_1 & 0 & & \dots & 0 & \tilde{b}_1 \\ 0 & \tilde{d}_2 & c_2 & \ddots & & \vdots & \tilde{b}_2 \\ 0 & 0 & \tilde{d}_3 & \ddots & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & c_{n-2} & 0 & \tilde{b}_{n-2} \\ \vdots & & \ddots & 0 & \tilde{d}_{n-1} & c_{n-1} & \tilde{b}_{n-1} \\ 0 & \dots & & 0 & 0 & \tilde{d}_n & \tilde{b}_n \end{array}\right) \tag{19}$$

From this form one can find the solutions for the vector components v_i by backward substitutions:

$$v_i = \frac{\tilde{b}_{i-1} - c_{i-1} \times v_{i+1}}{\tilde{d}_{i-1}} \tag{20}$$

where the index i runs backwards from $n-1$ to 1. By following this procedure the approximate solutions for the discretized one-dimensional Poisson Equation Eq. (7) are found.

Now some few precautions: In the C++ programming language the index of an array starts from 0, aka the first element of an n-dimensional array is labelled v_0 and the last one v_{n-1}, so in the case of our algorithm for the sake graph-plotting, we let the integer index i run from $i = 1, ..., n$, assigning the boundary values to the vector elements v_0, v_{n+1}, x_0 and x_{n+1}, since the vectors $vandx$ have length $n+2$.

$$\tilde{d}_0 = d_0 \wedge \tilde{b}_0 = b_0; \tag{21}$$

$$v_{n+1} = v_0 = 0; \tag{22}$$

$$v_n = \frac{\tilde{b}_{n-1}}{\tilde{d}_{n-1}}; \tag{23}$$

In the special case Eq. (12), since the elements of the matrix are known and $a_i = c_i$, this algorithm simplifies considerably as shown below:

$$\tilde{d}_i = \frac{i+1}{i}; \tag{24}$$

$$\tilde{b}_i = b_i + \frac{(i-1) \times \tilde{b}_{i-1}}{i}; \tag{25}$$

$$v_{i-1} = \frac{i-1}{i \times (\tilde{b}_{i-1} + v_i)} \tag{26}$$

This will also reduce the number of floating point operations (FLOPs) from $9n$ to $4n$, which decreases the time needed to run the special case of the main program in comparison to the general one.

D. LU-decomposition

Instead of solving a set of linear equations by Gaussian elimination directly, one can instead use LU-decomposition to make the process of finding the unknown variables easier. From the lecture notes [1] we know that a non-singular square matrix A can be written as a product of a lower triangular matrix L (with ones on the diagonal) and an upper triangular matrix U. This decomposition is unique. In the case where $A \in \mathbb{R}^{4\times4}$, the following equation shows how an LU-decomposition would look.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

The matrix equation

$$A\mathbf{v} = \mathbf{b} \tag{27}$$

then becomes

$$A\mathbf{v} = LU\mathbf{v} = L\mathbf{w} = \mathbf{b}, \tag{28}$$

where we have set

$$U\mathbf{v} = \mathbf{w}. \tag{29}$$

Solving $L\mathbf{w} = \mathbf{b}$ from Eq.(28) is easy. After the LU-decomposition, both L and $\mathbf{b}$ are know quantities, and since U is upper triangular, we can find $\mathbf{w}$ just by backward substitution.

After finding $\mathbf{w}$ as an intermediate step, we use this vector and solve Eq.(29) for $\mathbf{v}$. Because L is lower triangular, this step is done by forward substitution.

Later, when discussing the number of number of FLOPS needed to solve for $\mathbf{v}$, we'll see why it is advantageous to have an LU-decomposition of A. However, we can already now see that even with a different right hand side $\mathbf{b}$ in Eq.(27), the LU-decomposition of A will stay the same.

III. METHOD

In order to write this report, we developed program code that have the functionality listed below:

- Solve the matrix equation $A\mathbf{v} = \mathbf{b}$, where A is an arbitrary tridiagonal matrix of dimension n using the Thomas algorithm.
- Solve the above matrix equation, but now using the specific tridiagonal matrix A from Eq.(12).
- The same as the above point, but now using the C++ linear algebra package Armadillo to LU-decompose the matrix and solve the matrix equation.

- Compare the relative error between the analytical solution $u(x)$ in Eq.(15) with the computed solutions in all three cases.

- Compare the logged CPU time during backwards and forwards substitution in all three cases.

IV. RESULTS

When the matrix A has $n = 10$ elements along both sides, we can see from Fig. 1 that all of the numerical approximations give roughly the same value. By inspection, the calculated values of v_i differ quite a bit from the analytical solution of $u(x)$.

