FYS4150 – Project 1

Joachim Falck Brodin, Fredrik Jaibeer Mahal Nordeng,
and Endrias Getachew Asgedom

September 9, 2020

Abstract

Numerical methods for solving the one-dimensional Poisson equation with Dirichlet boundary conditions are investigated. In this report, we show how to discretize a one-dimensional Poisson equation and transform it into a linear tridiagonal set of equations. To solve these equations numerically, three different algorithms are developed (i.e., general tridiagonal Gaussian elimination, special tridiagonal Gaussian elimination, and LU-decomposition) and the results are analysed in terms of their computational cost, memory usage, and numerical error. Comparing our numerical solution with the closed form solution, we observe that the numerical approximation error is proportional to the square of the discretization step size, until we experience the loss of precision, due to round off error, when the discretization step size is less then $\sim 10^{-5}$. Moreover, the specialized algorithm out performs the two other algorithms in terms of computational cost and memory usage.

Contents

GitHub repository at `https://github.com/endrias34/FYS4150`

1 Introduction

Problems in the field of computational sciences is often formulated in terms of differential equations. Unfortunately, most of these differential equations do not have a closed form solution and hence needs to be solved using numerical methods. In this project, we develop algorithms for a numerical method that solves the one-dimensional Poisson equation with Dirichlet boundary conditions. We use central Finite Difference (FD) method to approximate the second-order differentiation and consequently cast the Poisson equation into a tridiagonal matrix equations. This linear system of equations may be solved using a Gaussian elimination method customized for tridiagonal matrices, also known as Thomas algorithm. Two versions of the Thomas algorithm are considered here; the first assumes a general tridiagonal matrix while the second assumes a Toepliz tridiagonal matrix. As a benchmark, we have also used a more general LU-decomposition method. The accuracy, computational efficiency, and memory usage of the different algorithms are quantitatively analysed.

This report is structured in the following way: First we will briefly present how to discretize the one-dimensional Poisson equation and generate the tridiagonal matrix equations. Second, we will show how to formulate the three different algorithms used to solve the problem. We then analyse the performance of the different algorithms in terms of the number of floating point operations (FLOPs), CPU time, and numerical approximation and precision errors. Finally, we will make a concluding remark.

2 Theory

To discretize the one-dimensional Poisson equation and write it in a matrix-vector form, we start with the continuous equation and its corresponding boundary conditions

$$-u''(x) = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0, \tag{1}$$

where $f(x)$ is a known function, called the *source term* and $u(x)$ is a dimension-less physical quantity defined in a dimension-less domain x. To solve equation (1) numerically, we first need an approximation of the second derivative of u. This can be achieved by performing the Taylor expansion of $u(x)$ around x

$$u(x \pm h) = u(x) \pm hu'(x) + \frac{h^2 u''(x)}{2!} \pm \frac{h^3 u'''(x)}{3!} + \mathcal{O}(h^4). \tag{2}$$

To find an expression containing only the second derivative, we perform the following operation

$$u(x+h) + u(x-h) = 2u(x) + \frac{2h^2 u''}{2!} + \mathcal{O}(h^4). \tag{3}$$

We can now solve for u'' as

$$u''(x) = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + \mathcal{O}(h^2). \tag{4}$$

Notice that the approximation error in equation (4) is $\mathcal{O}(h^2)$ since we divided equation (3) by h^2.

For a discrete domain $x_i = ih$ with $x_0 = 0$ and $x_{n+1} = 1$ as the boundary points, our unknown physical quantity u is represented as v_i. Here, the step size is given by $h = 1/(n+1)$. The boundary conditions can now be written as $v_0 = v_{n+1} = 0$. Finally, the discrete form of equation (1) is thus given by

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \ldots, n, \tag{5}$$

where $f_i = f(x_i)$ is the discrete version of $f(x)$. To represent equation (5) in a matrix-vector form, we multiply equation (5) with h^2 on both sides and write out the equations for all i as follows

$$\begin{aligned}
2 * v_1 - 1 * v_2 &= f_1 * h^2 \\
-1 * v_1 + 2 * v_2 - 1 * v_3 &= f_2 * h^2 \\
-1 * v_2 + 2 * v_3 - 1 * v_4 &= f_3 * h^2 \\
&\vdots \\
-1 * v_{n-2} + 2 * v_{n-1} - 1 * v_n &= f_{n-1} * h^2 \\
-1 * v_{n-1} + 2 * v_n &= f_n * h^2.
\end{aligned} \tag{6}$$

