# FYS3150 - Project 1

Sigurd Sørlie Rustad and Vegard Falmår
(Dated: September 28, 2020)

We have explored three different numerical implementations of a finite difference method for solving the one-dimensional Poisson equation. This is a second order partial differential equation. The main goal of the project was to observe how the relative error relates to the step size of discretization. As expected, we saw a higher degree of precision with smaller step sizes, until loss of numerical precision started to affect the results causing the relative error to increase. We also derived how the number of FLOPs, and thus the time needed for computation, depend on the number of steps for the different algorithms. We were then able to briefly discuss the tradeoff between high precision, low computational time and memory usage.

By discretizing the equation, one can rewrite it to a system of linear equations, in truth a quite simple linear algebra problem. The three different methods we have implemented for solving this set of linear equations is the general LU decomposition of a matrix, a general method for solving linear equations with tridiagonal matrices and finally a highly specialized algorithm for the specific matrix that arises from discretizing the second derivative of a function.

We obtained the best results with the specialized algorithm. With step size $h = 10^{-6}$ we got results with a maximum relative error of as little as $10^{-10}$. Beyond that, loss of numerical precision gave reduced precision. For the general tridiagonal algorithm, we observed a serious loss of precision at $h = 10^{-6}$. Both $h = 10^{-5}$ and $h = 10^{-4}$ gave better presicion than $h = 10^{-6}$. The LU decomposition is unable to handle step sizes smaller than $10^{-4}$ due to memory comsumption.

## I. INTRODUCTION

We can describe the evolution of many physical systems with the help of differential equations, but because of their complexity we are often unable to find analytical solutions. Therefore we have to use numerical methods in order to approximate the solution. Computers are limited in both their memory and accuracy, so we have to be careful when both selecting the numerical method and how we implement it. In this report we are going to explore these issues by trying to solve the one-dimensional Poisson equation with Dirichlet boundary conditions, described in equation 3 in the theory section.

The example function we are going to use in our studies is described by the equation (1) and has an analytical solution (2) we can use to compare our results.

$$-\frac{d^2u}{dx^2} = f(x) = 100e^{-10x} \tag{1}$$

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{2}$$

In order to solve equation (1) we end up with a set of linear equations described by a tridiagonal matrix multiplied with a vector (see the theory section for further explanation). Now there are many ways we can solve this set of equations, each with their own pros and cons. The methods we are going to explore is one where we solve for a general tridiagonal matrix, one where we specialize the algorithm to our tridiagonal matrix and lastly by using LU decomposition. For each method we will study the accuracy, cpu-time used and the number of floating-point operations (FLOPS).

For our studies we have used c++ for heavy computation, python for visualization and bash for automation. All the code along with instructions on how to run it, can be cloned fro our GitHub repository here [3].

## II. THEORY

### A. Matrix formulation of the discrete one-dimensional Poisson equation

The one-dimensional Poisson equation with Dirichlet boundary conditions that we are going to study can be written as equation 3.

$$-\frac{\mathrm{d}^2u(x)}{\mathrm{d}x^2} = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0 \tag{3}$$

We define the discretized approximation of $u$ to be $v_i$ at points $x_i = ih$ evenly spaced between $x_0 = 0$ and $x_{n-1} = 1$. The step length between the points is $h = 1/(n-1)$. The boundary conditions from equation 3 then give $v_0 = v_{n-1} = 0$. Deriving an approximation to the second derivative of $u$ from the Taylor expansion, equation 3 can be written

$$\frac{-v_{i-1} + 2v_i - v_{i+1}}{h^2} = f_i \quad \text{for } i = 1, 2, ..., n-2 \tag{4}$$

where $f_i = f(x_i)$.