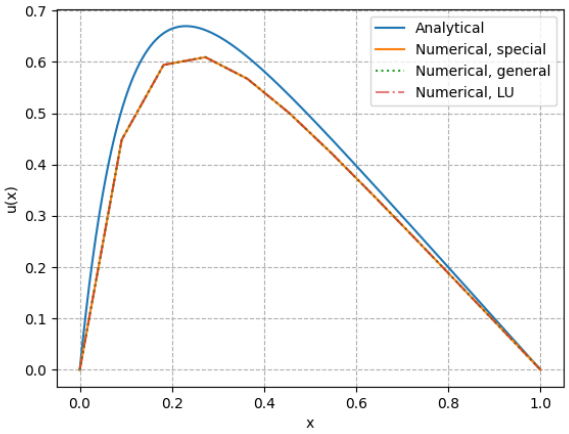

Figure 1. Comparison between numerical and analytic solutions for $n = 10$.

In Fig. 2 we have increased the matrix dimensions to $n = 100$, and the discrepancy is no longer visual. For higher values of n, $n = \{10^3, 10^4, 10^5, 10^6, 10^7\}$, a better way of judging the accuracy of the solutions is to find their maximum relative error, given by

$$\varepsilon_{\text{rel}} = \frac{u(x_i) - v_i}{u(x_i)}.$$

The maximum relative error for different values of h (that corresponding to increasing n) are shown in Fig. 3. For the LU-decomposition there are no data points for h lower than 10^{-4} due to memory constraints.

We have also measured the CPU time spent during the backward and forward substitution for the three cases. A logarithmic plot of these times for a single run (one sample per matrix size and method) is shown in Fig. 4. Note that the LU-decomposition itself is not included in the measured time.

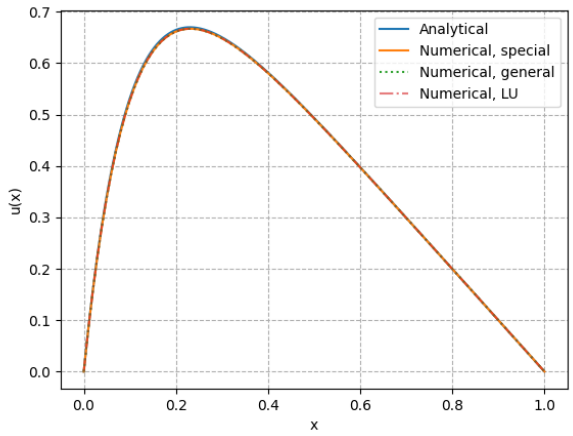

Figure 2. Comparison between numerical and analytic solutions for $n = 100$.

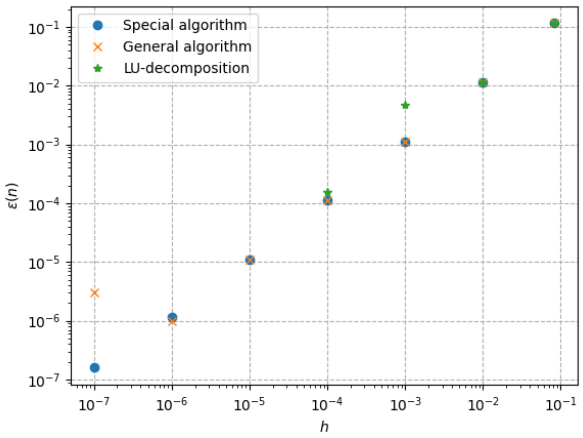

Figure 3. Comparison of $\max\{|\varepsilon_{\text{rel}}|\}$ for the different numerical methods.

V. DISCUSSION

In Fig. 3 we see that as we decrease h (increase n), the maximum relative error decreases proportionally. In Appendix A we argue that an optimal step-size h that minimizes the relative error is in the range between 10^{-4} to 10^{-5}. In theory, lower values for h than this will increase the relative error due to round-off errors. However, for the algorithms we've made that utilize Gaussian elimination, this loss of precision can't be seen before h has decreased to 10^{-6}. For the special algorithm, the precision seems to further improve as h reaches 10^{-7}. We have no compelling explanation for this, but it would certainly be easier to spot when the loss of precision occurs if we had run the program for more values of n.

In the case of the LU-decomposition, Fig. 3 also shows that the code stops to run properly when $n \geq 10^5$. This

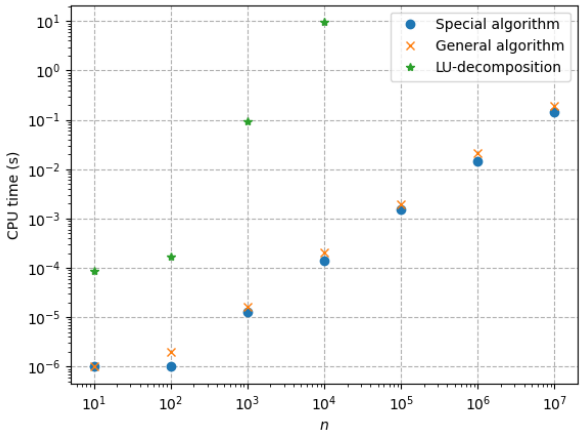

Figure 4. CPU time for the different methods. See text for details.

is because the space the matrices take up in memory exceeds what's available. Double precision floating point numbers uses 8 bytes (64 bits), so storing a matrix with dimension $10^4 \times 10^4$ will use around $8 \cdot (10^4)^2$ bytes or 0.8 GB. Increasing n to 10^5 increases this number to 80 GB. When a program is running the declared variables are stored in RAM, which for a modern laptop is between 8-16 GB, far below what's needed for a $10^5 \times 10^5$ matrix. In addition, the LU decomposition requires at least three times the memory of one matrix, because one needs to define the upper and lower triangular matrices as well. We manage to circumvent this problem by only storing the numbers in the diagonals of the matrix A, so that we don't fill up the memory with zeros.

