FYS3150 - Project 1

Frida Larsen

This report investigates the difference between the Thomas algorithm, a more specialized version of the Thomas algorithm and the method of LU-decomposition for solving a linear matrix equation involving a tridiagonal matrix. Although the specialized version is the fastest, both it and the general Thomas algorithm are only applicable to tridiagonal matrices. For very small matrices, all three algorithms were subject to large timing errors due to the granularity of the CPU clock. The LU-decomposition method was the slowest and became useless for large grid sizes due to memory consumption and a high number of floating point operations.

I. INTRODUCTION

There are many ways to approach a given problem, and many possible ways for reaching a solution. The aim of this project is to investigate and compare three different methods of solving the same problem, primarily in terms of numerical accuracy.

Three algorithms will be employed to solve a matrix equation, the Thomas algorithm, a specialized version of the Thomas algorithm and the method of LU-decomposition. The equation in question is the one-dimensional Poisson equation, which is given by

$$-\frac{\mathrm{d}^2}{\mathrm{d}x^2}u(x) = f(x), \tag{1}$$

with Dirichlet boundary conditions

$$u(0) = u(1) = 0 \tag{2}$$

on the interval $x \in (0, 1)$.

We will start by discretizing and rewriting this equation as a matrix equation, before moving on to develop the algorithms for solving it.

All relevant code may be found in the GitHub repository 'FYS3150-Computational-Physics'[1] under the Project1 folder. This folder also includes a Figures folder, which holds all the figures presented in this text as well as some additional figures.

II. THEORY

The second derivative of a general discretized function g_i can be approximated by

$$g_i'' \approx \frac{g_{i+1} + g_{i-1} - 2g_i}{h^2} \tag{3}$$

The discretized version of the Poisson equation (1) then becomes

$$-\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = f_i, \tag{4}$$

where we have defined the discretized approximation to u and x as u_i and x_i respectively, such that $x_i = ih$, $x_0 = 0$ and $x_{n+1} = 1$. We also have $f_i = f(x_i)$. The step length h is defined as

$$h = \frac{1}{n+1}, \tag{5}$$

where n is the total number of grid points on the interval (0,1). The discretized Dirichlet boundary conditions (2) are

$$u_0 = u_{n+1} = 0. \tag{6}$$

The discrete Poisson equation can be further simplified to

$$-u_{i+1} - ui - 1 + 2u_i = h^2 f_i.$$

By introducing $\tilde{d}_i = h^2 f_i$ and remembering that $u_0 = 0$ the Poisson equation for the first few i-values are

$$\begin{aligned} i = 1: &\quad 2u_1 - u_2 = \tilde{d}_1 \\ i = 2: &\quad -u_1 + 2u_2 - u_3 = \tilde{d}_2 \\ i = 3: &\quad -u_2 + 2u_3 - u_4 = \tilde{d}_3 \\ &\dots \end{aligned}$$

This set of equations can be rewritten as a matrix equation

$$\mathbf{Au} = \tilde{\mathbf{d}} \tag{7}$$

with

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{pmatrix}. \tag{8}$$

[1] https://github.com/fridalarsen/FYS3150-Computational-Physics

A. LU-decomposition

Any square matrix may be factorized by LU-decomposition such that

$$\mathbf{PM}_{(n\times n)} = \mathbf{LU}, \tag{9}$$

where $\mathbf{L}$ and $\mathbf{U}$ are lower and upper triangular matrices respectively. The matrix $\mathbf{P}$ is a permutation matrix, which describes the row permutations required on $\mathbf{M}$ for the LU-decomposition to be backwards compatible.

In order to solve a linear matrix equation (like equation 7) one can use the LU-decomposition of the matrix by rewriting the equation as

$$\mathbf{LUu} = \mathbf{P}\tilde{\mathbf{d}}, \tag{10}$$

given $\mathbf{PA} = \mathbf{LU}$. There are then two steps to solving the equation; First solve $\mathbf{Ly} = \mathbf{P}\tilde{\mathbf{d}}$ for $\mathbf{y}$ and then finally $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$.

