

# Project 1 in FYS4150

Eirik Ramsli Hauge

September 27, 2016

## Abstract

The aim of this numerical experiment is to solve the one-dimensional Poisson equation with Dirichlet boundary conditions. At first we will use the formula  $\mathbf{A}\mathbf{v} = \mathbf{p}$  and the fact that  $\mathbf{A}$  is a tridiagonal matrix to develop an algorithm that we then can compare to more general methods. We found that a specialized algorithm is better than a general and that using LU-decomposition is the slowest method. The relative error of the approximation compared to the exact solution is best at  $\log_{10}(h) = -5$  where  $h$  is the step length.

## Introduction

### Theory

In this project we want to look closer at the one-dimensional Poisson's equation given as:

$$\frac{1}{r^2} \frac{\partial}{\partial r} \left( r^2 \frac{\partial \Phi}{\partial r} \right) = -4\pi\rho(r) \quad (1)$$

where  $\Phi$  is the electrostatic potential generated by a localized charge distribution  $\rho(r)$ . If we now do the substitution  $\Phi(r) = \frac{\phi(r)}{r}$ , we can rewrite equation (1) as follows:

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

By letting  $\phi \rightarrow u$  and  $r \rightarrow x$  we end up with:

$$-u''(x) = f(x), \quad x \in (0, 1), \quad u(0) = u(1) = 0 \quad (2)$$

If we define the discretized approximation to  $u$  as  $v_i$ , the second derivative of  $u$  can be approximated with:

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

Where  $x_i = ih$  are grid points in the interval  $x_0 = 0$  to  $x_{n+1} = 1$  with step length  $h = 1/(n+1)$  and  $f_i = f(x_i)$ . With the boundary conditions  $v_0 = v_{n+1} = 0$  we can see that for  $i = 0$  we get:

$$-v_1 + 2v_0 = f_0 h^2$$

For  $i = 1$ :

$$-v_2 - v_0 + 2v_1 = f_1 h^2$$

For  $i = 2$

$$-v_3 - v_1 + 2v_2 = f_2 h^2$$

We can easily see that this gives us:

$$\underbrace{\begin{pmatrix} 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ \vdots \\ \vdots \\ v_n \end{pmatrix}}_{\mathbf{v}} = \underbrace{\begin{pmatrix} f_1 h^2 \\ f_2 h^2 \\ f_3 h^2 \\ \vdots \\ \vdots \\ \vdots \\ f_n h^2 \end{pmatrix}}_p$$

By setting  $f_i h^2 = p_i$  we can write this as:

$$\mathbf{A} \cdot \mathbf{v} = \mathbf{p}$$

The task **Referanse** asks us to make a general algorithm to solve this scenario for any values in the tridiagonal matrix. Assuming a general tridiagonal 4x4-matrix  $\tilde{\mathbf{A}}$  for simplicity we can illustrate the method for finding  $v$  as follows:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix}$$

This gives us the following equations:

$$\begin{aligned} \text{I} : v_1 b_1 + c_1 v_2 &= p_1 \\ \text{II} : a_2 v_1 + b_2 v_2 + c_2 v_3 &= p_2 \\ \text{III} : a_3 v_2 + b_3 v_3 + c_3 v_4 &= p_3 \\ \text{IV} : a_4 v_3 + b_4 v_4 &= p_4 \end{aligned}$$

We want only zeroes on the left side of the diagonal:

$$\begin{aligned} p_2^* &= p_2 - p_1 \cdot \frac{a_2}{b_1} \\ &= a_2 v_1 + b_2 v_2 + c_2 v_3 - v_1 a_2 - v_2 \frac{c_1 a_2}{b_1} \\ &= v_2 \left( b_2 - \frac{c_1 a_2}{b_1} \right) + c_2 v_3 \\ &= v_2 b_2^* + c_2 v_3 \end{aligned}$$

Now we have the following matrix:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ 0 & b_2^* & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{pmatrix}$$

As we can see, the  $a_2$ -term disappears from the second row. Following this trail of thought we do the same for the next row.

$$\begin{aligned} p_3^* &= p_3 - p_2 \cdot \frac{a_3}{b_2^*} \\ &= v_3(b_3 - \frac{c_2 a_3}{b_2^*}) + c_3 v_4 \\ &= v_3 b_3^* + c_3 v_4 \end{aligned}$$

