FYS3150 - Project 1

Sigurd Sørlie Rustad and Vegard Falmår
(Dated: September 13, 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.

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

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

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 is given by equation 3.

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

We define the discretized approximation to u to be v_i at points $x_i = ih$ evenly spaced between $x_0 = 0$ and $x_n = 1$. The step length between the points is $h = 1/(n-1)$. The boundary conditions from equation 3 then give $v_0 = v_n = 0$. An approximation to the second derivative of u, derived from the Taylor expansion, is

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

where $f_i = f(x_i)$.

Written out for all i, equation 4 becomes

$$-v_0 + 2v_1 - v_2 = h^2 f_1$$
$$-v_1 + 2v_2 - v_3 = h^2 f_2$$
$$...$$
$$-v_{n-2} + 2v_{n-1} - v_n = h^2 f_{n-1}$$
$$-v_{n-1} + 2v_n - v_{n+1} = h^2 f_n$$

In general, this can be rearranged slightly so that

$$2v_1 - v_2 = h^2 f_1 + v_0$$
$$-v_1 + 2v_2 - v_3 = h^2 f_2$$
$$...$$
$$-v_{n-2} + 2v_{n-1} - v_n = h^2 f_{n-1}$$
$$-v_{n-1} + 2v_n = h^2 f_n + v_{n+1}$$

This system of equations can be written in matrix form as

$$\boldsymbol{A}\boldsymbol{v} = \tilde{\boldsymbol{b}}, \tag{5}$$

explicitly

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

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

B. 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.

$$
\boldsymbol{A}\boldsymbol{v} =
\begin{bmatrix}
b_1 & c_1 & 0 & ... & ... & ... \\
a_1 & b_2 & c_2 & ... & ... & ... \\
 & a_2 & b_3 & c_3 & ... & ... \\
 ... & ... & ... & ... & ... & \\
 & & & a_{n-2} & b_{n-1} & c_{n-1} \\
 & & & & a_{n-1} & b_n
\end{bmatrix}
\begin{bmatrix}
v_1 \\ v_2 \\ ... \\ ... \\ ... \\ v_n
\end{bmatrix}
=
\begin{bmatrix}
\tilde{b}_1 \\ \tilde{b}_2 \\ ... \\ ... \\ ... \\ \tilde{b}_n
\end{bmatrix}
\tag{6}
$$

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

$$b_1 v_1 + c_1 v_2 = \tilde{b} \implies v_1 + \alpha_1 v_2 = \rho_1, \quad \alpha_1 = \frac{c_1}{b_1} \wedge \rho_1 = \frac{\tilde{b}_1}{b_1} \tag{7}$$

Doing the second matrix multiplication we get

$$a_1 v_1 + b_2 v_2 + c_2 v_3 = \tilde{b}_2 \tag{8}$$

If we multiply equation 7 by a_1, and subtract it from equation 8 the resulting expression becomes

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

where $\alpha_2 = \frac{c_2}{b_2 - \alpha_1 a_1} \wedge \rho_2 = \frac{\tilde{b}_2 - \rho_1 a_1}{b_2 - \alpha_1 a_1}$

Noticing the pattern in ρ and α we can generalize the terms.

$$\alpha_i = \frac{c_i}{b_i - \alpha_{i-1} a_{i-1}} \quad \text{for} \quad i = 2, 3, ..., n-1 \tag{9}$$

$$\rho_i = \frac{\tilde{b}_i - \rho_{i-1} a_{i-1}}{b_i - \alpha_{i-1} a_{i-1}} \quad \text{for} \quad i = 2, 3, ..., n \tag{10}$$

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

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

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

$$v_{i-1} = \rho_{i-1} - \alpha_{i-1} v_i \quad \text{for} \quad i = n, n-1, ..., 2 \tag{11}$$

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 (9) and (10) to get the expressions (12) and (13).

$$\alpha_i = \frac{-1}{2 + \alpha_{i-1}} \tag{12}$$

$$\rho_i = \frac{\tilde{b} + \rho_{i-1}}{2 + \alpha_{i-1}} \tag{13}$$

C. 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 (14) (see [4]).

$$y_i = \frac{1}{l_{ii}}\left(b_i - \sum_{j=1}^{i-1} l_{ij}y_j\right), \quad y_1 = \frac{b_1}{l_{11}} \qquad (14)$$

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 (15) (see [4]).

$$x_i = \frac{1}{u_{ii}}\left(y_i - \sum_{j=i+1}^{N} u_{ij}x_j\right), \quad x_N = \frac{y_N}{u_{NN}} \qquad (15)$$

