Project 1: Computational Physics - FYS3150

Fredrik Hoftun & Mikkel Metzsch Jensen

September 10, 2020

Abstract

In this project we have looked at three different numerical algorithms for the solving of the one-dimensional Poisson equation with Dirichlet boundary conditions in matrix form. The first two algorithm were based on Gaussian elimination with a forward and backward substitution. The first one were a more general algorithm while the second one was more speliazed for our exact problem. The third algorithm was a LU decomposition of the matrix. We found that the precision of the algorithm increased as $\mathcal{O}(h^2)$ with step size $h \geq 10^{-4}$. At smaller stepsizes the round-off error would dominate giving os best precision for the general algorithm at $h = 10^{-5}$ and the special algorithm at $h \geq 10^{-6}$. We found that the first two algorithms performed similarly regarding CPU time with the special algorithm being 1.5 faster at best. The LU decomposition used considerably more time. This did somewhat reflect on the calculated number of floating point operations for each algorithm which were which were $\mathcal{O}(9n)$, $\mathcal{O}(4n)$ and $\mathcal{O}(2/3\, n^3)$ for general, special, and LU decomposition respectively.

Link to our GitHub repository: `https://github.com/mikkelme/Project_1_FYS3150`

Contents

1 Introduction

In this project we will investegate different algorithms for the solving of the one-dimensional Poisson equation with Dirichlet boundary conditions. This is a problem that is used in many different scientiffic applications. It is used to describe electrostatic and megneto-static phenonema in a quantitative maner, and it is also helps to understand diffusion and propagation problems. The solution is generally used in a wide range of fields such as engineering, physics, mathematics, biology, chemistry, etc. [1]. In this project we have used a second derivative approximation in order to rewrite the problem as a set of linear equations. We have then written three different algorithms: A general and a specialiazed algorithm using gaussian elimination and then one using LU demposition. Theese are described individually in the method sections along with a calculation of the number of floating point operation for each of them. Since we know the analytical solution for our problem we have calculated the relative error for different choices of stepsize h in the algorithm. We have also timed the execution of the algorithms and we compare CPU time of theese computations. We found a significant difference in both precesion and CPU time, which are presented in the results section.

2 Method

2.1 Defining the problem

We are going to solve the one-dimensional Poisson equation with Dirichlet boundary conditions given as follows:

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

In our case we will use the function:

$$f(x) = 100e^{-10x}$$

Note that the algorithms for solving the problem will not be depedent on this specefic choice of f(x). The reason why we stick to this function is for the praticality of having an analytical solution $u(x)$ for f(x):

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

We can ensure that this is a valid Solution by inserting it into the Poission equation. First we calculate the double derivative of u(x):

$$u'(x) = -(1 - e^{-10}) + 10e^{-10x}, \quad u''(x) = -100e^{-10x}$$

We now see that the solution satisfies the Poisson equation:

$$-u''(x) = 100e^{-10x} = f(x)$$

2.2 Rewritting the problem as a set of linear equations

In order to solve the Poisson equation numerically we discretize u as v_i with $n+2$ grid points $x_i = ih$ for $i = 0, 1, \dots, n+1$. To clarify this gives us a spacing with step length $h = 1/(n+1)$ and end points $x_0 = 0$, $x_{n+1} = 1$. The boundary conditions is given as $v_0 = v_{n+1} = 0$. We use the following second derivative approximation:

$$-u''(x_i) \approx -\frac{v_{i+1}+2v_i-v_{i+1}}{h^2} = f(x_i) \quad \text{for } i = 1,\dots,n$$

$\iff$

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f(x_i)$$

Notice that we cannot calculate the second derivative approximation at the end points $i = 0$ and $i = n+1$ since we need to have avaliable points $v_{i\pm1}$ on either side of v_i for the calculation. We define the colum vector $\mathbf{v} = [v_1, v_2, \dots, v_n]$ and try to write out the equation for every step $i = 1,\dots,n$. As we do this we see a usefull pattern appearing:

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = h^2 f(x_1)$$

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} f(x_1) \\ f(x_2) \end{bmatrix}$$

$$\vdots$$

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f_n \end{bmatrix}$$

From this we see that we can write the problem as a set of linear equation using a matrix $\mathbf{A}$:

$$\mathbf{Av} = \mathbf{g}$$

With the following definitions:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, \tilde{\mathbf{g}} = h^2 \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f_n \end{bmatrix}$$