From equation (6) we can clearly see how equation (5) can be represented in a matrix-vector form as

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{f}}, \tag{7}$$

where $\tilde{f}_i = f_i * h^2$,

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \ldots \\ v_{n-1} \\ v_n \end{bmatrix} \quad \text{and } \tilde{\mathbf{f}} = \begin{bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \ldots \\ \tilde{f}_{n-1} \\ \tilde{f}_n \end{bmatrix}.$$

Here we can see how each row in the matrix-vector equation corresponds to the sequence of terms as we wrote it out in equation (6). We can also observe that $\mathbf{A}$ is tridiagonal matrix.

2.1 Gaussian Elimininationfor a Tridiagonal Matrix

Gaussian elimination, or row reduction, is a method for solving a system of linear equations. In the form presented here the procedure is also known as

the *Thomas algorithm*. We rewrite our matrix $\mathbf{A}$ in terms of one-dimensional vectors a, b, c and the linear equations can then be presented on the form

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & & \vdots \\ 0 & a_3 & b_3 & c_3 & 0 & \\ \vdots & 0 & a_4 & b_4 & c_4 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \tilde{f}_4 \\ \vdots \\ \tilde{f}_n \end{bmatrix}. \tag{8}$$

Next, we eliminate the a_2 from the second row. Here, the first step is to subtract $\frac{a_2}{b_1}$ times the first row from the second row and obtain

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ 0 & b_2' & c_2 & 0 & & \vdots \\ 0 & a_3 & b_3 & c_3 & 0 & \\ \vdots & 0 & a_4 & b_4 & c_4 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{f}_1 \\ \tilde{f}_2' \\ \tilde{f}_3 \\ \tilde{f}_4 \\ \vdots \\ \tilde{f}_n \end{bmatrix}, \tag{9}$$

where $b_2' = b_2 - \frac{a_2}{b_1} c_1$ and $\tilde{f}_2' = \tilde{f}_2 - \frac{a_2}{b_1} \tilde{f}_1$. For the following rows we proceed in the same manner and we can generally express the different rows as

$$b_i' = b_i - \frac{a_i}{b_{i-1}'} c_{i-1}, \tag{10}$$

with $b_1' = b_1$, and

$$\tilde{f}_i' = \tilde{f}_i - \frac{a_i}{b_{i-1}'} \tilde{f}_{i-1}', \tag{11}$$

where $\tilde{f}_1' = \tilde{f}_1$. Finally, the system of equations become an upper triangular matrix equation of the form

$$\begin{bmatrix} b_1' & c_1 & 0 & \dots & \dots & 0 \\ 0 & b_2' & c_2 & 0 & & \vdots \\ 0 & 0 & b_3' & c_3 & 0 & \\ \vdots & 0 & 0 & b_4' & c_4 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & 0 & b_n' \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{f}_1' \\ \tilde{f}_2' \\ \tilde{f}_3' \\ \tilde{f}_4' \\ \vdots \\ \tilde{f}_n' \end{bmatrix}. \tag{12}$$

The process of converting the matrix equation from (8) to (12) is known as *forward substitution*. To find the solution v_i for the matrix equation (12), we can start from the bottom and solve for v_n, then continue to solve for v_{n-1} until we reach v_0. This way of solving the matrix equation is known as *backward substitution* and it can be generalized to [1]

$$v_i = \begin{cases} \tilde{f}_n'/b_n', & \text{if } i = n \\ (\tilde{f}_i' - c_i v_{i+1})/b_i', & \text{otherwise.} \end{cases} \tag{13}$$