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

D. 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.

E. 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 (16).

$$E = \frac{x_0 - x}{x} \qquad (16)$$

Where E is the relative error, x_0 the computed result and x the actual result. You can also look at the logarithm of the relative error, this is given by equation (17).

$$log_{10}(E) = \epsilon = log_{10}\left(\left|\frac{x_0 - x}{x}\right|\right) \qquad (17)$$

III. METHOD

First we implement the numerical methods discussed in the theory section. As mentioned in the introduction, all the code and how to run it can be found in our GitHub repository here [3]. To implement the general algorithm, where we don't assume identical elements in the tridiagonal matrix (see equation (6)), we use the algorithms described in equations (9), (10) and (11). Counting the number of FLOPS in the algorithm we notice $N_1 = 3(n-2)$ from equation (9) (3 FLOPS per iteration, $n-2$ iterations), $N_2 = 5(n-1)$ from equation (10) (5 FLOPS per iteration $n-1$ iterations) and $N_3 = 3(n-2)$ from equation (11) (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 (17).

[METDOE FAST LINEQ]

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 (14) and (15) we can count the rest of the FLOPS. From equation (14) we get $N_1 = 2$ [FIKS TELLING AV 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 (17).
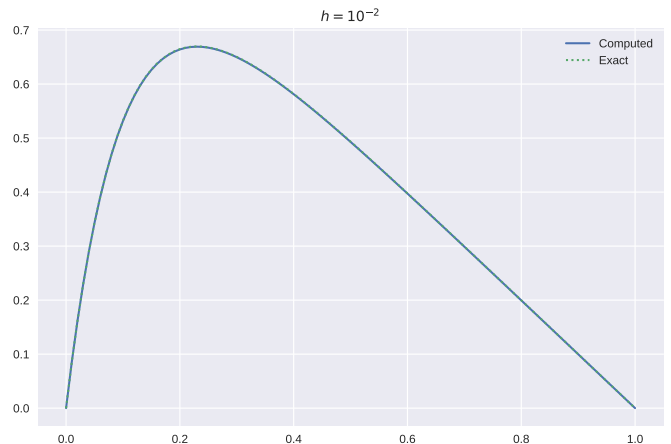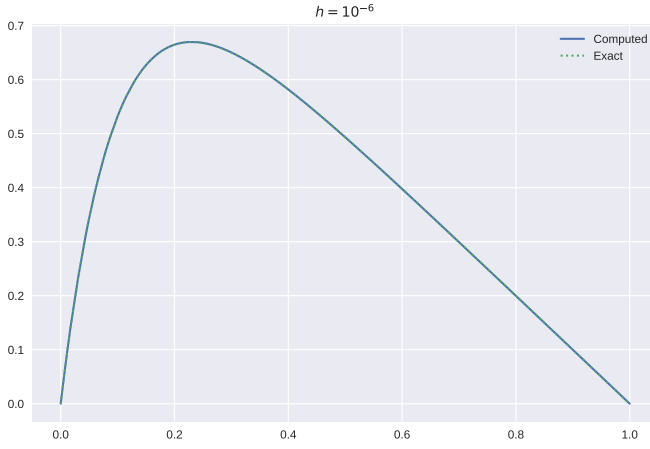
IV. RESULTS

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 $h = 10^{-2}$.

V. DISCUSSION

VI. CONCLUSION

Figure 2. This figure shows the numeric solution for our general algorithm and the exact solution on top (green dashed line). The step size used is $h = 10^{-6}$.

$\log_{10}(h)$	ϵ
-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 V for table plotted).

[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
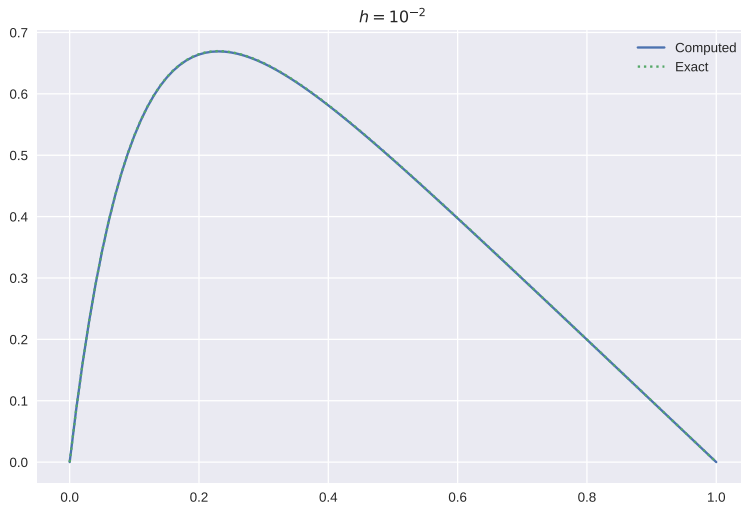
VII. APPENDIX