By looking at the slope of the elapsed CPU times in Fig. 4, we see that the CPU time follows the number of FLOPS needed for each algorithm. The lecture notes [1] show us that the FLOPS needed for the Thomas algorithm goes as $\mathcal{O}(n)$, and as $\mathcal{O}(n^2)$ when having a LU-decomposed matrix. This matches very well with the straight lines we observe in the logarithmic plot, except for with the smallest matrix. This may have to do with the granularity of the timer; very small time intervals are hard to measure accurately. In order to improve the results we could have measured the time more than once for every combination matrix size-algorithm, and averaged over the measured times in order to reduce the statistical uncertainty.

VI. CONCLUSION

Drawing some conclusions: we have analyzed how to approach a mathematical problem and solve it with a numerical approximation, exploring the perks and downsides of the tridiagonal matrix algorithm (Gaussian elimination and Thomas algorithm) and LU-decomposition, finding that the former is highly favorable taking into account CPU times and memory allocation. We have also gotten acquainted git as a version control software and with C++ as a programming language.

Appendix A: Error analysis

When computing the second derivative of $v(x)$ as shown in Eq.(8), the mathematical error in the computation $\varepsilon_{\text{math}}$ will decrease as we make h smaller. On the other hand, when h is small enough, the difference between the points v_i and $v_{i\pm h}$ will not be computed correctly, as the difference between the values becomes the machine precision ε_M. In this section we will try to derive an expression for a value of h that gives the optimal trade-off between the mathematical error and round-off error.

From the lecture notes [1] we have that the three-point formula is

$$\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = v_i'' + 2\sum_{j=1}^{\infty} \frac{u_i^{(2j+2)}}{(2j+2)!} h^{2j},$$

where the latter sum are the error terms. If we now assume that the first term is dominant, we see that

$$\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} \simeq v_i'' + \frac{v_i^{(4)}}{12} h^2 = v_i'' + \mathcal{O}(h^2).$$

Note that $v_i^{(4)}$ refers to a 4th-derivative, not an exponent. We can now let

$$\varepsilon_{\text{math}} = \frac{v_i^{(4)}}{12} h^2.$$

We can now study the round-off error ε_{RO}. The floating point number representation of an arbitrary number a is

$$fl(a) = a(1 \pm \varepsilon_M),$$

where where the machine precision is $|\varepsilon_M| \leq 10^{-15}$ for double precision numbers. When h is small, the three-point formula gives

$$\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = \frac{(v_{i+1} - v_i) + (v_{i-1} - v_i)}{h^2} = \frac{2\varepsilon_M}{h^2},$$

so we set the round-off error to be

$$\varepsilon_{\text{RO}} = \frac{2\varepsilon_M}{h^2}.$$

This gives the model a total error

$$\varepsilon_{\text{model}} = \varepsilon_{\text{math}} + \varepsilon_{\text{RO}} = \frac{v_i^{(4)}}{12}h^2 + \frac{2\varepsilon_M}{h^2}.$$

Finding a h that minimizes $\varepsilon_{\text{model}}$ means finding a solution to

$$\frac{\mathrm{d}\varepsilon_{\text{model}}}{\mathrm{d}h} = 0,$$

and after calculating the derivative we find that

$$h_{\text{min}} \leq \left(\frac{24\varepsilon_M}{v_i^{(4)}}\right)^{1/4} \tag{A1}$$

The function $v(x_i)$ is the discretized version of $u(x)$ given in Eq.(14). Since

$$u''(x) = -100e^{-10x},$$

then

$$u^{(4)}(x) = -10^5 e^{-10x},$$

and

$$v_i^{(4)} = -10^5 e^{-10x_i}.$$

We want to find an upper bound for h_{min} by minimizing $|v_i^{(4)}|$, which happens for $x_i = 1$ when $x_i \in [0, 1]$. Using that

$$\min |v_i^{(4)}| = 10^5 e^{-10}$$

and inserting $|\varepsilon_M| = 10^{-15}$ into Eq.(A1), we find that

$$h_{\text{min}} \approx 2.7 \cdot 10^{-4}. \tag{A2}$$

This means that we will start seeing loss of precision for the situations where $h = 10^{-5}$ and smaller. In terms of the matrix dimension n, this is when $n \geq 10^5$.

[1] Morten Hjort-Jensen, Computational Physics: Lecture notes 2015