III. METHOD

A. The Thomas Algorithm

The first algorithm we will be employing is known as the Thomas algorithm[1] or the tridiagonal matrix algorithm. The algorithm is a two-step process involving a forward substitution and a backward substitution, and aims to solve equation 7 where $\mathbf{A}$ is the general tridiagonal $n \times n$ matrix

$$\tilde{\mathbf{A}} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & 0 & \dots & \dots \\ 0 & a_2 & b_3 & c_3 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & 0 & a_{n-2} & b_{n-1} & c_{n-1} \\ \dots & \dots & \dots & 0 & a_{n-1} & b_n \end{pmatrix} \tag{11}$$

The first step is given by

$$c'_i = \begin{cases} \frac{c_i}{b_i} & ; \quad i = 1 \\ \frac{c_i}{b_i - a_i c'_{i-1}} & ; \quad i = 2, 3, \dots, n-1 \end{cases} \tag{12}$$

and

$$\tilde{d}'_i = \begin{cases} \frac{\tilde{d}_i}{b_i} & ; \quad i = 1 \\ \frac{\tilde{d}_i - a_i \tilde{d}'_{i-1}}{b_i - a_i c'_{i-1}} & ; \quad i = 2, 3, \dots, n \end{cases} \tag{13}$$

This requires 2 floating point operations (FLOPs) for $i = 1$, $3(n-1)$ FLOPs for the remaining c'_i's and $5n$ FLOPs for the remaining $\tilde{d}'_i$'s.

The second step is given by

$$u_n = \tilde{d}'_n; \quad i = n \tag{14}$$

$$u_i = \tilde{d}'_i - c'_i u_{i+1}; \quad i = n-1, n-2, \dots, 1. \tag{15}$$

This substitution requires 1 FLOP for $i = n$ and $2(n-1)$ FLOPs for the remaining computations. The total number of floating point operations for the Thomas algorithm is then

$$\text{FLOPs} = (10n - 2) \propto \mathcal{O}(n). \tag{16}$$

B. The Specialized Thomas Algorithm

Although the Thomas algorithm is specialized for tridiagonal matrices, it does not take into account the equal elements along the diagonals of $\mathbf{A}$ (8), nor the fact that the upper and lower diagonals are equivalent. A more accurate generalization of the original matrix would be

$$\mathbf{A}_g = \begin{pmatrix} b_1 & a_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & a_2 & 0 & \dots & 0 \\ 0 & a_3 & b_3 & a_3 & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & a_{n-1} \\ \dots & \dots & \dots & \dots & a_n & b_n \end{pmatrix}, \tag{17}$$

where all $a_i = a$ and $b_i = b$. We observe that

$$\mathbf{A}_g \underset{\sim}{II - \frac{a_2}{b_2} I} \begin{pmatrix} b_1 & a_1 & 0 & 0 & \dots & 0 \\ 0 & \tilde{b}_2 & a_2 & 0 & \dots & 0 \\ 0 & a_3 & b_3 & a_3 & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & a_{n-1} \\ \dots & \dots & \dots & \dots & a_n & b_n \end{pmatrix},$$

with $\tilde{b}_2 = b_2 - \frac{a_2 a_1}{b_1}$. Repeating this process for all rows 1 through n removes the lower diagonal and replaces the diagonal elements by

$$\tilde{b}_i = \begin{cases} b_1, & i = 1 \\ b_i - \frac{a_i a_{i-1}}{\tilde{b}_{i-1}}, & i = 2, 3, \dots, n \end{cases} \tag{18}$$

Applying this process to the right hand side of equation 7 results in

$$\tilde{d}'_i = \begin{cases} \tilde{d}_1, & i = 1 \\ \tilde{d}_i - \frac{a_i \tilde{d}'_{i-1}}{\tilde{d}'_{i-1}}, & i = 2, 3, \dots, n \end{cases} \tag{19}$$

Backward-substituting these expressions into equation 7 yields

$$u_i = \begin{cases} \frac{\tilde{d}'_i}{\tilde{b}_i}, & i = n \\ \frac{\tilde{d}'_i - a_i u_{i+1}}{\tilde{b}_i}, & i = n-1, n-2, \dots, 1 \end{cases} \tag{20}$$