Written out for all $i$, equation 4 becomes

$$\begin{aligned}
-v_0 + 2v_1 - v_2 &= h^2 f_1 \\
-v_1 + 2v_2 - v_3 &= h^2 f_2 \\
&\cdots \\
-v_{n-4} + 2v_{n-3} - v_{n-2} &= h^2 f_{n-3} \\
-v_{n-3} + 2v_{n-2} - v_{n-1} &= h^2 f_{n-2}
\end{aligned}$$

In general, this can be rearranged slightly so that

$$\begin{aligned} 2v_1 - v_2 &= h^2 f_1 + v_0 \\ -v_1 + 2v_2 - v_3 &= h^2 f_2 \\ &... \\ -v_{n-4} + 2v_{n-3} - v_{n-2} &= h^2 f_{n-3} \\ -v_{n-3} + 2v_{n-2} &= h^2 f_{n-2} + v_{n-1} \end{aligned}$$

This system of equations can be written in matrix form as

$$\boldsymbol{Av} = \tilde{\boldsymbol{b}}, \tag{5}$$

explicitly

$$\begin{bmatrix} 2 & -1 & 0 & 0 & ... & 0 \\ -1 & 2 & -1 & 0 & ... & 0 \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ 0 & ... & 0 & -1 & 2 & -1 \\ 0 & ... & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_{n-3} \\ v_{n-2} \end{bmatrix} = \begin{bmatrix} h^2 f_1 + v_0 \\ h^2 f_2 \\ . \\ . \\ . \\ h^2 f_{n-3} \\ h^2 f_{n-2} + v_{n-1} \end{bmatrix}$$

With $v_0 = v_{n-1} = 0$, the right side reduces to $\tilde{b}_i = h^2 f_i$ for $i = 1, 2, ..., n-2$.

### B. Lower-upper (LU) decomposition

LU decomposition is a method where you factorize a matrix $A$ into two matrices $L$ and $U$, where $L$ is lower triangular and $U$ is upper triangular. We can use this decomposition to solve a matrix equation.

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

Where $A$ and $\mathbf{b}$ is known. First we can solve $L\mathbf{y} = \mathbf{b}$ with the algorithm given by equation (6) (see [4]).

$$y_i = \frac{1}{l_{ii}}\left(b_i - \sum_{j=1}^{i-1} l_{ij} y_j\right), \quad y_1 = \frac{b_1}{l_{11}} \tag{6}$$

Where $y_i$ is element $i$ in $\mathbf{y}$, $l_{ij}$ element $ij$ in matrix $L$ and $b_i$ element $i$ in $\mathbf{b}$. We can then solve $U\mathbf{x} = \mathbf{y}$ with the algorithm from equation (7) (see [4]).

$$x_i = \frac{1}{u_{ii}}\left(y_i - \sum_{j=i+1}^{N} u_{ij} x_j\right), \quad x_N = \frac{y_N}{u_{NN}} \tag{7}$$

Here $x_i$ is element $i$ in $\mathbf{x}$.

### C. Floating-point operations

Whenever you do a mathematical operation on a computer you call it a floating-point operation. This includes division, multiplication, subtraction and addition. Floating-point operations (what we from now on will refer to as FLOPS) is a count of the total number of mathematical operations needed for an algorithm.

Many FLOPS usually leads to slower code, and if you have round off errors it can propagate and create larger errors than expected.

### D. Relative error

To better interpret results it can be a good idea to calculate the relative error. The formula for calculating the relative error is given by equation (8).

$$E = \frac{x_0 - x}{x} \tag{8}$$

Where $E$ is the relative error, $x_0$ the computed result and $x$ the actual result. In this project, we will treat the logarithm of the relative error, given by equation (9).

$$\epsilon = log_{10}(|E|) = log_{10}\left(\left|\frac{x_0 - x}{x}\right|\right) \tag{9}$$

## III. METHOD

### A. Solve tridiagonal matrix equation

In order to solve the tridiagonal matrix below we need to develop an algorithm. As mentioned in the exercise set [2] we first need to do a decomposition and forward substitution.

$$\mathbf{Av} = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{bmatrix} \tag{10}$$

Looking at the first matrix multiplication we get the following expression.