This gives us the general idea and we can write a general expression for both  $p_i^*$  and  $b_i^*$ :

$$p_i^* = p_i - p_{i-1} \frac{a_i}{b_{i-1}^*}, \quad \text{for } i = 2, \dots, n, \quad \text{and } p_1^* = p_1 \quad (3)$$

$$b_i^* = b_i - \frac{c_{i-1} a_i}{b_{i-1}^*}, \quad \text{for } i = 2, \dots, n, \quad \text{and } b_1^* = b_1 \quad (4)$$

Using equations (4) and (3) through the whole matrix is called forward substitution and we end up with:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ 0 & b_2^* & c_2 & 0 \\ 0 & 0 & b_3^* & c_3 \\ 0 & 0 & 0 & b_4^* \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2^* \\ p_3^* \\ p_4^* \end{pmatrix}$$

We want to find an expression for  $v_i$  and from the last row we can find a simple equation for  $v_4$

$$b_4^* v_4 = p_4^* \Rightarrow v_4 = \frac{p_4^*}{b_4^*}$$

From the second last row we find an expression for  $v_3$

$$b_3^* v_3 + c_3 v_4 = p_3^* \Rightarrow v_3 = \frac{p_3^* - c_3 v_4}{b_3^*}$$

Again, doing this for the next row (going downward and up) we find that  $v_i$  can be expressed in a general way as:

$$v_i = \frac{p_i^* - c_i v_{i+1}}{b_i^*}, \quad \text{for } i = n-1, \dots, 1, \quad \text{and } v_n = \frac{p_n^*}{b_n^*} \quad (5)$$

This is called backwards substitution.

However, since matrix  $\mathbf{A}$  has the same values for a, b and c for all  $i$ , we can specialize equations (3), (4) and (5). By inserting  $a_i = c_i = -1$  and  $b_i = 2$  in (4) we can easily see that:

$$b_i^* = \frac{i+1}{i}, \quad \text{for } i = 2, \dots, n, \quad \text{and } b_1^* = b_1 = 2 \quad (6)$$

$$p_i^* = p_i + \frac{p_{i-1}^*}{b_{i-1}^*}, \quad \text{for } i = 2, \dots, n, \quad \text{and } p_1^* = p_1 \quad (7)$$

$$v_i = \frac{p_i^* + v_{i+1}}{b_i^*}, \quad \text{for } i = n-1, \dots, 1, \quad \text{and } v_n = \frac{p_n^*}{b_n^*} \quad (8)$$

Since our  $v_i$  is an approximation to the known solution  $u_i$  we can find the relative error by:

$$\epsilon_i = \log_{10} \left( \left| \frac{v_i - u_i}{u_i} \right| \right) \quad (9)$$

where the known solution  $u_i$  is given as:

$$u_i = u(x) = 1 - (1 - e^{-10})x - e^{-10x} \quad (10)$$

## LU-decompostion

Another method that can be used is the method of LU-decompostion. If the determinant of a matrix  $\mathbf{A}$  is different from zero, we can factorize  $\mathbf{A}$  into a lower diagonal matrix  $\mathbf{L}$  and an upper diagonal matrix  $\mathbf{U}$ :

$$\mathbf{A} = \mathbf{LU}$$

$$\begin{pmatrix} a_{1,1} & \cdots & \cdots & a_{1,n} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{n,1} & \cdots & \cdots & a_{n,n} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ l_{n,1} & \cdots & l_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & u_{n,n} \end{pmatrix}$$

Using this in our case we get:

$$\mathbf{LU}\mathbf{v} = \mathbf{p}$$

Rewriting this we get:

$$\mathbf{Ly} = \mathbf{p}, \quad \mathbf{Uv} = \mathbf{y}$$

This is solvable by using backwards substitution to find  $\mathbf{y}$  and use that to find  $\mathbf{v}$ . Usually, for LU-decomposition and backwards substitution, solving it goes as  $\mathcal{O}(\frac{2}{3}n^3)$ .

## Programs

In this section I will present the different key parts of my program. For the full program, please visit my github.

