FYS 4150 - Computational Physics
Project 1: Solving Poisson's equation in one dimension

Markus Leira Asprusten   Maren Rasmussen   Metin San
https://github.com/MetinSa/FYS4150/tree/master/Project_1

4. September 2018

Abstract

This project involves solving the one-dimensional Possion equation with dirichlet boundary conditions using two different algorithms. The first method is the tridiagonal matrix algorithm while the second is the LU decomposition. The two methods are then compared in terms of efficiency. The conclusion of the project is that a specialized version of the tridiagonal algorithm is much faster.

1. Introduction

The main goal of this project is to get familiar with the programming language c++. The focus will be directed at obtaining an understanding of vector and matrix operations with memory allocation in addition to managing library packages such as the c++ library Armadillo.

We will address this issue by studying and solving Possion's differential equation numerically. Many of the most important differential equations in physics can be written as linear second-order differential equations. It is therefore of importance to be able to to solve these systems.

The report starts of with a theory section where the Possion equation is introduced and discretized. What follows is a method and algorithm section where we derive and experiment with two specific numerical algorithms which can be applied to solve the equation at hand. The first is the tridiagonal matrix algorithm, and the second is the LU-decomposition method. The more general tridiagonal matrix algorithm is further tailored to specifically deal with the Possion equation in order to increase the methods efficiency. Both algorithms are then tested at different precision levels using varying grid points. The speed and efficiency, along with the errors produced from the two methods are then presented in the results section, and further compared and discussed in the final discussion section.

2. Theoretical Background

2.1. The Poisson Equation.

Poisson's equation is a classical and well known differential equation from electromagnetism. The equation describes the potential field Φ generated from a charge distribution $\rho(\mathbf{r})$. For three dimensions, the equation is given by

$$\nabla^2\Phi = -4\pi\rho(\mathbf{r}), \tag{1}$$

where $\nabla =$ is the Laplace operator. If both Φ and $\rho(\mathbf{r})$ is spherically symmetric, the equation can be simplified to a one dimensional equation in r,

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

By substituting $\Phi = \phi(r)/r$, we can simplify the equation even more, giving

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

The final equation can be written in a general form by letting $\phi \to u$ and $r \to x$, and defining the right hand side of the equation as f,

$$-u''(x) = f(x). \tag{2}$$

In this study we will assume that the source term is $f(x) = 100e^{-10x}$ with $x \in [0,1]$. We will also assume that we have Dirichlet boundary conditions, such that $u(0) = u(1) = 0$. With these assumptions the equation have an exact solution on the form $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. The exact solution is of importance as it can be compared to the numerical calculation in order to verify the accuracy of our results.

2.2. Discretization of the problem.

To solve the Poisson equation nummerically, we need to discretize the problem. The discretization can be done using $(n+1)$ x-values, so that $x \in [x_0, x_1, x_2, ..., x_n]$ with $x_0 = 0$ and $x_n = 1$, and $u(x_i) = u_i$. The x-values are then given as

$$x_i = x_0 + ih,$$

where $h = (x_n - x_0)/n$ is the step size.

A discretized version of $u''(x)$ can be found using Taylor expansion. We know that

$$u(x+h) = u(x) + hu' + \frac{h^2}{2!}u'' + O(h^2),$$
$$u(x-h) = u(x) - hu' + \frac{h^2}{2!}u'' + O(h^2).$$

The $O(h^2)$-term is the remaining terms from the Taylor expansion, or the error we get by excluding these terms. By adding $u(x+h)$ and $u(x-h)$, we can derive the desired expression:

$$u(x+h) + u(x-h) = 2u(x) + \frac{2}{2!}h^2u'' + O(h^2),$$
$$u''(x) = \frac{u(x+1) + u(x-h) - 2u(x)}{h^2} + O(h^4),$$

$$u'' = \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} + O(h^2) \tag{3}$$

By excluding the rest of the terms in the Taylor expansion, and using the definition in equation (2) discretized, we can define $f_i^* = f_i h^2$. This reduces equation (3) to

$$u_{i+1} - u_{i-1} + 2u_i = f_i^*. \tag{4}$$

Inserting for specific i-value leaves us with a set of linear equations

$$\begin{aligned} -u_2 - u_0 + 2u_1 &= f_1^*, \\ -u_3 - u_1 + 2u_2 &= f_2^*, \\ &\vdots \\ -u_n - u_{n-2} + 2u_{n-1} &= f_{n-1}^*. \end{aligned}$$