$$b_1 v_1 + c_1 v_2 = \tilde{b} \implies v_1 + \alpha_1 v_2 = \rho_1, \quad \alpha_1 = \frac{c_1}{b_1} \wedge \rho_1 = \frac{\tilde{b}_1}{b_1} \tag{11}$$

Doing the second matrix multiplication we get

$$a_1 v_1 + b_2 v_2 + c_2 v_3 = \tilde{b}_2 \tag{12}$$

If we multiply equation 11 by $a_1$, and subtract it from equation 12 the resulting expression becomes

$$(b_2 - \alpha_1 a_1)v_2 + c_2 v_3 = \tilde{b}_2 - \rho_1 a_1$$
$$\implies v_2 + \frac{c_2}{b_2 - \alpha_1 a_1} v_3 = \frac{\tilde{b}_2 - \rho_1 a_1}{b_2 - \alpha_1 a_1}$$
$$\implies v_2 + \alpha_2 v_3 = \rho_2$$
$$\text{where} \quad \alpha_2 = \frac{c_2}{b_2 - \alpha_1 a_1} \wedge \rho_2 = \frac{\tilde{b}_2 - \rho_1 a_1}{b_2 - \alpha_1 a_1}$$

Noticing the pattern in $\rho$ and $\alpha$ we can generalize the terms.

$$\alpha_i = \frac{c_i}{b_i - \alpha_{i-1} a_{i-1}} \quad \text{for} \quad i = 2, 3, ..., n-1 \tag{13}$$

$$\rho_i = \frac{\tilde{b}_i - \rho_{i-1} a_{i-1}}{b_i - \alpha_{i-1} a_{i-1}} \quad \text{for} \quad i = 2, 3, ..., n \tag{14}$$

Inserting the terms into the matrix above, we get a much simpler set of equations.

$$\mathbf{Av} = \begin{bmatrix} 1 & \alpha_1 & 0 & \dots & \dots & \dots \\ 0 & 1 & \alpha_2 & \dots & \dots & \dots \\ & 0 & 1 & \alpha_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & 0 & 1 & \alpha_{n-1} \\ & & & & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \rho_1 \\ \rho_2 \\ \dots \\ \dots \\ \dots \\ \rho_n \end{bmatrix}.$$

Now the last step is to do a backward substitution. Starting with $v_n = \rho_n$ we can work our way backward, with the general expression

$$v_{i-1} = \rho_{i-1} - \alpha_{i-1} v_i \quad \text{for} \quad i = n, n-1, ..., 2 \tag{15}$$

Now in this report we are going to consider a matrix with elements $b_n = 2$ and $a_n = c_n = -1$. We can insert this into equations (13) and (14) to get the expressions (16) and (17).

$$\alpha_i = \frac{-1}{2 + \alpha_{i-1}} \tag{16}$$

$$\rho_i = \frac{\tilde{b} + \rho_{i-1}}{2 + \alpha_{i-1}} \tag{17}$$

Counting the number of FLOPS in the algorithm we notice $N_1 = 3(n-2)$ from equation (13) (3 FLOPS per iteration, $n-2$ iterations), $N_2 = 5(n-1)$ from equation (14) (5 FLOPS per iteration $n-1$ iterations) and $N_3 = 3(n-2)$ from equation (15) (3 FLOPS per iteration $n-1$ iterations). Now to find the total number of flops for our algorithm we can sum them up.

$$N_{tot} = N_1 + N_2 + N_3 = 5(n-1) + 6(n-2) \approx 11n$$

The approximation holds when $n$ is large. Running the algorithm for $n = 10^i, \quad i = 2, 3, ..., 6$, we can plot the result, time the code and find the relative error using equation (9).