Figure 3. This figure shows the numeric solution for our specific algorithm and the exact solution on top (green dashed line). The step size used is $h = 10^{-2}$.
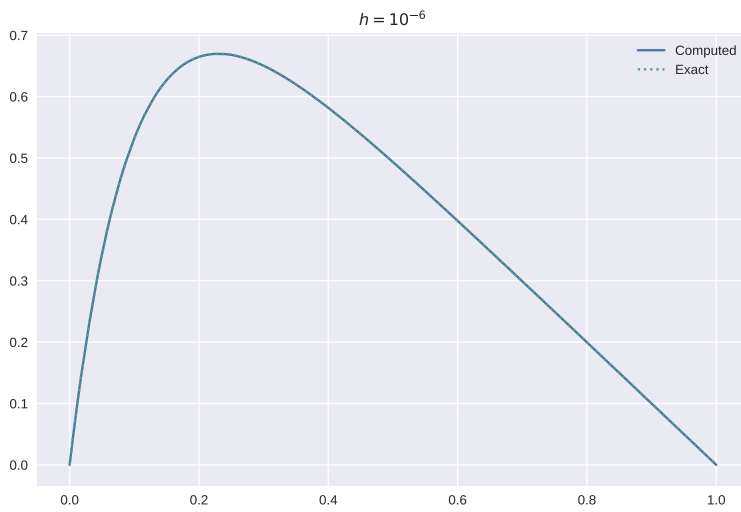

Figure 4. This figure shows the numeric solution for our specific algorithm and the exact solution on top (green dashed line). The step size used is $h = 10^{-6}$.

$\log_{10}(h)$	ϵ
-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 V for table plotted).
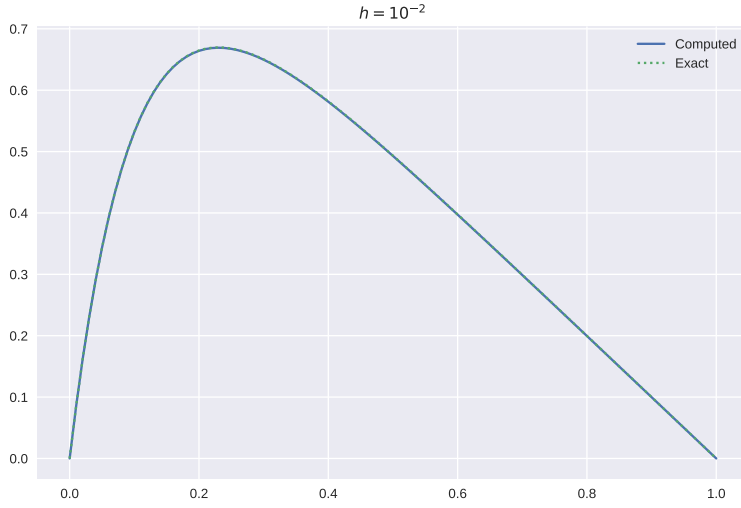

Figure 5. This figure shows the numeric solution for the LU-algorithm and the exact solution on top (green dashed line). The step size used is $h = 10^{-2}$.
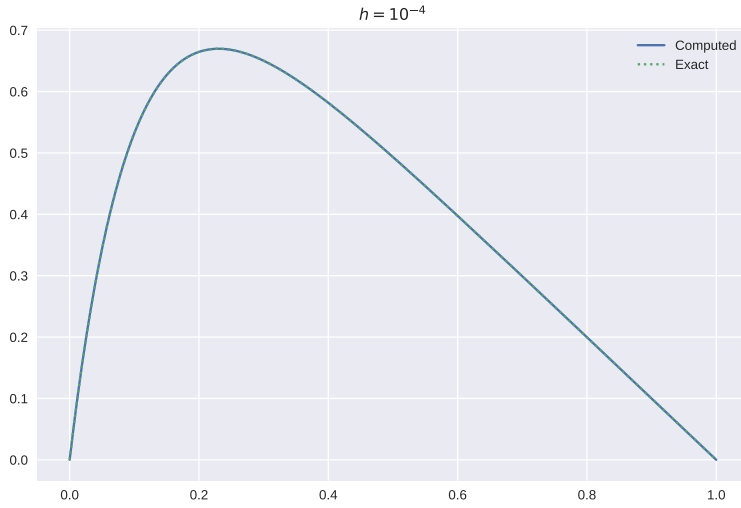

Figure 6. This figure shows the numeric solution for the LU-algorithm and the exact solution on top (green dashed line). The step size used is $h = 10^{-4}$.