This set of equations can also be represented in terms of matrices. The LHS of the equation can be splitted up into the product of a matrix $\hat{\mathbf{A}}$ and a vector $\hat{\mathbf{u}}$

$$\hat{\mathbf{A}}\hat{\mathbf{u}} = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \end{bmatrix},$$

and the RHS as a vector $\hat{\mathbf{f}}$

$$\hat{\mathbf{f}} = \begin{bmatrix} f_1^* \\ f_2^* \\ \vdots \\ f_{n-1}^* \end{bmatrix}.$$

This means that the set of equations can be written as a linear algebra problem on the form

$$\hat{\mathbf{A}}\hat{\mathbf{u}} = \hat{\mathbf{f}}.$$

3. Algorithm & Implementation

With the problem now formulated in terms of linear algebra, the next step is to solve it. We will tackle this problem through the implementation of two algorithms. The first is the Tridiagonal matrix algorithm, also known as the Thomas algorithm. The second is the LU-decomposition algorithm.

3.1. Tridiagonal Matrix Algorithm.

This algorithm is a simplified form of Gaussian elimination which can be used to solve tridiagonal systems of equations. In the general case, a tridiagonal system of n unknowns can be represented as

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i, \tag{5}$$

where $a_1 = c_1 = 0$. Or in matrix representation as $\mathbf{\hat{A}}\mathbf{\hat{u}} = \mathbf{\hat{f}}$. We spot that this corresponds to our linear algebra problem with Possion's Equation. Written out in the 4×4 case, this becomes

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}. \tag{6}$$

The algorithm is quite simple and consists of mainly two steps, a forward substitution and a backwards substitution. The forward substitution reduces the tridiagonal matrix $\mathbf{\hat{A}}$ to an upper tridiagonal matrix. This is achieved through Gaussian elimination. We want to get rid of the a_i terms located on the lower secondary diagonal. We perform the following row reduction on both sides of the equation

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \xrightarrow{\text{II}-\frac{a_2}{b_1}\text{I}} \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix}, \qquad \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} \xrightarrow{\text{II}-\frac{a_2}{b_1}\text{I}} \begin{bmatrix} f_1 \\ \tilde{f_2} \\ f_3 \\ f_4 \end{bmatrix}$$

where $\tilde{b_2} = b_2 - a_2c_1/b_1$, and $\tilde{f_2} = f_2 - f_1a_2/b_1$, and II and I denotes the row 1 and 2 in the $\mathbf{\hat{A}}$. Similarly for the second row

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \xrightarrow{\text{III}-\frac{a_3}{b_2}\text{II}} \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix}, \qquad \begin{bmatrix} f_1 \\ \tilde{f_2} \\ f_3 \\ f_4 \end{bmatrix} \xrightarrow{\text{III}-\frac{a_3}{b_2}\text{II}} \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ f_4 \end{bmatrix}$$

where $\tilde{b_3} = b_3 - a_3c_2/b_2$, and $\tilde{f_3} = f_3 - f_2a_3/b_2$. Finally we compute the last row reduction

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \xrightarrow{\text{IIII}-\frac{a_4}{b_3}\text{III}} \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & 0 & \tilde{b_4} \end{bmatrix}, \qquad \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ f_4 \end{bmatrix} \xrightarrow{\text{III}-\frac{a_3}{b_2}\text{II}} \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ \tilde{f_4} \end{bmatrix}$$

where $\tilde{b_4} = b_4 - a_4c_3/b_3$, and $\tilde{f_4} = f_4 - f_3a_4/b_3$.