2.2 Gaussian Elimination for a Toepliz Tridiagonal Matrix

Matrix **A** in equation (7) has a Toepliz tridiagonal structure. This means the non-zero diagonal elements have the same value along all the diagonal. In our case, we have $a_i = c_i = -1$ and $b_i = 2$. The fact that the elements of matrix **A** only have two different values greatly simplifies the Gaussian elimination method discussed in the previous section. The forward substitution step in equations (10) and (11) can now be generalized to

$$b'_i = \frac{i+1}{i}, \qquad i = 1, \ldots, n-1, \tag{14}$$

and

$$\tilde{f}'_i = \tilde{f}_i + \frac{\tilde{f}'_{i-1}}{b'_{i-1}}, \qquad i = 1, \ldots, n-1. \tag{15}$$

Similarly, the backward substitution step in equation (13) simplifies to

$$v_i = \frac{\tilde{f}'_i + v_{i+1}}{b'_i} \qquad i = n-2, n-1, \ldots, 0. \tag{16}$$

2.3 LU-decomposition Method

The LU-Decomposition method is the most commonly used method to solve a linear system of equation which has a densely populated matrix. The method is a special form of Gaussian elimination method and considers we have a non-singular matrix **A** that can be decomposed into a product of two triangular matrices **L** and **U**, where **L** is a lower-triangular matrix with all diagonal entries equal to 1 and **U** is an upper-triangular matrix [2]. To solve the linear set of equation $\mathbf{Av} = \tilde{\mathbf{f}}$ using LU-decomposition, we substitute $\mathbf{A} = \mathbf{LU}$ such that

$$\mathbf{LUv} = \mathbf{Ly} = \tilde{\mathbf{f}}, \tag{17}$$

where $\mathbf{y} = \mathbf{Uv}$. Consequently, the solution **v** can now be obtained by solving the following two triangular linear set of equations

$$\begin{aligned} \mathbf{Ly} &= \tilde{\mathbf{f}} \\ \mathbf{Uv} &= \mathbf{y}. \end{aligned} \tag{18}$$

Solving triangular set of equations is trivial, as the lower triangular equation can be solved using *forward substitution* while the upper triangular equation may be solved by *backward substitution*.

3 Method

The three different algorithms required to solve the tridiagonal matrix equation are implemented using C++ and the corresponding source codes can be found in the github repository address linked to this report. In this section, we outline the algorithms structure using pseudo-codes, discuss about the FLOPs counts, and the memory handling of each of the algorithms.

3.1 General Tridiagonal Matrix Algorithm

The algorithm we will present here is known as the Thomas algorithm. The main elements of the algorithm are the *forward* and *backward* substitution. The algorithm take the vectors $\mathbf{a} \in \mathbb{R}^{n-1}$, $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{c} \in \mathbb{R}^{n-1}$, and $\tilde{\mathbf{f}} \in \mathbb{R}^n$ as input and outputs the solution $\mathbf{v} \in \mathbb{R}^n$ (cf. Algorithm 1). Notice that we do not reserve storage for the full $\mathbf{A} \in \mathbb{R}^{n \times n}$ matrix, but rather only store the nonzero tridiagonal elements as vectors. This allows us to utilize this algorithm for solving problems with a very large number of n without running into a memory allocation problem.

Algorithm 1 General Tridiagonal Matrix Algorithm

Input: $\mathbf{a}, \mathbf{b}, \mathbf{c}, \tilde{\mathbf{f}}$
Output: $\mathbf{v}$

1: $b'_0 = b_0$ // First element along main diagonal
2: $\tilde{f}'_0 = \tilde{f}_0$ // First element in the RHS
3:
4: // Forward substitution
5: **for** $i = 1$ to $i = n - 1$ **do**
6: $b'_i = b_i - a_{i-1} * c_{i-1}/b'_{i-1}$ // Eliminate lower diag
7: $\tilde{f}'_i = \tilde{f} - a_{i-1} * \tilde{f}_{i-1}/b'_{i-1}$ // Change RHS
8: **end for**
9:
10: // Backward substitution
11: $v_{n-1} = \tilde{f}'_{n-1}/b'_{n-1}$ // Last element of the solution
12:
13: **for** $i = n - 2$ to $i = 0$ **do**
14: $v_i = (\tilde{f}'_i - c_i * v_{i+1})/b'_i$
15: **end for**

The number of FLOPs for the Thomas algorithm is composed of $6(n-1)$ for the forward and $3(n-1)$ for the backward substitution. Moreover, pre-computing a_{i-1}/b'_{i-1} (cf. line 6 and 7 in Algorithm 1) in the forward substitution, we can reduce the number of FLOPs by $(n-1)$. Therefore, in total the Thomas algorithm requires $8(n-1)$ FLOPs.

