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

Markus Leira Asprusten Maren Rasmussen Metin San

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 $\hat{\mathbf{A}}\hat{\mathbf{u}} = \hat{\mathbf{f}}$. 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 $\hat{\mathbf{A}}$ 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 $\tilde{b_2} = b_2 - a_2 c_1 / b_1$, and $\tilde{f_2} = f_2 - f_1 a_2 / b_1$, and II and I denotes the row 1 and 2 in the $\hat{\mathbf{A}}$. 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 $\tilde{b_3} = b_3 - a_3 c_2 / b_2$, and $\tilde{f_3} = f_3 - f_2 a_3 / b_2$. 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 $\tilde{b_4} = b_4 - a_4 c_3 / b_3$, and $\tilde{f_4} = f_4 - f_3 a_4 / b_3$.

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.

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

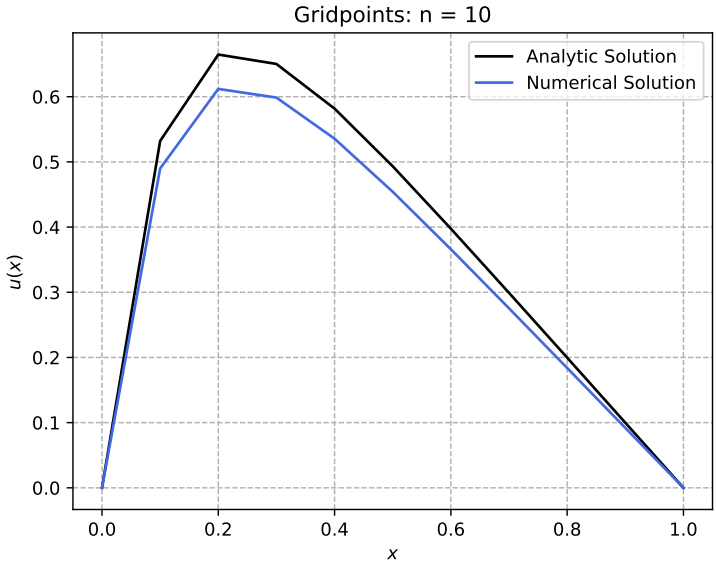

Figure 1: Numerical and analytic solution to Poisson's equation for $n = 10$ grid points.

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.

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.

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

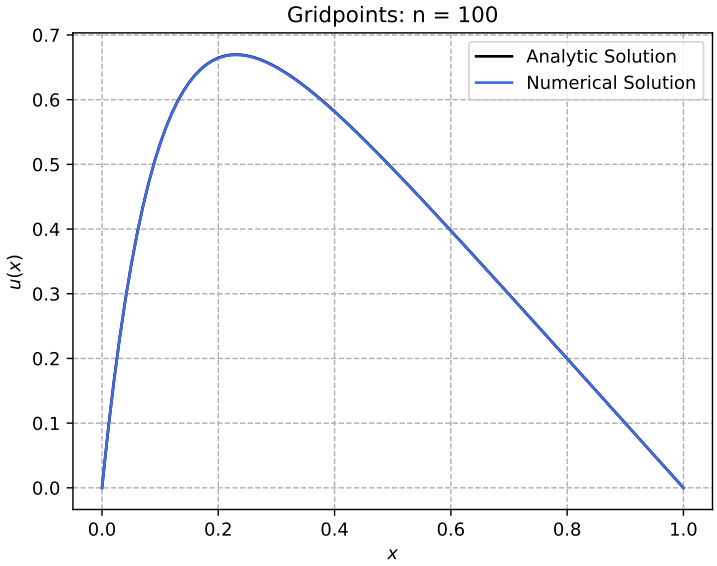

Figure 2: Numerical and analytic solution to Poisson's equation for $n = 100$ grid points.

each run time. These are then used to find an average executing time. The run time results can be seen in figure **??**.

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 find that it the execution time was reduced by 20.16%. 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.

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.

Conclusion / Discussion ellerno

The results presented in section ? shows that by only including the first terms of the Taylor expansion to approximate the second derivative, does not give a big error. At least not when using a big n. [...]