Using the fact that our original matrix (8) has $a = -1$ and $b = 2$, we get

$$\begin{aligned} \tilde{b}_i &= \begin{cases} 2, & i = 1 \\ 2 - \frac{1}{\tilde{b}_{i-1}}, & i = 1, 2, \dots, n \end{cases} \\ &= \frac{i+1}{i}, \quad i = 1, 2, \dots, n, \end{aligned} \tag{21}$$

which leads to

$$\tilde{d}'_i = \begin{cases} \tilde{d}_1, & i = 1 \\ \tilde{d}_i + \frac{\tilde{d}'_{i-1}}{i/(i-1)}, & i = 2, 3, \ldots, n \end{cases}$$
$$= \tilde{d}_i + \frac{(i-1)\tilde{d}'_{i-1}}{i}, \quad i = 1, 2, \ldots, n. \tag{22}$$

This forward substitution requires $2n$ and $4n$ FLOPs for calculating the $\tilde{b}_i$'s and $\tilde{d}'_i$'s respectively. The final solution is then ultimately found by backwards substitution,

$$u_{i-1} = \frac{i-1}{i}(\tilde{d}'_{i-1} + u_i), \; i = n-1, \ldots, 2 \tag{23}$$

with $u_n = \tilde{d}'_n/\tilde{b}_n$. This requires $1 + 4(n-1)$ FLOPs, which means that the specialized algorithm has a total number of FLOPs given by

$$\text{FLOPs} = 9n - 3 \propto \mathcal{O}(n). \tag{24}$$

C. LU-decomposition

In order to solve equation 7 using LU-decomposition we will implement the functions `lu_factor` and `lu_solve` from SciPy's Linalg Library[2].

D. Comparison

In order to test these algorithms, we will use a function for the right hand side of equation 1 given by

$$f(x) = 100e^{-10x}, \tag{25}$$

which has closed-form solution

$$u_s(x) = 1 - (1 - e^{-10})x - e^{-10x}. \tag{26}$$

In addition to comparing the algorithms' solutions, we will also compare their CPU times for several grid sizes n.

IV. RESULTS

Figures 1a and 1b compare the solution of the Thomas algorithm to the analytic solution for matrix sizes $n = 10$ and $n = 100$. Figure 1c compares the solution of the specialized algorithm to the analytic solution for a matrix of size $n = 100$. The relative errors of the specialized algorithm for different values of n are presented in table I.

Grid size, n	1e1	1e2	1e3	1e4	1e5	1e6
Relative error	-1.18	-3.09	-5.08	-7.08	-9.08	-10.16

TABLE I. Logarithmic relative error of the specialized Thomas algorithm for different matrix sizes, n.

Figure 1d compares the solution of the LU-decomposition method to the analytic solution for a matrix of size $n = 100$. We note that while the Thomas algorithms support $n > 10^5$, it was impossible to run the LU-decomposition method for matrices of size $n = 10^5$ or larger.

Figure 2 shows the average CPU run times for the three methods with the standard deviations indicated as error-bars. The average was taken over 5 runs. Note that the horizontal lines are not error-bars, but a consequence of the errors being very small. We observe that the Thomas algorithm and the special Thomas algorithm both have a linear relationship between n and the run time, whilst the LU-decomposition method does not. The latter has a more rapid growth than a linear relationship would. We also observe that the lower n-values have a larger uncertainty in time.

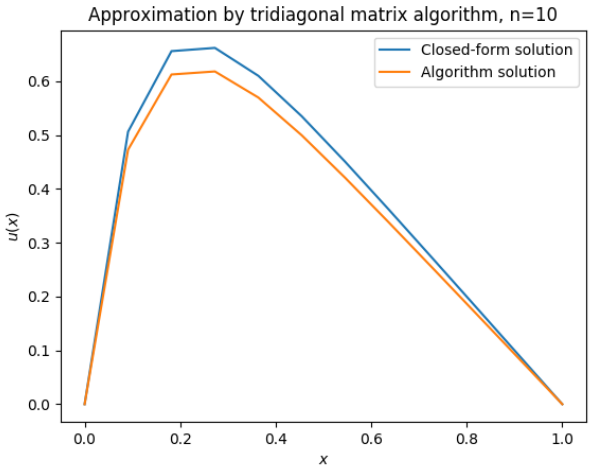
V. DISCUSSION

All the algorithms used in this project managed to reproduce the closed-form solution for matrix sizes $n \geq 100$. This is reflected in the relative errors for the specialized Thomas algorithm, table I, where we saw that the error became significantly smaller as the grid size increased.