3.2 Toepliz Tridiagonal Matrix Algorithm

The Toepliz structure of matrix $\mathbf{A}$ allows us to reduce both the memory demand and the FLOPs of the Thomas algorithm. This specialized form of the Thomas algorithm takes only $\tilde{\mathbf{f}} \in \mathbb{R}^n$ as input and outputs the solution $\mathbf{v} \in \mathbb{R}^n$. Similarly, the FLOPs reduce to $2(n-1)$ for the forward and another $2(n-1)$ for the backward substitution. The reduction in the forward substitution is due to the fact that the diagonal elements in line 6 of Algorithm 2 can now be fully precomputed since they have analytical expression. Therefore, the total FLOPs for the specialized Thomas algorithm is $4(n-1)$ which is a reduction by half compared to the general Thomas algorithm.

Algorithm 2 Toepliz Tridiagonal Matrix Algorithm

Input: $\tilde{\mathbf{f}}$
Output: $\mathbf{v}$

$b'_0 = 2$ // First element along main diagonal
$\tilde{f}'_0 = \tilde{f}_0$ // First element in the RHS

// Forward elimination
for $i = 1$ to $i = n - 1$ **do**
    $b'_i = (i + 2)/(i + 1)$ // Eliminate lower diag (pre-calculated)
    $\tilde{f}'_i = y_i + \tilde{f}_{i-1}/b'_{i-1}$ // Change RHS
end for

// Backward elimination
$v_{n-1} = \tilde{f}'_{n-1}/b'_{n-1}$ // Last element in solution
for $i = n - 2$ to $i = 0$ **do**
    $v_i = (\tilde{f}'_i + v_{i+1})/b'_i$
end for

3.3 LU-decomposition based Toepliz Tridiagonal Matrix Algorithm

Here, we used *Armadillo* (a C++ Linear Algebra Library [3]) to perform LU-decomposition and solve the Toepliz triangular set of linear equations. Armadillo has a built in function called *solve* that is a highly optimized function that performs LU-decomposition by default to solve a linear set of equations. However, in our pseudo-code (cf. Algorithm 3) we have separately specified the LU-decomposition and solving the two triangular equations for clarity of presentation. Any LU-decomposition based algorithm is composed of three steps; first LU-decomposition of the system matrix, and then triangular forward substitution, and at last triangular backward substitution. The pseudo-code in Algorithm 3 takes as input $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\tilde{\mathbf{f}} \in \mathbb{R}^n$, and outputs the solution $\mathbf{v} \in \mathbb{R}^n$. Here, it is pertinent to note that Algorithm 3 takes a matrix as input, and hence requires a large amount of memory to store this matrix. Nevertheless, LU-decomposition is a method of choice when solving a linear set of equation with a densely populated matrix. For the case $\mathbf{A} \in \mathbb{R}^{n\times n}$, the FLOPs for LU-decomposition takes $O(2/3n^3)$, and the two substitutions take each $\sim O(n^2)$ [2]. However, here we are interested in solving a system of equations with a Toepliz tridiagonal matrix structure. For such a case the LU-decomposition has an analytical expression and takes only $O(3n)$ FLOPs, the forward substitution takes $O(2n)$ while the backward substitution takes $O(3n)$ [4]. In total the number of FLOPs for LU-decomposition based Toepliz tridiagonal matrix system of equations is $8n$.