2.3 General solution using Gausian elimination

We can solve our set of linear equations using Gaussian elimination on the matrix $\mathbf{Av} = \mathbf{g}$. In the beginning we assume a more generilazed matrix $\mathbf{A}$ as:

$$A = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_1 & b_2 & c_2 & 0 & \cdots & \cdots \\ 0 & a_2 & b_3 & c_3 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \cdots & & 0 & a_{n-1} & b_n \end{bmatrix}$$

Here a_i are the elements below the diagonal, b_i are the elements on the diagonal and c_i are the elements above the diagonal. In order to solve this we use first a forward substitution and then a backwards substitution [2]. The implementation of this is showed in Algorithm 1:

Algorithm 1 General algorithm

```
1: for $i = 2, \ldots, n$ do                          ▷ Forward substitution eliminating $a_i$
2:     $b_i = b_i - c_{i-1} \cdot a_i / b_{i-1}$                          ▷ Update $b_i$
3:     $g_i = g_i - g_{i-1} \cdot a_i / b_{i-1}$                          ▷ Update $g_i$
4: end for

5: $v_0 = v_{n+1} = 0$                                ▷ Backward substitution obtaining $v_i$
6: $v_n = g_n / B_n;$
7: for $i = n-1, \ldots, 1$ do
8:     $v_i = (g_i - c_i \cdot v_{i+1}) / b_i$
9: end for
```

By running this algorithm for decreasing step length we se that the numerical solution aproaches the analytical quite fast (see figure 1)

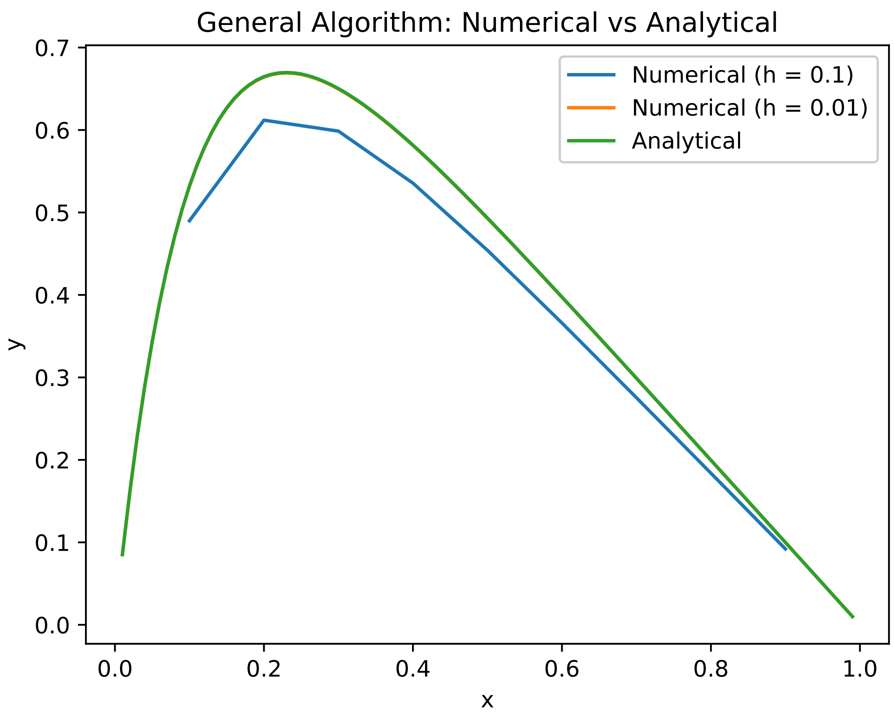

Figure 1: The figure shows the analytical solution and the numerical solution using the generel algorithm for $h = 0.1$ and $h = 0.01$ respectively. We see that the numerical solution lines up with the analytical solution already for $h = 0.01$.

We can calculate the algorithms number of Floating Point Operations. In our forward substitution we have $2 \cdot 3$ operations for each loop which runs for a total of $n-1$ times. In the backward substitution we have a single leading operation and then 3 operations which also loops for a total of $n-1$. The total number of the operations is then

$$(6+3)(n-1)+1 = 9n-8$$

For large n we can approximate this as 9n.