We are then left with the row reduced form of the set of equations $\tilde{\mathbf{A}}\hat{\mathbf{u}} = \tilde{\mathbf{f}}$, or in matrix notation

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & 0 & \tilde{b_4} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ \tilde{f_4} \end{bmatrix}. \tag{7}$$

If one takes a closer look at the steps which we carried out, one notices the following pattern for $\tilde{b}$ and $\tilde{f}$. These can be expressed on the general form

$$\tilde{b_i} = b_i - \frac{a_i c_{i-1}}{\tilde{b}_{i-1}}, \qquad \tilde{f_i} = f_i - \frac{a_i \tilde{f}_{i-1}}{\tilde{b}_{i-1}}, \qquad i \in [2, 4], \tag{8}$$

where $b_1 = \tilde{b_1}$ and $f_1 = \tilde{f}$. In general for a $(n \times n)$ matrix we would have $i \in [2, n]$. The forward substitution has been implemented in the following way in c++:

```
f_tilde[1] = f[1];
double ab;
// forward substitution
for (int i = 2; i < n; i++){
    ab = a[i]/b[i-1];
    b[i] = b[i] - ab*c[i-1];
    f_tilde[i] = f[i] - ab*f_tilde[i-1];
}
```

Note that instead of allocating memory for a seperate $\tilde{b}$ array, we have rather reused the b array. We also compute $a_i/\tilde{b}_{i-1}$ at the start of the loop which saves us a FLOP as it appears in both the expression for $\tilde{b}$ and $\tilde{f}$.

The last part of the tridiagonal algorithm is the backwards substitution. By setting up the set of equations in (7), we are able to solve each of these for their respective solution u_i. The first equation along with its solution is then

$$\tilde{b}_1 u_1 + c_1 u_2 = \tilde{f}_1 \qquad \rightarrow \qquad u_1 = \frac{\tilde{f}_1 - c_1 u_2}{\tilde{b}_1},$$

where we have used that $b_1 = \tilde{b}_1$. Similarly for the second and the third rows

$$\tilde{b}_2 u_2 + c_2 u_3 = \tilde{f}_2 \qquad \rightarrow \qquad u_2 = \frac{\tilde{f}_2 - c_2 u_3}{\tilde{b}_2},$$

$$\tilde{b}_3 u_3 + c_3 u_4 = \tilde{f}_3 \qquad \rightarrow \qquad u_3 = \frac{\tilde{f}_3 - c_3 u_4}{\tilde{b}_3}.$$

For the final row, we simply get

$$\tilde{b}_4 u_4 = \tilde{f}_4 \qquad \rightarrow \qquad u_4 = \frac{\tilde{f}_4}{\tilde{b}_4}.$$

This is a result of the chosen dirichlet boundary conditions. Again we notice the solution u_i follows the following pattern

$$u_i = \frac{\tilde{f}_i - c_i u_{i+1}}{\tilde{b}_i}. \tag{9}$$

This is implemented in the code as

```
// backward substitution
u[n-1] = f_tilde[n-1]/b[n-1];         //setting the last term

for (int i = n-2; i > 0; i--){
    u[i] = (f_tilde[i] - c[i]*u[i+1])/b[i];
}
```

where we see that the last term has been computed separately as it differs from the general algorithm. The code prior to these two snippets addresses the allocation of memory to the different vectors that are to be used in the algorithm. The code displayed here has used classic c++ memory allocation. We have however also created corresponding codes to every program which use the package Armadillo. These can be found on the Github alongside the the standard code.

In general one of the most important aspects of any algorithm is its efficiency. The tridiagonal matrix algorithm is known to be a relatively fast algorithm as it only uses three diagonal vectors to represent the entire $(n \times n)$ matrix which severely reduces the number of floating point operations (FLOPS) required to solve the set of equations. We will assume that addition, subtraction, multiplication and division all counts as FLOPS. In reality, division operations are "heavier" and requires the most computation time. The forward substitution method requires 6 FLOPS for each iteration, and it is computed $(n-2)$ times which results in a total of $6(n-2)$ FLOPS. The backward substitution requires $2(n-2)+1$ FLOPS where the $+1$ term comes from the definition of the last term, which has to be computed just once. All together the algorithm requires $8(n-2)+1$ FLOPS. For large numbers of n, the algorithm can be said to require $O(8n)$ FLOPS, as the constants can be neglected.