Algorithm 3 LU-decomposition based Toepliz Tridiagonal Matrix Algorithm

Input: $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{f}}$
Output: **v**

$lu(L, U, A)$ // LU-decomposition
$y = solve(L, \tilde{f})$ // Forward substitution
$v = solve(U, y)$ // Backward substitution

3.4 Error Analysis

The errors generated by the different algorithms are quantified by comparing the results of the algorithm with the closed form (analytical) solution. We use the relative error which is defined by

$$\epsilon_i = \left| \frac{v_i^{num} - v_i^{ana}}{v_i^{ana}} \right|, \quad i = 1, \ldots, n, \tag{19}$$

where v_i^{num} and v_i^{ana} are the numerical and analytical solutions at the i^{th} descritization location, respectively. Moreover, $|.|$ is the absolute value operator. The analytical solution for the one-dimensional Poisson equation with a known source function is derived in Appendix A.

4 Results

Here, we present the quantitative analysis of the different algorithms. The computational efficiency and error analysis is investigated in detail.

4.1 Computational Time

To compare the computational efficiency of the three different algorithms discussed in this project we analysed their execution time elapsed in terms of CPU time (cf. Tabel 1). The time is computed by taking an average of 1000 runs of each of the algorithms. Considering the number of FLOPs count for the three different algorithms, one might expect to see the CPU time for LU-decomposition and general Thomas algorithm to be similar while the special Thomas algorithm being twice as fast as the two other algorithms. However, what we actually observe from the CPU time is that the special Thomas algorithm is just slightly faster than the general Thomas algorithm and the LU-decomposition algorithm is at least an order of magnitude slower then the two Thomas algorithms. Another observation that we can make from the CPU time is that, as the matrix size gets larger the special Thomas algorithm becomes much faster than the general Thomas algorithm. Furthermore, the CPU time difference between the two Thomas algorithms and the LU-decomposition reduces as the matrix size gets larger.

One possible reason for explaining the discrepancy between the FLOPs count and the CPU time is that, not all operations consume the same CPU time. For example it is known that division is the most computationally costly operation. Thus, comparing the two Thomas algorithms, it is possible to see that the number of division operations are the same. Therefore, it might be possible

Matrix size	General [s]	Special [s]	LU-decomp. [s]
10×10	$6.970 \cdot 10^{-7}$	$6.910 \cdot 10^{-7}$	$1.139 \cdot 10^{-5}$
100×100	$2.114 \cdot 10^{-6}$	$2.047 \cdot 10^{-6}$	$6.882 \cdot 10^{-5}$
1000×1000	$1.676 \cdot 10^{-5}$	$1.577 \cdot 10^{-5}$	$2.722 \cdot 10^{-4}$

Table 1: CPU time in seconds for the three different algorithms (i.e., General and special Thomas, and LU-decomposition).

that the division operation is the reason we are not seeing that the special algorithm run twice as fast as the general algorithm. On the other hand, the very slow LU-decomposition algorithm could be because Armadillo might have treated the matrix $\mathbf{A}$ as a fully populated matrix, and even spend some time checking the best algorithm to use.

4.2 Algorithm Benchmark and Error Analysis

Due to the availability of the analytical solution to the one-dimensional Poisson equation, we have bechmarked our numerical algorithms by computing the relative error. We considered three different cases (i.e., $n = 10$, $n = 100$, and $n = 1000$) and show the results only for the general Thomas algorithm. This is mainly because the two other algorithms provide a similar result as that of the general Thomas algorithm. Figure 1 presents the numerical solution for the three different cases as well as the corresponding analytical solution. Here, we can observe that for the cases $n = 100$ and $n = 1000$ the numerical solutions looks to be the same as the analytical solution. To quantify the difference between the numerical and analytical solutions, we computed the relative error for the three different cases and shown it in logarithmic scale in Figure 2. Notice that the relative error is the same for all values of x and that for $n = 10$, $\epsilon \sim 10^{-1.2}$, for $n = 100$, $\epsilon \sim 10^{-3}$, and for $n = 1000$, $\epsilon \sim 10^{-5}$.

To analyse the effect of having extremely small descritization step size (or having very large number of grid points n) on the precision of the numerical algorithm, we computed the relative error for the general Thomas algorithm for different values of descritization step sizes (cf. Tabel 2 and Figure 3). The logarithmic relative error as a function of the logarithmic descritization step size shows a linear trend and it reduces until it reaches a descritization step size of $\sim 10^{-5}$ (or $n = 10^5$), where it increases again. This loss of precision is due to round off errors which arises from the fact that computers do not represent numbers with an infinite decimal places and hence need to be rounded off to a number the computer can represent.