Section ? shows that even though we know from the mathematical theory that a bigger n and a smaller stepsize h shoud give better results, this is not the case nummerically. The reason for this, is that there is a limit on how accurate a

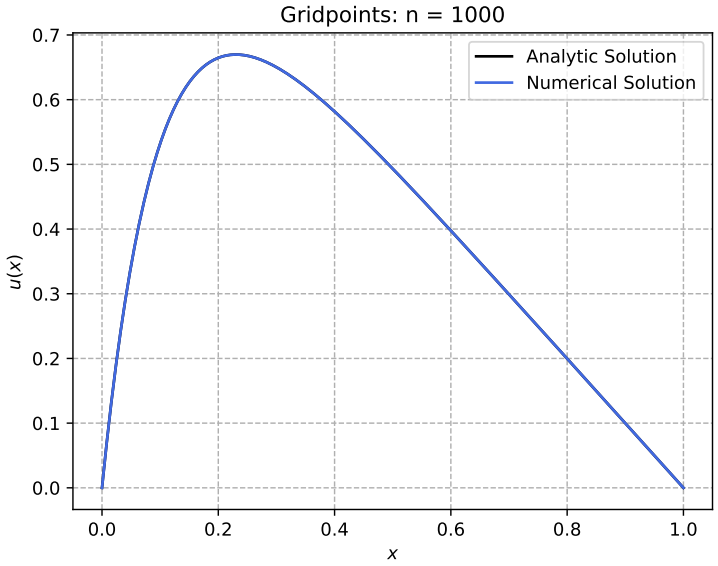

Figure 3: Numerical and analytic solution to Poisson's equation for $n = 1000$ grid points.

number can be represented on a computer. Beacuse of this limit, the computer has to make round offs, which increases the total error. This means that there has to be a ideal stepsize h, giving the best results. From figure ?, we found that this ideal stepsize was ??. [...]

From table ? we can see that it was not big difference in the run time between the general and the special algorithm. The special was only a bit faster, even thogh it had half as many flops as the general. But when we ran the program on a older computer with a worse processor we could see a much bigger difference between the two run times. On this computer the special algorithm was almost twice as fast as the general. This is also what we expected, since this algorithm had half as many FLOPS as the general.

Bare noen tanker, trenger ikke være med.

Punkter som bør diskuteres: (hvis vi skjønner hvorfor)
- Hvorfor numerisk kommer neden ifra (figure 1)
- Hvorfor run time er som den er
- Relative Error, hvorfor den begynner å øke
- Noe om LU

Andre ting som må fikses:
- Sette inn LU info i method, litt i resultater og diskusjon
- Skrive en bedre Abstract siden den bør skrives til slutt.
- Prøve å fikse posisjonen til figurene og bestemme hvilke/hvilken tabell med

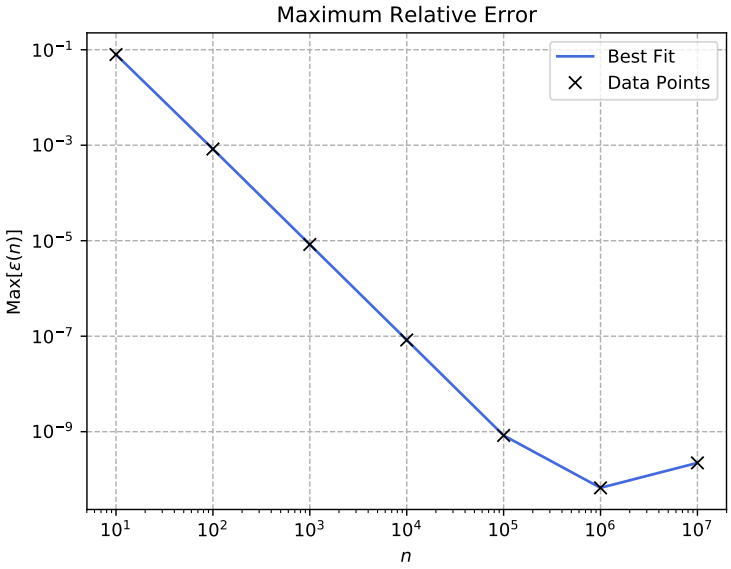

Figure 4: Maximum relative error as a function of grid points.

run times vi skal bruke.
- Finne en bedre tittel??

General Algorithm	Special Algorithm	LU-Decomposition
0.030738	0.023905	
0.028017	0.024106	
0.028026	0.024124	
0.028299	0.023557	
0.028742	0.024067	
0.028511	0.023537	
0.028791	0.02406	
0.028829	0.024148	
0.028296	0.023556	
0.028575	0.023642	

Average Time Spent	
0.0286824	0.0238702

General Algorithm	Special Algorithm
0.106575	0.066029
0.106026	0.066126
0.115654	0.066128
0.106315	0.068742
0.106237	0.067858
0.105891	0.067724
0.106279	0.068543
0.107324	0.068350
0.105830	0.067883
0.106344	0.068466

Average Time Spent	
0.1072475	0.068466