3.1.1 **Optimizing the Tridiagonal Matrix Algorithm.** The number of floating point operations in the algorithm can be severely reduced if we tailor it to the Poisson equation. Since we are only interested in the tridiagonal matrix which resulted from the discretization of the second derivative, we can use the precomputed matrix $\hat{\mathbf{A}}$ with known diagonal elements. This allows us to rewrite the expressions for the forward and backwards substitution. If one inserts for the constant $a_i = c_i = -1$ and $b_i = 2$ into equations (8) and (9), we find that we can in fact rewrite these into the form

$$\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}} = \frac{i+1}{i}, \tag{10}$$

$$\tilde{f}_i = f_i + \frac{(i-1)\tilde{f}_{i-1}}{i}, \tag{11}$$

$$u_i = \frac{i}{i+1}(\tilde{f}_i + u_{i+1}). \tag{12}$$

Since the diagonal elements $\tilde{b}_i$ can be precomputed as they only depend on i, this calculation can be moved outside of the main algorithm. Further, we spot that we can rewrite equation (11) in terms of $\tilde{b}_{i-1} = i/(i-1)$, to the form

$$\tilde{f}_i = f_i + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}} \tag{13}$$

Which has now been reduced to 2 FLOPS down from 3. Similarly we rewrite the equation (12) in terms of $\tilde{b}_i$ to the from

$$u_i = \frac{\tilde{f}_i + u_{i+1}}{\tilde{b}_i}, \tag{14}$$

which has now also been reduced to 2 FLOPS down from 3. The new specialized algorithm is implemented in the following way

```
// forward substitution
f_tilde[1] = f[1];
for (int i = 2; i < n; i++){
    f_tilde[i] = f[i] + (f_tilde[i-1]/d[i-1]);
}

// backward substitution
u[n-1] = f_tilde[n-1]/d[n-1];        // setting the last term

for (int i = n-2; i > 0; i--){
    u[i] = (f_tilde[i] + u[i+1])/d[i];
}
```

where the total number of FLOPS have been reduced to $4(n-2)+1$ operation, which is half that of the general algorithm. For large numbers of n the running time can be said to go as $O(4n)$.

These algorithms are then ready to be used. We are however interested in finding out how much the computed numerical solution deviates from the analytic. The following equation gives us the relative error ϵ_i in the data set, where $i = 1, ..., n$

$$\epsilon_i = \log_{10}\left(\left|\frac{u_i - v_i}{v_i}\right|\right), \tag{15}$$

where u_i is the numerically computed solution and v_i is the exact analytic solution. The relative error ϵ_i can then be computed for different number of grid points.

3.2 LU-decomposition

Solving the linear algebra problem with an LU decomposition is relatively simple. By decomposing the matrix $\hat{\mathbf{A}}$ into the lower triangular matrix $\hat{\mathbf{L}}$ and the upper triangular matrix $\hat{\mathbf{U}}$, where all the diagonal elements in $\hat{\mathbf{L}}$ is 1, in such a way that $\hat{\mathbf{A}} = \hat{\mathbf{L}}\hat{\mathbf{U}}$. We can then rewrite the linear algebra problem into

$$\begin{aligned}
\hat{\mathbf{A}}\hat{\mathbf{u}} &= \hat{\mathbf{f}} \\
\hat{\mathbf{L}}\hat{\mathbf{U}}\hat{\mathbf{u}} &= \hat{\mathbf{f}} \\
\hat{\mathbf{U}}\hat{\mathbf{u}} &= \hat{\mathbf{L}}^{-1}\hat{\mathbf{f}} = \hat{\mathbf{y}} \\
\implies \hat{\mathbf{L}}\hat{\mathbf{y}} &= \hat{\mathbf{f}}\,,\ \hat{\mathbf{U}}\hat{\mathbf{u}} = \hat{\mathbf{y}}.
\end{aligned} \tag{16}$$

The problem is then to solve two equations, firstly for $\hat{\mathbf{y}}$ and lastly for $\hat{\mathbf{u}}$. Suppose the matrices have dimension $(n \times n)$. The solution can then be found by iterating n-times for each equation, to a total of $2n$ iterations. The Armadillo library solves this problem simply with the `solve`-function.

4. Results

All three algorithms are tested with varying precision. The produced results, figures and errors are presented here.

4.1. **The General Tridiagonal Algorithm.**

As previously mentioned, the general tridiagonal matrix algorithm can solve problems for matrices of $(n \times n)$. We have tested the algorithm for $n = 10$, $n = 100$, and $n = 1000$. The numerical solution is computed alongside the exact analytic solution so they can be compared.