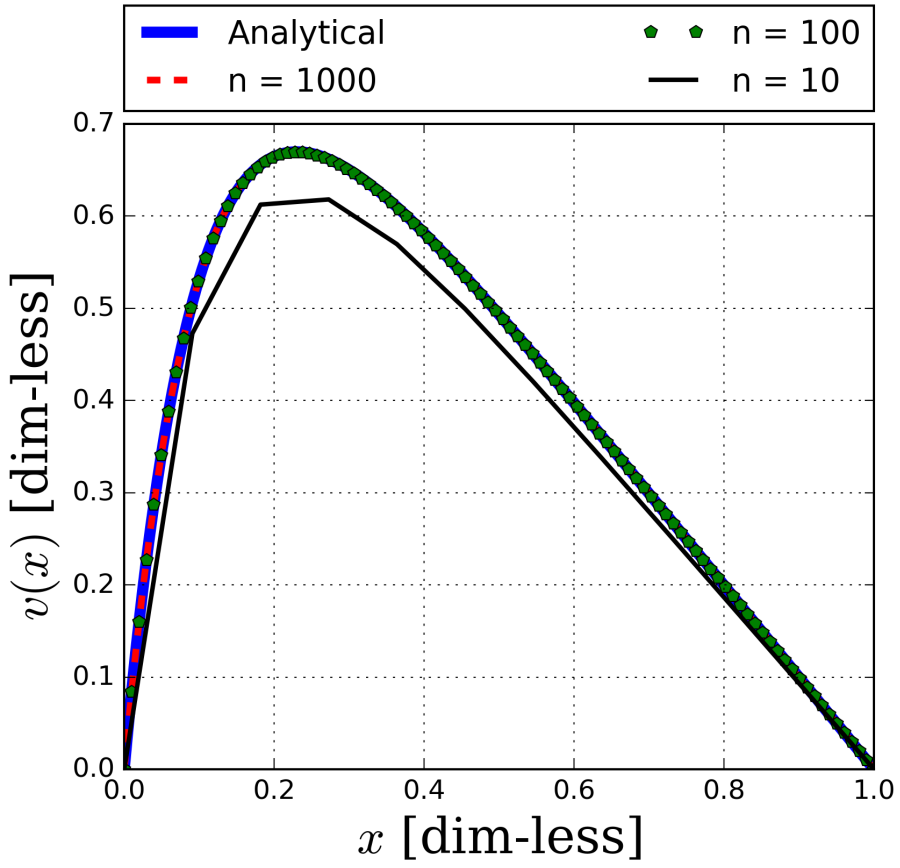

Figure 1: Numerical approximation and analytical solution to the 1D Poisson equation. The numerical solution is shown for different $n \times n$ matrix sizes. The convergence of the numerical method is apparent as its graph approaches the analytical solution as the number of points n grows. When $n >= 10^2$, the numerical solution becomes indistinguishable from the analytical solution.