### B. Specialized algorithm for our spesific problem

Our specific problem is defined by the matrix equation 5. We will stay with only the left side of the equation to save som space during the derivation. In order to solve this matrix equation, we want to perform row operations on the matrix $\boldsymbol{A}$ to transform it to the identity matrix. We start by dividing the first line by 2, giving

$$\begin{bmatrix} 1 & -1/2 & 0 & 0 & ... & 0 \\ -1 & 2 & -1 & 0 & ... & 0 \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ 0 & ... & 0 & -1 & 2 & -1 \\ 0 & ... & 0 & 0 & -1 & 2 \end{bmatrix}$$

Now, adding the first line to the second gives

$$\begin{bmatrix} 1 & -1/2 & 0 & 0 & ... & 0 \\ 0 & 3/2 & -1 & 0 & ... & 0 \\ 0 & -1 & 2 & -1 & & \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ 0 & ... & 0 & -1 & 2 & -1 \\ 0 & ... & 0 & 0 & -1 & 2 \end{bmatrix}$$

Multiplying line 2 with $2/3$ gives

$$\begin{bmatrix} 1 & -1/2 & 0 & 0 & ... & 0 \\ 0 & 1 & -2/3 & 0 & ... & 0 \\ 0 & -1 & 2 & -1 & & \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ 0 & ... & 0 & -1 & 2 & -1 \\ 0 & ... & 0 & 0 & -1 & 2 \end{bmatrix}$$

Already, we see a pattern. On each line from line 2 onwards, we will add the previous line, giving us 0 to the left of the diagonal. It looks like the element to the right of the diagonal on line $i-1$ is given by

$$A_{i-1,i} = -\frac{i-1}{i} \tag{18}$$

As we add line $i-1$ to line $i$, the diagonal element of line $i$ becomes

$$2 - \frac{i-1}{i} = \frac{2i - i + 1}{i} = \frac{i+1}{i}. \tag{19}$$

We will then multiply line $i$ with $\frac{i}{i+1}$ giving us 1 on the diagonal and $-\frac{i}{i+1}$ directly to the right of the diagonal. This is the exact expression we assumed in expression 18, showing that it is in fact correct.

The forward substitution can therefore be done by dividing the first line by 2, and then for each following line (numbered by $i$), add the previous line and multiply the

result with $\frac{i}{i+1}$. After this procedure we end up with the following matrix

$$\begin{bmatrix} 1 & -1/2 & 0 & 0 & ... & 0 \\ 0 & 1 & -2/3 & 0 & ... & 0 \\ 0 & 0 & 1 & -3/4 & & \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ 0 & ... & 0 & 0 & 1 & -(n-3)/(n-2) \\ 0 & ... & 0 & 0 & 0 & 1 \end{bmatrix}$$

The backward substitution can then be done with a simple expression. Starting from the second to last line, numbered by $i = n-3$, we would like to add $\frac{i}{i+1}$ times the line below.

The forward and backward substitution can be performed by the following lines of code. Assume we have already filled the array $\boldsymbol{v}$ with values $v_i = h^2 f_i$ for $i = 1, 2, ..., n-2$.

```
v[0] = v[n-1] = 0   # Boundary conditions
v[1] = v[1]/2   # Divide the first line by 2

for (i = 2; i < n-1; i++) {
    v[i] = i/(i+1) * (v[i] + v[i-1])
}
for (i = n-3; i > 0; i--) {
    v[i] = v[i] + i/(i+1) * v[i+1]
}
```

By storing the values $i/(i+1)$ in an array, we can replace these two FLOPS with a memory fetch in our loops. We then have 2 FLOPS per iteration in the forward substitution (one addition and one multiplication) and two FLOPS per iteration in the backward substitution (one addition and one multiplication). For large values of $n$, the total number of FLOPS for the specialized algorithm is then

$$N_{\text{fast}} \approx 4n \tag{20}$$

### C. LU decomposition

In order to implement the LU decomposition we use the armadillo library (see [1] for documentation). From [5] we get that the LU decomposition itself requires $N_{LU} = 2n^3/3$ FLOPS. Looking at the equations (6) and (7) we can count the rest of the FLOPS. From equation (6) we get approximately $N \approx 4n$, giving us a total of $N_{tot} = 2n^3/3 + 4n$ FLOPS. Again running the algorithm for $n = 10^i, \quad i = 2, 3, 4, 5$, we can plot the result, time the code and find the relative error using equation (9).