2.4 Simplified specific solution

Since we actually have the fixed values $a_i = -1,\ b_i = 2,\ c_i = -1$ for all i for the matrix $\mathbf{A}$, we can simplify our algorithm even more. Now we can do the forward substitution before hand:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix} \sim \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 3/2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix}$$

$$\sim \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 3/2 & -1 & 0 & \cdots & \cdots \\ 0 & 0 & 4/3 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix} \sim \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 3/2 & -1 & 0 & \cdots & \cdots \\ 0 & 0 & 4/3 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & 0 & i_n/i_n - 1 & -1 \\ 0 & \cdots & & 0 & 0 & i_n + 1/i_n \end{bmatrix}$$

We see that the updated diagonal element b_i follows the formula:

$$b_i = \frac{i+1}{i}$$

This means that the update of the diagonal element now use 2 floating point operations per n (1 less than before), but more important this can be precalculated and in practice excluded from the algorithm. We can therefore justify not to count theese floating point operations. The update of g_i with known values for a, b and c simplifies to:

$$g_i = g_i + g_{i-1}/b_{i-1}$$

[2]. This gives us the following special algorithm:

Algorithm 2 Special algorithm, where $a_i = -1,\ b_i = 2,\ c_i = -1$

```
1:  for $i = 1, \ldots, n$ do                    ▷ Pre-calculating $b_i$
2:      $b_i = i + 1/i$
3:  end for

4:  for $i = 2, \ldots, n$ do                    ▷ Forward substitution eliminating $a_i$
5:      $g_i = g_i + g_{i-1}/b_{i-1}$            ▷ Update $g_i$
6:  end for

7:  $v_n = 0$                                    ▷ Backward substitution obtaining $v_i$
8:  for $i = n - 1, \ldots, 1$ do
9:      $v_i = \frac{g_i + v_{i+1}}{b_i}$
10: end for
```

Similarly to the general algorithm we can calculate the total number of floating point operations (not including calculation of b_i):

$$(2+2)(n-2) = 4n - 8$$

For large n this can be approximated to $4n$. When running the script with this algorithm we see indically results for $h = 0.1$ and $h = 0.01$ (see figure 2) as we did with the gerneral algorithm.

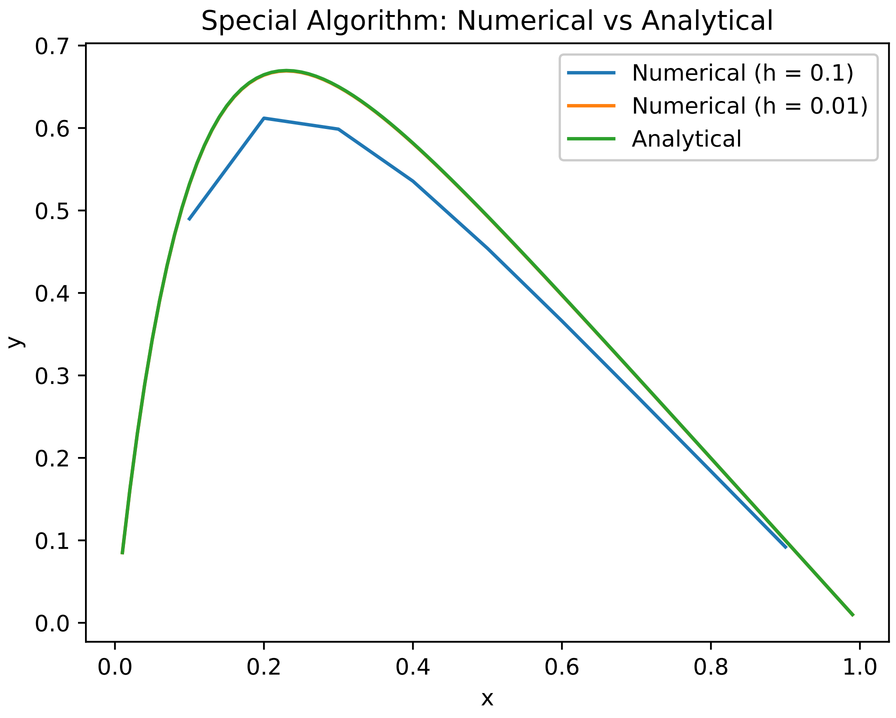

Figure 2: The figure shows the analytical solution and the numerical solution using the special algorithm for $h = 0.1$ and $h = 0.01$ respectively. We see that the numerical solution lines up with the analytical solution already for $h = 0.01$.