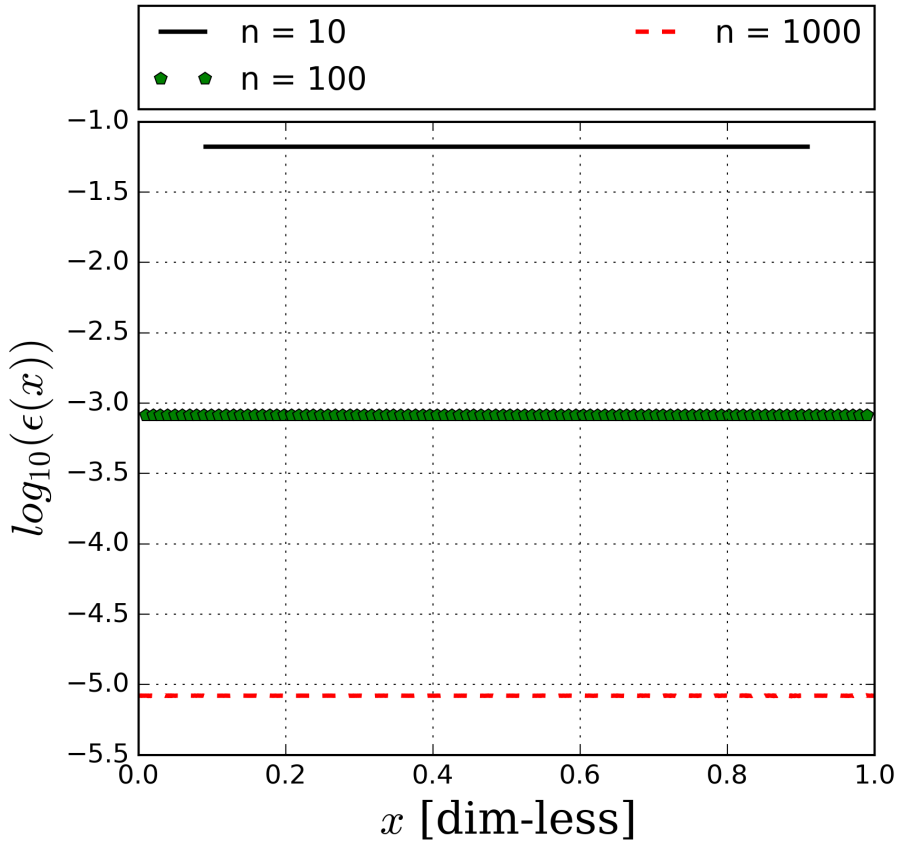

Figure 2: Relative error to the numerical approximations of the 1D Poisson equation. The relative error is shown for different $n \times n$ matrix sizes, $x \in (0, 1)$.

Grid points (n)	Step size ($log_{10}(h)$)	Relative error ($log_{10}(\epsilon)$)
10^1	-1.041	-1.179
10^2	-2.004	-3.0880
10^3	-3.000	-5.0807
10^4	-4.000	-7.0791
10^5	-5.000	-8.843
10^6	-6.000	-6.075
10^7	-7.000	-5.525

Table 2: Logarithmic relative error as a function of logarithmic discretization step size (or the number of grid points). Here we have calculated the maximum relative error for each of the discretization step sizes. We note again that it start rising after 10^5.