### D. Code tests

We have constructed a quite simple test to assure that our algorithms performs as expected. Our code is written to solve the matrix equation

$$\begin{bmatrix} 2 & -1 & 0 & 0 & ... & 0 \\ -1 & 2 & -1 & 0 & ... & 0 \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ 0 & ... & 0 & -1 & 2 & -1 \\ 0 & ... & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_{n-3} \\ v_{n-2} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ b_{n-3} \\ b_{n-2} \end{bmatrix}$$

with $v_0 = v_{n-1} = 0$. We have used the following analytic solution to construct our test

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 6 \end{bmatrix} \implies \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} = \begin{bmatrix} 6 \\ 11 \\ 14 \\ 14 \\ 10 \end{bmatrix}$$

By providing this $\boldsymbol{b}$ as input and verifying that the computed solution is

$$\boldsymbol{v} = \begin{bmatrix} 0 \\ 6 \\ 11 \\ 14 \\ 14 \\ 10 \\ 0 \end{bmatrix}$$

we have checked that our code solves the matrix equation correctly.

## IV. RESULTS

Figure IV show the results obtained with step size $h = 10^{-2}$. The computed solution is plotted with the exact analytic solution. The plots for all the other step sizes can be found in the project GitHub repository [3] or can be produced as explained in the README-file of the project. They are all, however, indistinguishable from one another.

Tables I, II and III and figures IV, IV and IV show the values of $\epsilon$ (errors) for the different algorithms for all steps sizes.

| $\log_{10}(h)$ | $\epsilon$ |
|---|---|
| -6 | -6.08 |
| -5 | -8.84 |
| -4 | -7.08 |
| -3 | -5.08 |
| -2 | -3.08 |

Table I. This table shows the relative error for our general method. The firs column is the logarithm of our step size and the second column is the relative error (see figure IV for table plotted).

Table IV shows the computational time for each of the algorithm for all step sizes.

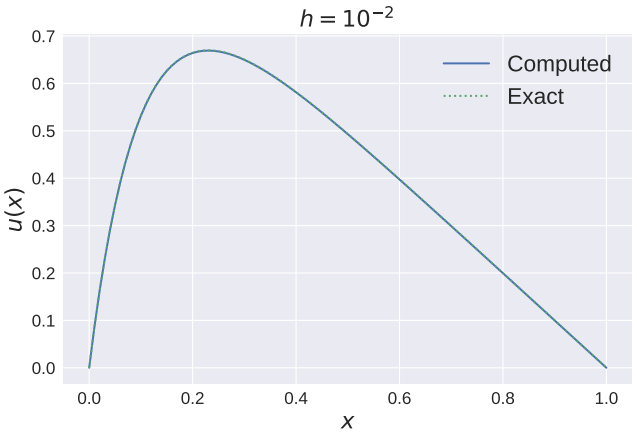


Figure 1. This figure shows the numeric solution for our general algorithm and the exact solution on top (green dashed line). The step size used is $h = 10^{-2}$.

| $\log_{10}(h)$ | $\epsilon$ |
|---|---|
| -7 | -9.65 |
| -6 | -10.18 |
| -5 | -9.08 |
| -4 | -7.08 |
| -3 | -5.08 |
| -2 | -3.08 |

Table II. This table shows the relative error for our specialized method. The firs column is the logarithm of our step size and the second column is the relative error (see figure IV for table plotted).

## V. DISCUSSION

Visually the plots does not change for different step sizes used, however looking at the relative error we can notice a change. For the first three step sizes $h = 10^{-2}, 10^{-3}, 10^{-4}$ the relative error is the same for all methods, however for smaller step sizes it differs quite a lot.