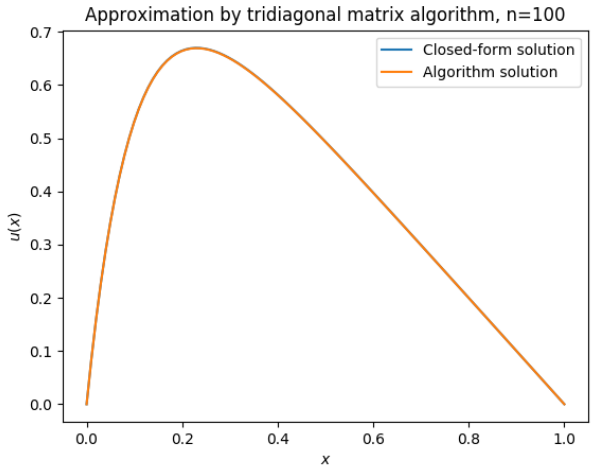
From the CPU times (figure 2) we saw that the LU-decomposition method was significantly slower than the other two methods, particularly for the larger grid sizes. In particular, the LU-decomposition method was unable to run on a standard laptop for matrix sizes above $n = 10^5$, without using hard drive memory. There are two aspects of the LU-decomposition method that may explain this.

First, the number of floating point operations required by the LU-decomposition method is of order n^3[3]. This is a very large number compared to the other two algorithms, both of which only require $\mathcal{O}(n)$ floating point operations. This results in the LU-decomposition method being much slower for large n.

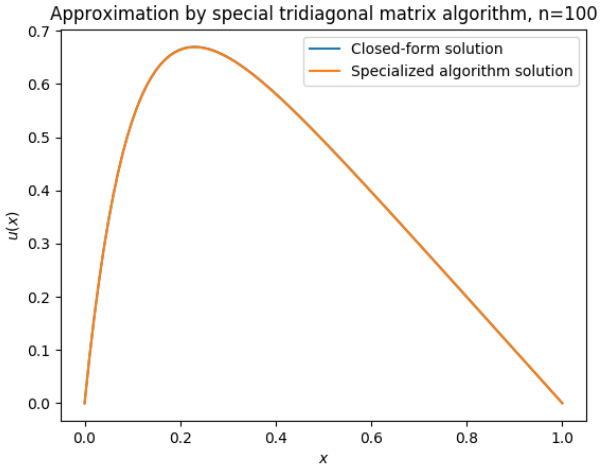
Second, the SciPy functions that performs the LU-decomposition and solves the matrix equation continually fetch and store matrix elements. This causes the functions to use significantly more memory than the Thomas algorithms, which in turn results in the method being unusable for large values of n. There simply isn't enough RAM on an ordinary laptop for handling $n \geq 10^5$.



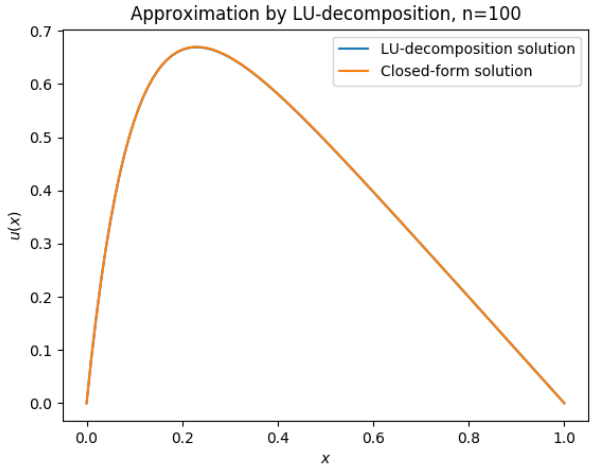
(a) Thomas algorithm, $n = 10$.