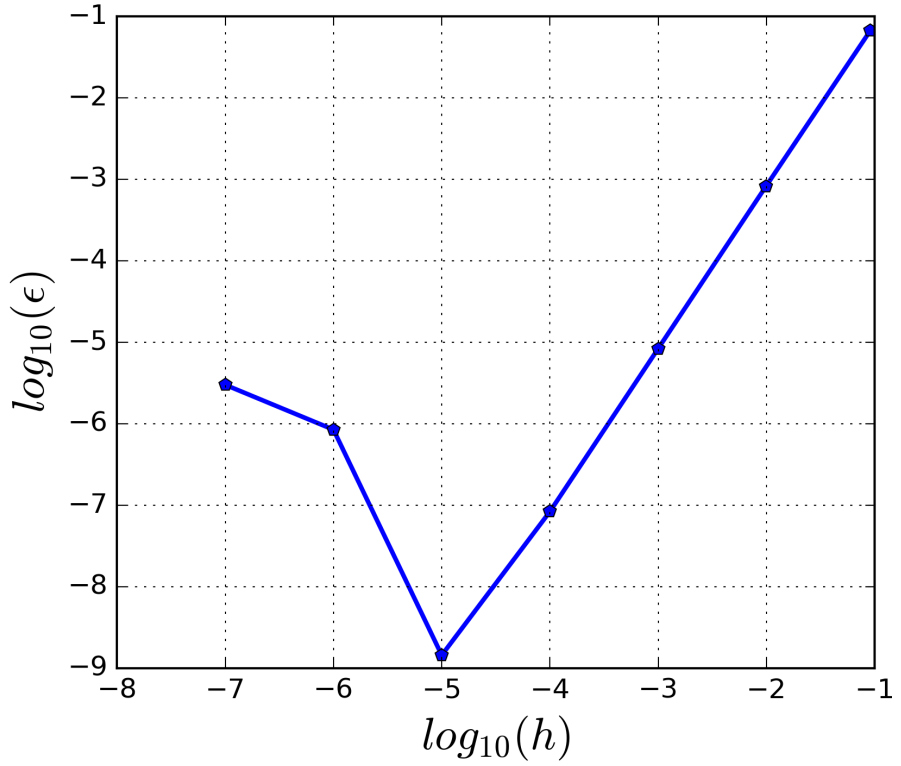

Figure 3: A plot of the logarithmic relative error as a function of the logarithmic discretization step size. Here we can see that the relative error follows our expectation for the truncation error, with a slope value of -2 on a logarithmic scale, up to a point of $n \sim 10^5$.

5 Discussion

The memory usage of the three different algorithms depends on whether the algorithm uses static or dynamic memory allocation. In addition, it also depends whether the algorithm has to store vectors or matrices. In this project we have allocated all our variables using dynamic memory. It is also important to notice that the LU-decomposition algorithm has to store a matrix while the two Thomas algorithms store only vectors. Consequently, we would run into a memory storage problem when using the LU-decomposition algorithm. For example, consider a problem with the number of grid points $n = 10^5$. This implies matrix $\mathbf{A} \in \mathbb{R}^{10^5 \times 10^5}$. To store this matrix in memory requires $10^{10} * 64/(8 \cdot 10^9) = 80$GB for double precision numbers. Unfortunately, this is by far larger than a normal personal computer memory ($\sim$ 8GB).

Another thing to notice is that the relative error as a function of the grid points is constant. This makes our numerical solution to be scaled easily from for example $n = 10$ to $n = 1000$ as long as we know the analytical function at only

one grid point (except at the boundaries). This is quite attractive when trying to reduce the computational cost by solving the problem with small number of grid points and scaling the result. On the other hand, if we do not know the analytical function values other than the boundaries, using low numbers of n might still be used as a way to take a look at where we need to add grid points.

6 Conclusion

Three different numerical algorithms were developed for solving the one dimensional Poisson equation with Dirichlet boundary conditions. The algorithms utilize the fact that the problem can be casted into a matrix equation. The general Thomas algorithm uses the tridiagonal structure of the matrix equation while the special Thomas and LU-decomposition algorithms use the Toepliz tridiagonal structure for the matrix $\mathbf{A}$. This allows the three algorithms to be computationally efficient compared to the general Gaussian elimination algorithm. Moreover, the results of the three algorithms approach the corresponding analytical solution with a small relative error for the number of grid points $n \geq 100$. The special Thomas algorithm out performed the two other algorithms by having the fastest computational speed and the smallest memory requirement. The LU-decomposition algorithm has the highest memory requirement and it was even impossible to use the method in a personal computer for $n > 10^4$. Finally, we have observed that the numerical precision of the algorithms increases together with n until it reaches the round off limit $n \sim 10^5$.

References

[1] B. N. Datta, *Numerical Linear Algebra and Applications*. SIAM, 2010.

[2] M. Hjorth-Jensen, "Computational physics," University Lecture Notes, 2015.

[3] C. Sanderson and R. Curtin, *Armadillo: a template-based C++ library for linear algebra.*, Journal of Open Source Software, http://dx.doi.org/10.21105/joss.00026, 2016.

[4] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge, USA: Cambridge University Press, 1992.

A Analytical Solution

The one-dimensional Poisson equation with Dirichlet boundary condition has a closed form solution for a known and twice integrable source function. To derive this closed form solution, we first rewrite equation (1) for a source function $f(x) = 100e^{-10x}$ as

$$-u''(x) = 100e^{-10x}, \qquad x \in (0,1), \qquad u(0) = u(1) = 0. \tag{20}$$

A closed form solution to equation (20) can now be found by integrating it twice respect to x. This results

$$u(x) = -e^{-10x} + A + Bx, \tag{21}$$

where A and B are integration constants. We now impose the Dirichlet boundary condition to equation (21) and find that $A = 1$ and $B = e^{-10} - 1$. Finally, replacing the values of A and B into equation (21) results the analytical solution

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x}. \tag{22}$$