The LU decomposition fails for $n = 10^{-5}$. This is not surprising when we look at how the method works. For $n = 10^{-5}$ we need to store $10^10$ numbers where each

| $\log_{10}(h)$ | $\epsilon$ |
|---|---|
| -4 | -7.08 |
| -3 | -5.08 |
| -2 | -3.08 |
| -1 | -1.10 |

Table III. This table shows the relative error for our LU decomposition method. The first column is the logarithm of our step size and the second column is the relative error (see figure IV for table plotted).

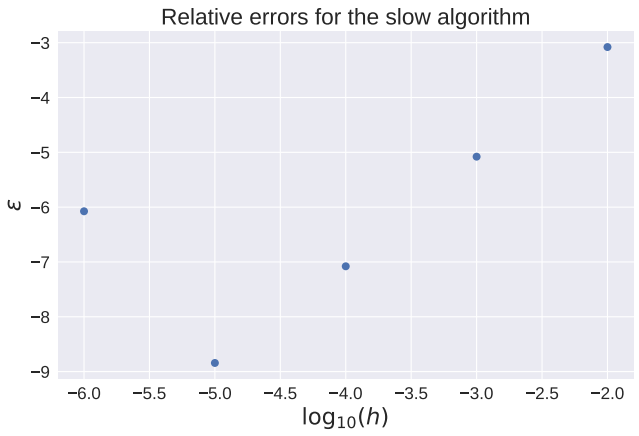


Figure 2. Here we have the table of errors (table I) from our general method plotted. Along the x-axis we have the logarithm of the step size ($log_{10}(h)$) and along the y-axis we have the relative error ($\epsilon$).

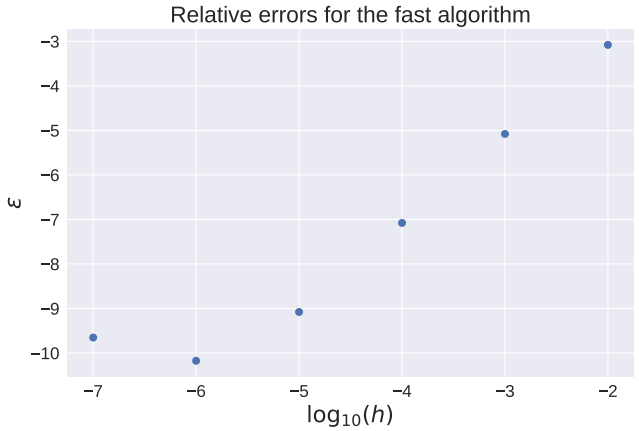


Figure 3. Here we have the table of errors (table II) from our specialized method plotted. Along the x-axis we have the logarithm of the step size ($log_{10}(h)$) and along the y-axis we have the relative error ($\epsilon$).

number uses eights bytes. This means we have to store 80GB of data, where normal laptops usually have 8-24GB of RAM (random access memory). Therefore our computers runs out of storage space. There are methods to counteract this problem. You can store the data to a bigger memory source like a hard drive, however this can be a lot slower than using RAM. We also notice that LU decomposition is slow compared to the two other methods. This is expected however, if we look at the number of FLOPS it's a lot more than the other two algorithms.

The most interesting comparison is the general and specialized algorithm. Time used is as expected, average speed is higher lower for the specialized algorithm across the board. More interesting however is how the errors

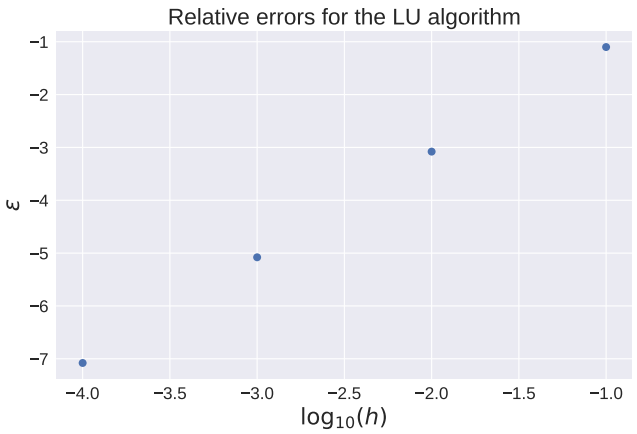


Figure 4. Here we have the table of errors (table III) from the LU decomposition method plotted. Along the x-axis we have the logarithm of the step size ($log_{10}(h)$) and along the y-axis we have the relative error ($\epsilon$).