2.5 LU decomposition

For the LU decomposition we have our triangular matrix $\mathbf{A}$ which can be divided into two invertible matrices, consisting of one upper triangular matrix $\mathbf{U}$ and one lower triangular matrix $\mathbf{L}$. We then get the LU decomposition:

$$\mathbf{Av} = \mathbf{g} = \mathbf{LUv}$$
$$\mathbf{Uv} = \mathbf{L}^{-1}\mathbf{g} = \mathbf{w}$$

The Armadillo [3] library for `C++` allows for easy computation of the matrices.
The LU decomposition uses $\mathcal{O}(2/3\ n^3)$ FLOPS for the computation, making it much heavier than the general and special algorithms.

Proof that L or U is invertible

A matrix is invertible if the determinant is not equal to zero; $\det A \neq 0$. A property of triangular matrices is that the determinant is the product of the values on the diagonal. In our case we have non-zero real numbers so the matrices are invertible.

Memory use

Since we need to define the whole matrix when using LU decompoistion it becomes more heavy one the Random Acces Memory (RAM). When defining a $n \times n$ matrix with double precision (64 bits) we use a total amount of memory storage:

$$\text{memory needed} = n^2 \cdot 64 \text{ bits} = n^2 \cdot 8 \text{ bytes}$$

We see that the memory needed grows quite fast when increasing n by 10:

Table 1: Memory (RAM) needed for a $n \times n$ matrix

n	Memory needed
10^1	800 B
10^2	80 kB
10^3	8 mB
10^4	800 mB
10^5	80 GB

The machine that we are using have 4 GB of RAM and therefore we cannot initialize a $10^5 \times 10^5$ matrix.

2.6 Comparing precision and time

In order to compare the precesion of the different algorithms we will use the relative error between the numerical solution (v_i) and the analytical solution ($f(x_i)$). For practical reasons we will use the base 10 logarithm so the relative error is given as

$$\epsilon_i = log_{10} \left| \frac{v_i - u_i}{u_i} \right|$$

Since we divide with the analytical solution u_i in this expression we cannot make the calculation for the end points where $u_0 = u_{n+1} = 0$. This means that we will only calculate ϵ_i for $i = 1, \ldots, n$. When we lower the step length h we expect the error to decrease as well. When plotting h with a logaritmic axis (base 10) against ϵ_i we should see a linear trend. The slope of this trend can then be usefull in order determain the relationship between step length and precesion. At some point though we could expect this trend to interrupted by erros due round-off. We want to investegate if and when this happens.
In addition we will also investegate whether we see any different in CPU time between the different algorithms. We expect that the timing will reflect back on the number of floating point operations done in each algorithm.

2.7 Implementation

We used `C++` to implement our algorithms. The results from `C++` scripts a written to text file and then manage in python for plotting and presentation of the results.

3 Results

We collected the maximum of the relative error ϵ for all the three algorithms for different values for the step length h. The result is shown in figure 3.