## General tridagonal solver

Implementing the equations for forward substitution, (3) and (4), were done as follows:

---

**Algorithm 1** Forward substitution

---

```
1:  $b_1^* = b_1$ 
2:  $p_1^* = p_1$ 
3: for  $i = 2, n$  do
4:    $b_i^* = b_i - a_{i-1} \cdot c_{i-1} / b_{i-1}^*$ 
5:    $p_i^* = p_i - p_{i-1}^* \cdot a_i / b_{i-1}^*$ 
6: end for
```

---

The backward substitution given in equation (5) gives us:

---

**Algorithm 2** Backward substitution

---

```
1:  $v_n = p_n^* / b_n^*$ 
2: for  $i = n - 1, 1$  do
3:    $v_i = (p_i^* - c_i \cdot v_{i+1}) / b_i^*$ 
4: end for
```

---

Counting number of floating point operations gives us 6 for forward substitution and 3 for backward. This gives us  $9n$  FLOPS in total.

## Specialized tridagonal solver

Since all a and c-values are -1 and all b-values are 2 throughout the whole matrix, we don't need the general algorithm, but can use a more specialized algorithm. Our specialized algorithm is based on equation (7) and (8) (equation (6) was calculated beforehand) and was implemented as follows:

---

**Algorithm 3** Specialized algorithm

---

```
1:  $p_1^* = p_1$ 
2: for  $i = 2, n$  do ▷ Forward substitution
3:    $p_i^* = p_i + p_{i-1}^* / d_{i-1}$ 
4: end for
5:  $v_n = p_n^* / d_n$ 
6: for  $i = n - 1, 1$  do ▷ Backwards substitution
7:    $v_i = (p_i^* + v_{i+1}) / d_i$ 
8: end for
```

---

This method gives us  $4n$  FLOPS.

## Results

To see if our approximation with forward and backward substitution was right, we plotted it against the known solution given in (10). For a starter we set the N-value to 10, this

gave us 10 different x-values. The result where we compare the known and approximated solution is shown in 1.

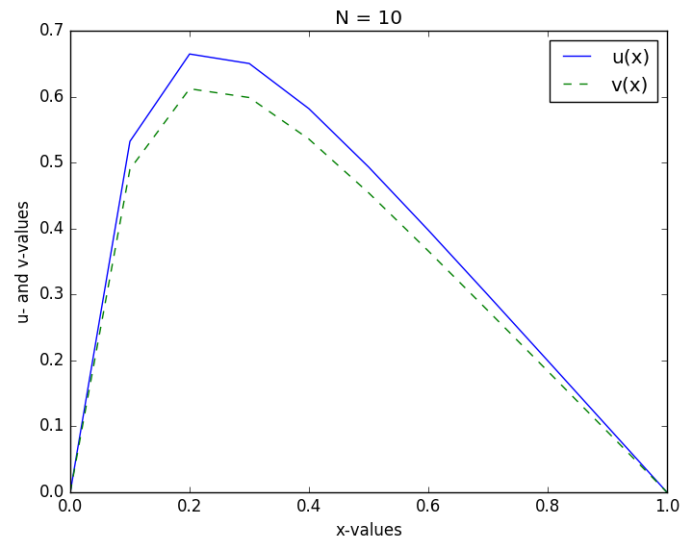
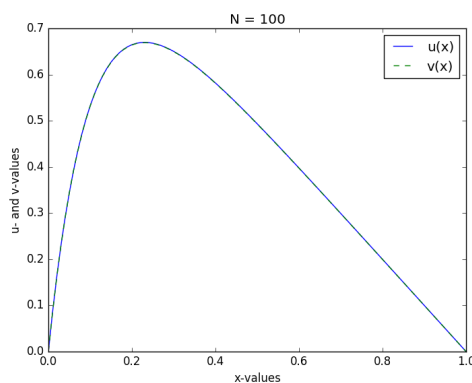
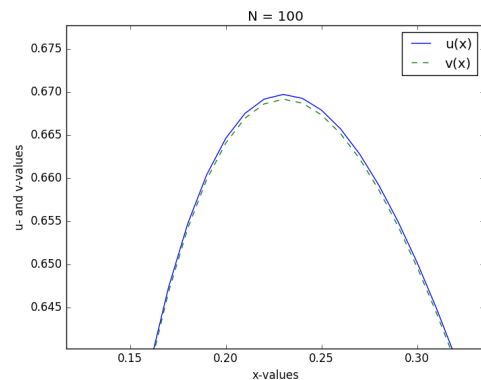


Figure 1: Our first plot where we have  $N = 10$  points.  $u(x)$  is the correct solution and  $v(x)$  is the approximated. As we can see that they almost align, but not quite. Especially in the top around  $x = 0.2$ .