$\log_{10}(h)$	ϵ
-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 firs column is the logarithm of our step size and the second column is the relative error (see figure V for table plotted).
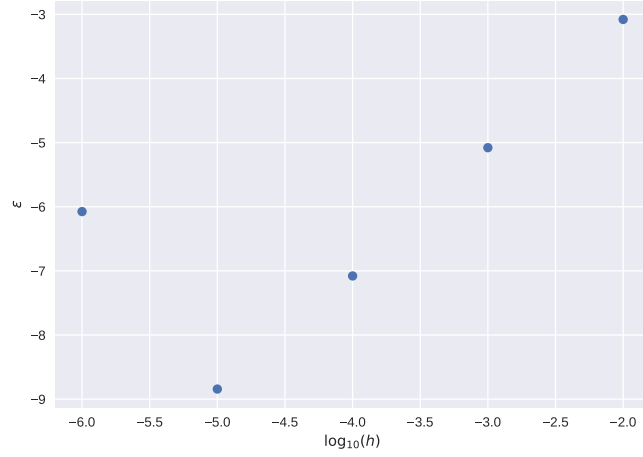

Figure 7. Here we have the table of errors (table I) from our general method plotted. Along the x-axis we have the logarithm of the step size ($log_{10}(h)$) and along the y-axis we have the relative error (ϵ).
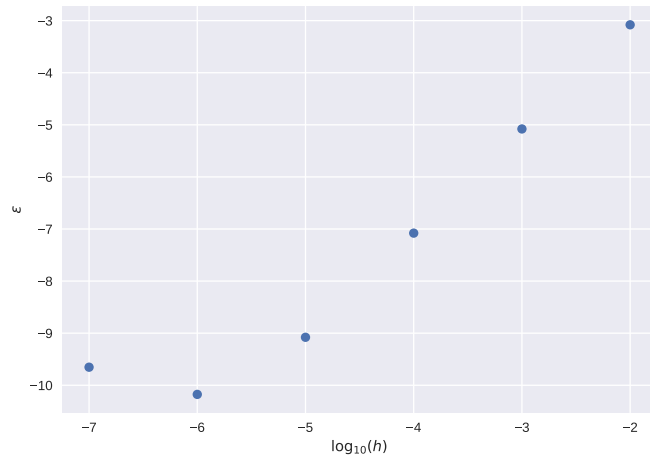

Figure 8. Here we have the table of errors (table II) from our specialized method plotted. Along the x-axis we have the logarithm of the step size ($log_{10}(h)$) and along the y-axis we have the relative error (ϵ).
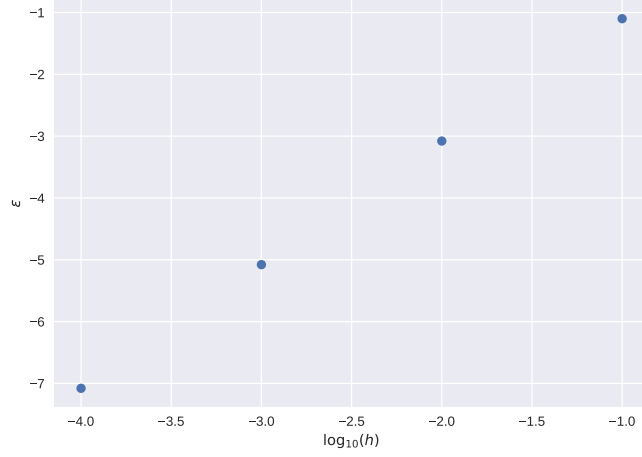
Figure 9. Here we have the table of errors (table III) from the LU decomposition method plotted. Along the x-axis we have the logarithm of the step size ($log_{10}(h)$) and along the y-axis we have the relative error (ϵ).

n	Slow [μs]	Fast [μs]	LU [μs]
10^1	-	-	140.00
10^2	2.60	2.00	2608.80
10^3	30.40	17.80	86337.00
10^4	572.60	200.60	10058519.80
10^5	7607.60	2741.00	-
10^6	35499.80	14172.00	-

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.

n	Slow	Fast	LU
10^1	-	-	140.00
10^2	2.60	2.00	2608.80
10^3	30.40	17.80	86337.00
10^4	572.60	200.60	10058519.80
10^5	7607.60	2741.00	-
10^6	35499.80	14172.00	-

Table V.