| $n$ | Slow, [$\mu$s] | Fast, [$\mu$s] | LU, [$\mu$s] |
|---|---|---|---|
| $10^1$ | - | - | 144.60 |
| $10^2$ | 2.00 | 2.60 | 3285.00 |
| $10^3$ | 36.60 | 22.20 | 86933.40 |
| $10^4$ | 441.60 | 397.80 | 10779421.60 |
| $10^5$ | 6561.00 | 3219.00 | - |
| $10^6$ | 34875.80 | 14222.40 | - |

Table IV. This table shows the time used for the different algorithms in microseconds. The first column is the number of steps used, second the general algorithm, third the specialized one and the third correspond to the LU decomposition.

differ for low $h$. The specialized algorithm manages a relative error of $\epsilon = -10.18$ at $h = 10^{-6}$ and the general methods caps at $\epsilon = -8.84$ where $h = 10^{-5}$. This is surprising because both methods use the same numerical method, where the only difference is implementation. Although the relative error is lower for the specialized method, the downwards linear trend ends for both methods at around $h = 10^{-5}$. This gives us an indication that numerical errors occur for both methods already at $h = 10^{-6}$. Now with this in mind, the reason for difference in numerical precision might be because of FLOPS. It is logical to conclude, that with a higher number of FLOPS, error propagate further than with fewer FLOPS. This is because we have more operations where error can occur

## VI. CONCLUSION

Overall, we saw the expected behaviour of our algorithms. It was a little surprising that the loss of numerical precision started to affect the results at different values of $h$ for the different algorithms. This can probably be explained by the different numbers of FLOPS. For the specialized algorithm, there are fewer numbers stored in the computer and fewer operations on them. The loss of numerical precision therefore comes later here than for the general tridiagonal matrix approach where we store more values and perform more numerical operations. For the LU decomposition algorithm, we run out of memory before we reach values of $h$ that are small enough to cause loss of numerical precision.

Before loss of numerical precision starts to affect the results, we see the exact same errors for the different algorithms. This is an "analytic" error that comes from the approximation we make when we discretize the PDE.

The best results were obtained with the specialized algorithm. With a step size $h = 10^{-6}$ we got a maximum relative error of less than $10^{-10}$. For the general algorithm for tridiagonal matrices, the best result was obtained with step size $h = 10^{-5}$ giving a maximum relative error of $10^{-8.8}$. For smaller step sizes, loss of numerical precision caused the relative error to grow. The LU decomposition matched the other algorithms in accuracy for step sizes as small as $h = 10^{-4}$. Beyond that point, the matrices become too large to be represented in the computer's memory.

The specialized algorithm also proved superior in terms of computing time. It used a little under half the time of the general algorithm for tridiagonal matrices. The LU decomposition code took up to about 25 000 times as long time to complete as the specialized code.

[1] Armadillo, http://arma.sourceforge.net/
[2] Dept. of Physics, UiO, 2020, *Project 1, deadline September 9*
[3] Rustad, S. and Falmår, V., 2020, *FYS3150 Project 1*, https://github.com/sigurdru/FYS3150/tree/master/Project1
[4] Weisstein, Eric, W. on Mathworld–A Wolfram Web Resource, *LU Decomposition*, read 13.09.20, https://mathworld.wolfram.com/LUDecomposition.html
[5] Wikipedia. 2020, *LU decomposition*, read 13.09.20, https://en.wikipedia.org/wiki/LU_decomposition