(b) Thomas algorithm, $n = 100$.

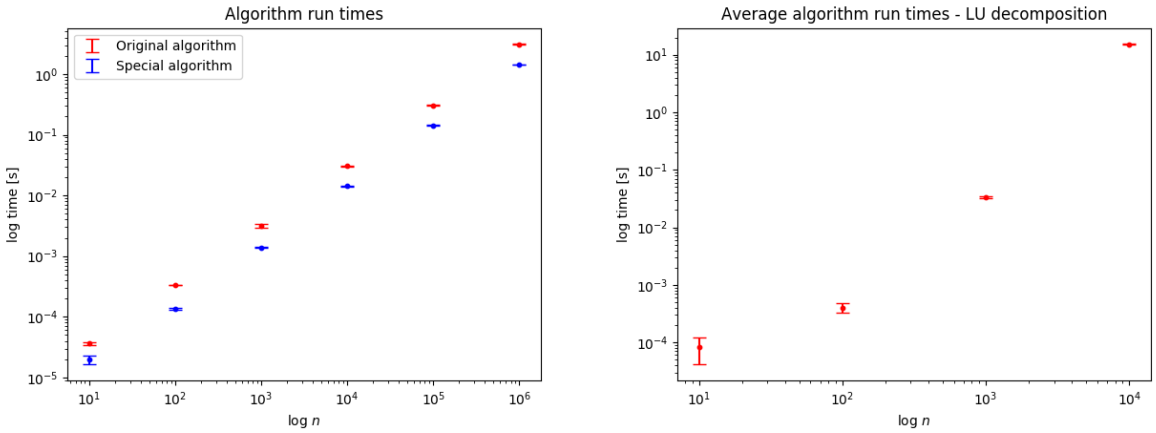


(c) Specialized Thomas algorithm, $n = 100$.



(d) LU-decomposition method, $n = 100$.

FIG. 1. Comparison of the algorithm solution and the analytical solution for the function f (25) for different sized matrices.



(a) Generalized and specialized Thomas algorithms.

(b) LU-decomposition algorithm.

FIG. 2. Log-log plots of the average time consumed by the three algorithms as a function of grid size n with errors.

The strength of the Thomas algorithms is their use of the tridiagonal matrix symmetries. This makes them able to solve the problem for larger grid sizes, where the LU-decomposition method fails. On the other hand, this makes them suitable for specific cases only, while the LU-decomposition method can be used in a larger variety of cases.

An aspect of all three methods with regards to the CPU times was a significantly larger error in the times for small n-values. The calculations for small grid sizes were a lot faster than those for larger grids. In fact, the runtimes were comparable to the CPU clock's granularity. The clock's precision simply wasn't high enough to capture the rapid calculation time.

The relationship between runtimes and number of floating point operations appears to be proportional. As mentioned above, whereas the Thomas algorithms perform $\mathcal{O}(n)$ floating point operations, the LU-decomposition method requires $\mathcal{O}(n^3)$. This is reflected in the runtime plots. The Thomas algorithms both have a linear relationship between the time required and n, while the LU-decomposition method grows faster.

VI. CONCLUSION

The aim of this project was to explore the precision and differences of the three algorithms presented. Overall, we found that the fastest algorithm was the special algorithm because it took into account the symmetries and particularities of the problem. The more general algorithms might be able to solve a larger variety of problems, but they are significantly slower in doing so.

We also found that solving problems numerically requires awareness of the inherent limitations of ordinary laptops. Aspects such as available memory and decimal precision may cause unpredicted problems.

[1] L.H. Thomas. Elliptic problems in linear differential equations over a network. *Watson Sci. Comput. Lab Report, Columbia University, New York*, 1949.

[2] The SciPy community. Scipy linear algebra library (scipy.linalg). https://docs.scipy.org/doc/scipy/reference/linalg.html.

[3] Morten Hjorth-Jensen. Computational physics lectures: Linear algebra methods. Lecture Notes, Sep 2018. http://compphysics.github.io/ComputationalPhysics/doc/pub/linalg/html/linalg.html.