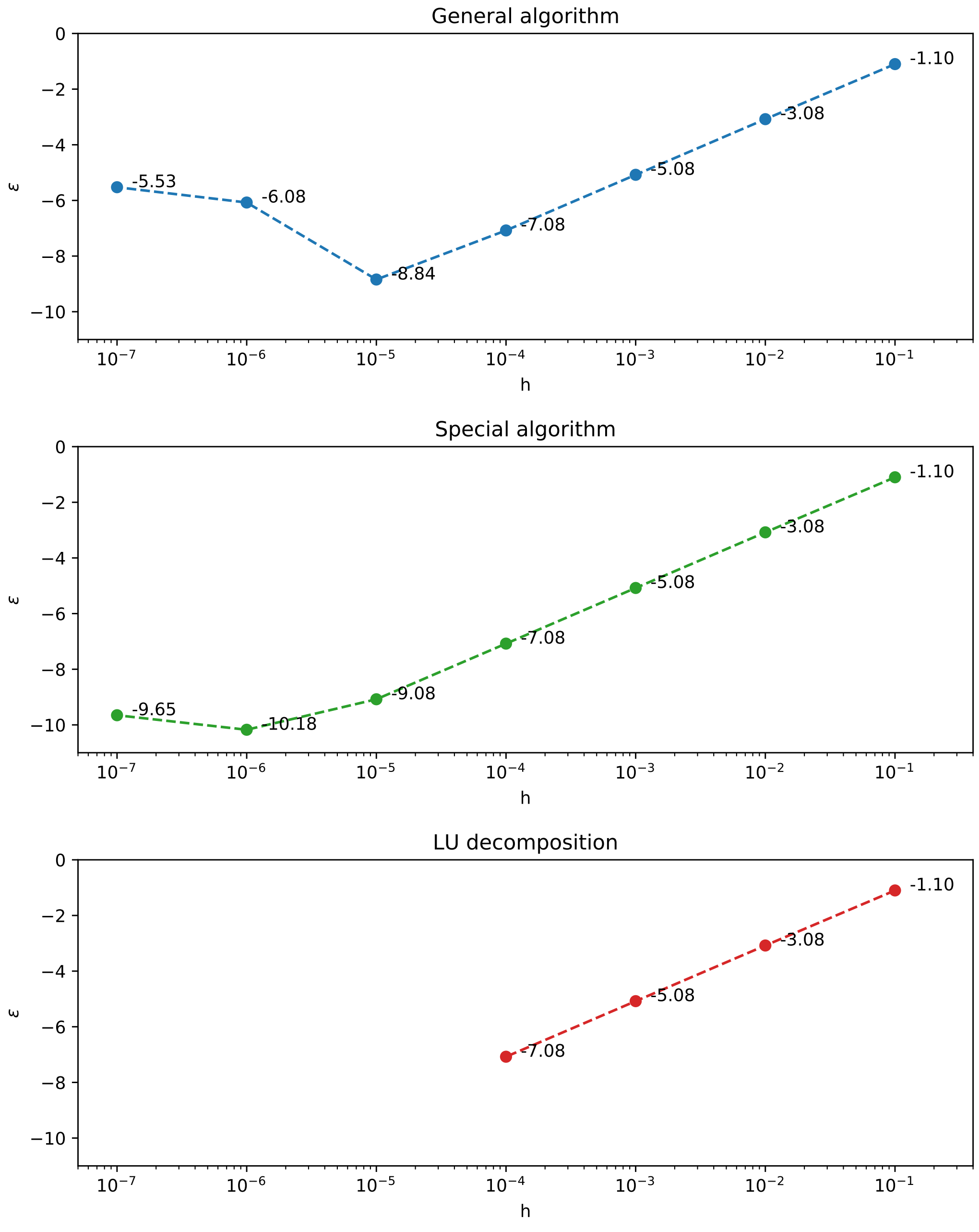

Figure 3: Relative error ϵ for different step length h for the three diferent algorithms. Note that relative error is defined with a logarithm and h is also plotted on a logaritmic axis.

Note that we were not able to produce the results for LU decompositions with $h \leq 10^{-5}$ because we did not have enough memory to store a matrix of size $10^5 \times 10^5$ or larger.

The CPU time for the execution of the different runs can be seen in table 2. Note that this varied a lot between each repetition of measurements, and it might therefore not be very reliable for small comparisons.

Table 2: CPU Time for the execution of the algorithms for diffrent choices of n, where n is the number of steps. Recall that we have $n \propto h^{-1}$

n	**General algorithm: t [s]**	**Special algorithm: t [s]**	**LU Decomposition: t [s]**
10^1	3×10^{-6}	2×10^{-6}	9.81×10^{-4}
10^2	4×10^{-6}	4×10^{-6}	1.96×10^{-4}
10^3	3.8×10^{-5}	3.2×10^{-5}	1.02×10^{-2}
10^4	3.41×10^{-4}	3.42×10^{-4}	2.45
10^5	3.79×10^{-3}	2.35×10^{-3}	nan
10^6	3.57×10^{-2}	2.38×10^{-2}	nan
10^7	3.16×10^{-1}	2.26×10^{-1}	nan

The computer which we ran all the programs have the following specs:

$$\text{CPU} = 2.5 \text{ GHz Intel Core i5}$$
$$\text{RAM} = 4 \text{ GB 1333 MHz DDR3}$$

4 Discussion

Regarding the development of the maximum relative error (showed in figure 3) we saw the expected trend that we were looking for. For all the algorithms we got a linear connection between $log_{10}(h)$ and ϵ for $h \geq 10^{-4}$. In this interval ϵ drops about 2 points for each logarithmic decreasement in h. This tell us that the error is proportional to h^2:

$$\epsilon = \mathcal{O}(h^2)$$

From the results we also see that the linear trend is broken at some point given us a point of minimum error. Since we were not able to go lower than $h = -4$ for the LU decomposition we can only compare the general and special algorithm here. In both theese cases the linear trend is broken around $h = -5$. For the general algorithm this happens quite drastically leaving us with a minimum relative error of -8.84. For the special algorithm the error keeps decreasing all the way down to -10.18 at $h = -6$. This loss of precision happens due to round-off erros when we are handling too many small calculations. In the case of the special algorithm we see that the simplified algorithm with fewer calculations make us able to archieve higher precesion. If we had to extract the most reliable solution from this problem we should choose to run the special algorithm with $h \approx -6$. This might not be the precise minimum point, and we could still vary h in even smaller increaments to check for lower relative errors.
Regarding the CPU time we got some mixed results. First of all we expected that the