The results are produced by compiling and executing the c++ program *problem_ b.cpp* with the commandline arguments 1 −1 2 −1. The first argument reads in number of grid points and is calculated by $n = 10^i$ where i is the commandline input. The next three arguments fills the diagonals of the tridiagonal matrix, which in this case becomes $a_i = c_i = -1$ and $b_i = 2$ which correspond to the discretized second derivative matrix. The result for the $n = 10$ can be seen in figure 1, $n = 100$ in figure 2, and $n = 1000$ in figure 3.

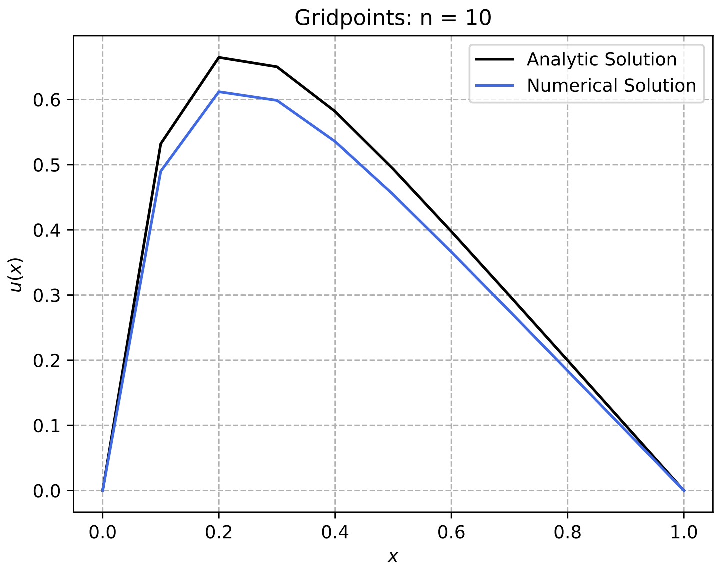

Figure 1: Numerical and analytic solution to Poisson's equation for $n = 10$ grid points.

In figure 1 we see that the both solutions have the same shape but the numerical one converges to $x = 1$ from below the analytic solution. The low number of grid points results in a non smooth curve for both the analytic and numerical solution. The Numerical solution ends up with a lower amplitude compared to the exact solution.

For $n = 100$, seen in figure 2, we see that the numerical solution lies nearly perfectly on top of the exact solution. For a simple 100 integration points, the algorithm seems to accurately reproduce the exact solution of the Poisson equation.

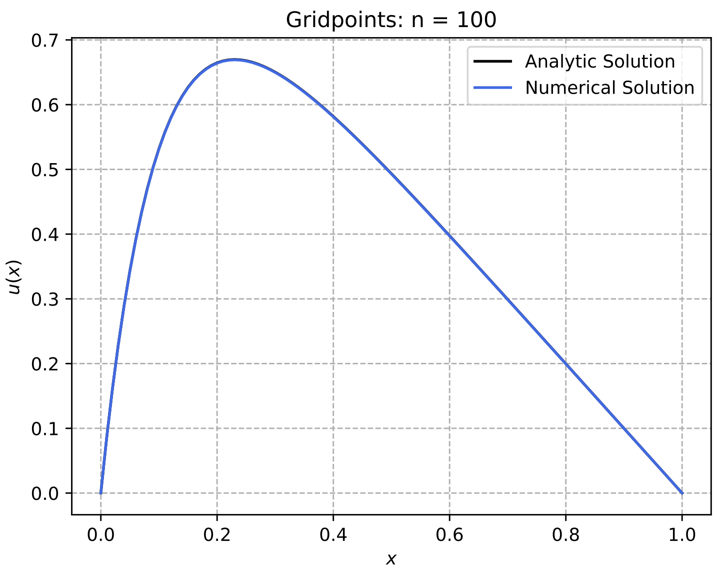

Figure 2: Numerical and analytic solution to Poisson's equation for $n = 100$ grid points.