Afterwards we set the  $N$ -value to 100 and got 100 points. The result of this can be seen in 2. As we can see from 2a and 2b we have to zoom in to see any difference. We also plotted for higher values of  $N$ , but the results were much the same as for  $N = 100$  only we had to zoom in even more to see the difference.



(a) The  $N = 100$  case



(b) The  $N = 100$  case, zoomed in.

Figure 2: The plot for  $N = 100$ .  $u(x)$  and  $v(x)$  are the same as above. As we can see the graphs are much more aligned now. On the left we have zoomed in on the top to show the small differences.

To see the effectiveness of our specialized algorithm we took the time of the general, the

specialized algorithm and the LU-decompositionmethod. The result is presented in table 1.

$\log_{10}(n)$	General [sec]	Specialized [sec]	LU-decomposition [sec]
1	$2.0 \cdot 10^{-6}$	$1.0 \cdot 10^{-6}$	$8.0 \cdot 10^{-6}$
2	$4.0 \cdot 10^{-6}$	$2.0 \cdot 10^{-6}$	$4.3 \cdot 10^{-3}$
3	$5.3 \cdot 10^{-5}$	$2.7 \cdot 10^{-5}$	2.2
4	$6.0 \cdot 10^{-4}$	$4.2 \cdot 10^{-4}$	-
5	$3.8 \cdot 10^{-3}$	$2.2 \cdot 10^{-3}$	-
6	$4.2 \cdot 10^{-2}$	$2.4 \cdot 10^{-2}$	-
7	0.43	0.26	-

Table 1: This table shows us the time used by the program for different N-values. The last column contains only 3 values because for  $N = 10000$  and above the calculations took too long.

Lastly we found the relative error using (9) and the result is presented in 2.

$\log_{10}(h)$	Relative error
-1	-1.1
-2	-3.1
-3	-5.1
-4	-7.1
-5	-9.0
-6	-6.8
-7	-13.5

Table 2: The relative error between our approximation and the exact solution as a function of step length  $h$ .

## Discussion

Comparing our three methods we can see that the specialized algorithm is superior when it comes to time usage. From table 1 we can see that there is about a factor of two between the general and the specialized algorithm. This fits our estimation of respectively  $9n$  and  $4n$  FLOPS. We can also see that the LU-decomposition is clearly inferior. This is because the technique calculates with a whole matrix, while our method with forward and backward substitution only uses the three diagonals that have any value other than zero in them. In our case with such small N-values one can debate if it really was necessary to make a specialized algorithm when the general method only used 0.43 seconds with  $n = 10^7$ . It is a fair point, however, if we had continued with higher values of  $n$  until the program used about half an hour completing the task, I would prefer the method that only used 15 minutes. With this argument we meet another problem. If we continued to high enough values of  $n$ , we would have overflow and then the program would not work even if it only used half the time to do it.

If i had done this assignment again or need the program in the future, I would want to

look over it with fresh eyes and try to make it more effective. I would have liked to find another way or library to do the LU-decomposition. Even if it can't be as effective in the tridiagonal case, it is always useful to have a good way to manipulate whole matrixes at once.

From figures 1 and 2 we can clearly see that, for these small values, increasing  $n$  will give a better approximation. This is also confirmed by 2 which shows us that the relative error decreases until we have  $\log_{10}(h) = -5$ . After that it becomes more and then it is smaller again. Why we have this sudden peak at  $\log_{10}(h) = -6$  is unclear to me. It may be round of errors or it may be that this exact number of steps gives an uncertainty not present elsewhere. If we increase the step length even further we can see that for  $\log_{10}(h) = -8$ , we have a relative error at about -2, going higher than that makes my computer crash. This could mean two things: Either we have an anomaly at  $\log_{10}(h) = -6$  and the relative error should be lower or at  $\log_{10}(h) = -8$  and the relative error should be higher. It would be natural to assume that  $\log_{10}(h) = -5$  is the optimal and that afterwards the relative error becomes worse and worse. If I were to do the project again, I would look more into this and run the program for smaller step lengths on a computer with more RAM.

## Conclusion

At the end of the day it is clear to us that the specialized algorithm uses the shortest time and is thus the optimal way to solve this specific task. The approximation becomes better and better until we reach step lengths as small as  $\log_{10}(h) = -5$ . Even though  $\log_{10}(h) = -8$  gives a better relative error, this could be because of something wrong in the program and should be double checked before it is trusted. The LU-decomposition is a slow method because it uses the whole matrix and does not take advantage of the properties of a tridiagonal matrix.