number of floating point operations would be proportional to the CPU time. Recall that we found the following relation for the floating point operations:

$$\text{General algorithm} = \mathcal{O}(9n)$$
$$\text{Special algorithm} = \mathcal{O}(4n)$$
$$\text{LU decompoistion} = \mathcal{O}(2/3\ n^3)$$

We see that increasements in n by the order of 10 also produces increasements in CPU time for the general end special algorithm by 10, just as expected. This does not hold true for the transistion from $n = 10$ to $n = 100$ but we will se past this since the short computation time might make these results more uncertain here. For the LU decomposition we expected to se an increasement in CPU time with a factor of $e3$ for each time, but we saw it increase with a factor of $e2$. We later found that the armadillo solver might have used a more clever approach to this problem. On the documentation it says that the solver can use different solving techniques if it recognize that the problem can be simplified: "By default, matrix A is analysed to automatically determine whether it is a general matrix, band matrix, diagonal matrix, or symmetric/hermitian positive definite (SPD) matrix; based on the detected matrix structure, a specialised solver is used for faster execution" [3]. Therefore we can not use this a reliable example of LU decompoistion. Regarding the difference CPU time between the general and special algorithm we expected the time difference to be $\frac{9n}{4n} = 2.25$. The results showed that the special algorithm was somewhat faster at large n. For n = 10^6 we had a difference of:

$$\frac{3.57 \times 10^{-2}}{2.38 \times 10^{-2}} = 1.5$$

This is not quite as fast as expected though. We do not have any certain explanation for this, but we cannot make any conclusion without having a mean timing for mutiple runs. Anyway by using the calculated number of operations for each algorithm and the timing we can check whether this comes close the number of FLOPS (Floating Point Operations Per Second) that the CPU should be able to perfom. This computer should be able to make a maximum of $2.5e9$ FLOPS. We can inevestegate the number of FLOPS for the general algorithm with $n = 10^7$:

$$FP = 9 \times 10^7 - 17 \approx 9 \times 10^7$$

$$\text{CPU Time} = 0.316\ s$$

$$\text{FLOPS = FP / CPU Time} \approx 0.28\ GHz$$

We see that there are some room up the maximum capacity of the processor (2.5 GHz), but this might be somewhat expectable. Remember that we did not take any other operation into account, such as writting and reading from memory.

5 Conclusion

In this report we have used three different ways of solving the one-dimensional Poisson equation with Dirichlet boundary conditions in matrix form and have seen that efficiency of the methods vary greatly. First of all we saw that precesion of the algorithms increased as $\mathcal{O}(h^2)$ with step size $h \geq 10^{-4}$. For smaller stepsizes we found that the precesion was affected by round-off errors. This lead to the general algorithm having the best precision at $h = 10^{-5}$ and the special algorithm having the best precision at $h \geq 10^{-6}$. We were not able to make this kind of conclusions on the LU decompoistion method because of memory lack. The general and special algorithms were comparable in computation time with the special algorithm being around 1.5 time faster than the general, while the LU decomposition lagged greatly behind. This were somewhat relatable to the number of floating point operations for each algorithms which were $\mathcal{O}(9n)$, $\mathcal{O}(4n)$ and $\mathcal{O}(2/3\ n^3)$ for general, special, and LU decomposition respectively. We have learned the importance of choosing the correct algorithm, and that specialising it will make for some more efficient and accurate code.

References

[1] S. B. Gueye, K. Talla and C. Mbow, "Solution of 1d poisson equation with neumann-dirichlet and dirichlet-neumann boundary conditions using the finite difference method", Journal of Electromagnetic Analysis and Applications, Vol. 6, No. 10, pp. 309, 2014.

[2] Hjort-Jensen, M., 2018. Computational Physics Lectures: Linear Algebra methods, accesable at course github repository. `http://compphysics.github.io/ComputationalPhysics/doc/pub/linalg/pdf/linalg-print.pdf`

[3] Armadillo's website: `http://arma.sourceforge.net/`

Appendix

GitHub repository: `https://github.com/mikkelme/Project_1_FYS3150`