Machine	General Algorithm	Special Algorithm
Old Dell (2012)	$1.1 \cdot 10^{-1}$ s	$6.8 \cdot 10^{-2}$ s
Macbook (2015)	$2.9 \cdot 10^{-2}$ s	$2.4 \cdot 10^{-2}$ s
Lenovo Y50-70 (2015)	$2.1 \cdot 10^{-2}$ s	$2.1 \cdot 10^{-2}$ s

Table 1: Average time usage for different machines with $n = 10^6$.

For the third and final precision test at $n = 1000$ seen in figure 3 the numerical solution is indistinguishable from the analytic one. These results suggest that our algorithm works as intended.

4.2. **Special Tridiagonal Algorithm.** The results produced by the special tridiagonal algorithm are the same as the once for the general algorithm, as was to be expected. The specialized algorithms only objective was to reduce the number of floating point operations and therefore cut the computation time of the calculation.

4.3 **Execution time.** The efficiency of the algorithms is compared by extracting the execution time for each of the algorithms separately. The execution time should be proportional to the number of FLOPS required in the algorithm. The following results has been found using a MacBook Pro (Early 2015) to run the program. The program has been executed 10 times in rapid succession collecting each run time. These are then used to find an average executing time.

For the tridiagonal algorithm, we find that that the specialized version has a faster run time than the general one. By taking the ratio of the time averages we

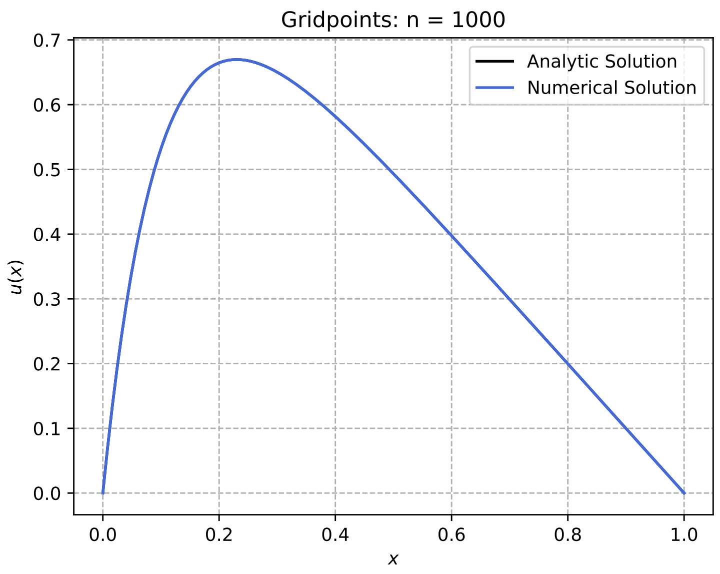

Figure 3: Numerical and analytic solution to Poisson's equation for $n = 1000$ grid points.

find that it the execution time was reduced by 20.16%, as seen in table 1. This is a marginal improvement in the algorithms efficiency. However, one would likely expect a run time reduction by as much as 50% as the number of FLOPS was halved in the specialized version of the algorithm. With an older Dell computer, this is seen more as expected, where the average time was reduced by almost 50%. The results from the Lenovo computer show no difference in run time.

4.4 **Relative Error.** Having computed the relative error for $n = 10, ..., 10^7$ grid points, we can extract the maximum value of the error for each set of grid points. Max$[\epsilon_i]$ is then plotted as a function if the grid points. The result is seen in figure 4.

We see that the relative error decreases linearly with increasing number of grid points, which is expected. However, once we reach $n > 10^6$ grid points, the error starts to increase again.

4.5 **LU-decomposition**

As described is section *3.2*, the `solve`-function is implemented in `problem_e.cpp`, with the number of columns and rows in the square matrix, n, given as a command line input. The runtimes for this code can be found in table 2. Generally speaking, we see clearly that the LU-decomposition has a longer computational time than the other algorithms to get the same results. Note also that the LU-solver would not run for $n = 10^5$, due to memory allocation error.

5. Discussion and Conclusion

The relative error in figure 4 is decreasing steadily to about $n = 10^6$, after which it starts increasing. In other words, the error in the calculations is at its

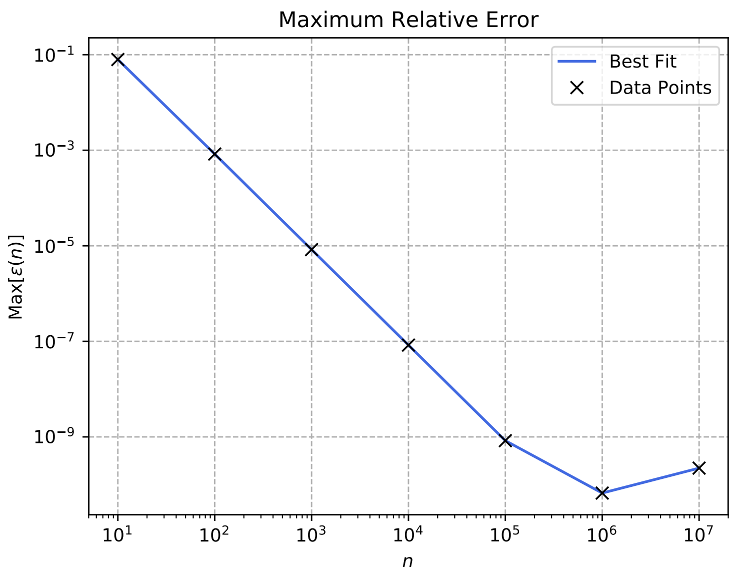

Figure 4: Maximum relative error as a function of grid points.

n	General Algorithm	Special Algorithm	LU-Decomposition
10^1	$1.2 \cdot 10^{-6}$ s	$1.2 \cdot 10^{-6}$ s	$4.4 \cdot 10^{-5}$ s
10^2	$3.1 \cdot 10^{-6}$ s	$2.9 \cdot 10^{-6}$ s	$8.8 \cdot 10^{-5}$ s
10^3	$2.2 \cdot 10^{-5}$ s	$2.2 \cdot 10^{-5}$ s	$5.2 \cdot 10^{-3}$ s
10^4	$2.2 \cdot 10^{-4}$ s	$2.3 \cdot 10^{-4}$ s	$5.1 \cdot 10^{-1}$ s
10^5	$2.2 \cdot 10^{-3}$ s	$2.2 \cdot 10^{-3}$ s	N/A
10^6	$2.1 \cdot 10^{-2}$ s	$2.1 \cdot 10^{-2}$ s	N/A

Table 2: Average time usage for 500 executions of the different algorithms, tested on Lenovo laptop running linux.

lowest at $n = 10^6$. The reason for the increase in error with n larger than this, might seem difficult to explain with an analytical mindset, but there is a lower limit to the precision of a number that a computer can represent. This means that that a larger n will actually make the computations more imprecise as the differences calculated becomes smaller than the machine precision. The ideal stepsize h is then $h = 1/n = 10^{-6}$.

The difference in run time between the specialized algorithm discussed in section 4.2 and the general algorithm discussed in section 4.1 seems to depend heavily on the computer doing the operations. As seen in table 2, the results from the Lenovo laptop has a minimal difference in rum time between the general and specialized algorithm, and the small differences there is, might be explained by random chance. The reason for this result is difficult to tell. As seen in table 1, the results differ when running on other machines. These values for computers with lower processing speeds have run times that are more as expected, with the specialized algorithm using noticeably shorter time. This makes more sense, as the specialized algorithm uses about half the FLOPS per iteration compared to the general algorithm. It might be natural to conclude that with higher speed processors, there are other factors that limit the lower time duration of the calculation, but this needs more research.

The results seen in section 4.5 and table 2 shows that the LU-decomposition is noticeably slower than the two other algorithms discussed here. This can be explained by the number of elements that are needed to compute the LU-decomposition. The more specialized only need to to operations with 3 vector of length n, while the solution with LU-decompositions needs to do operations on matrices with dimensions $n \times n$. The problem with this can be seen in table 2, where already at $n = 10^5$, the computer in use does not have enough memory to store all the numbers needed. On the other hand of this problem, the other algorithms exclusively solve a banded matrix, while the LU-decomposition can solve a more general linear algebra problem. So each method has its advantages and drawbacks.

References

[1] Hjorth-Jensen, M. (2018) *Overview of couse material: Computational Physics*, University of Oslo, URL: `https://compphysics.github.io/ComputationalPhysics/doc/web/course` (2018.09.10)