Henrik Lind Petlund Date: 09.09.2019

FYS3150 - Project 1

Git Repository: https://github.com/henrikx2/FYS3150

Abstract

In this report, an ordinary 2nd order differential equation (the Poisson equation) is solved numerically using an approximation to the 2nd derivative. The report shows how the approximation can be expressed as a matrix equation that can be simplified and specialized to obtain less memory allocation and faster calculation than i.e. a solution with LU-decomposition or a standard matrix multiplication. In the aspect of this, the loss of numerical precision when using small numbers is explored and documented.

1. Introduction

This experiment aims to solve the one-dimensional Poisson equation with Dirichlet boundary conditions. Poisson's equation is written as:

$$\nabla^2 \Phi = f \tag{1}$$

And the one-dimensional equation to solve is in this case expressed as:

$$-u''(x) = f(x) \ , \ \ x \in (0,1) \ , \ \ u(0) = u(1) = 0 \tag{2}$$

To solve the equation, a numerical approximation to the second derivative is derived and expressed as a matrix equation with a tridiagonal matrix. The matrix equation is solved generally with Gaussian elimination by rewriting it as three one-dimensional arrays and then specially by simplifying these expressions. Furthermore, the two algorithms are compared to a standard LU-decomposition for solving matrix equations. Lastly all three methods is to be compared by; the difference in number of floating points, CPU time usage, memory allocation and relative error as function of iterations n.

2. Methods and theory

2.1. Discretized values

To do integration on a computer, the variables have to be discretized. In this case the main function, u, is a function of x, and therefore x needs to be discrete. In other words;

$$x \to x_i = ih \ \ , \ \ \ i \in [1, \dots, n] \tag{3}$$

Where h is the distance (step size) between each discretized x and n is the number of x-values (mesh points) on the interval (0,1). In this explicit case, $x_0 = 0$ and $x_{n+1} = 1$, this means that there is $n + 1$ steps from x_0 to x_{n+1} and that the step size is:

$$h = \frac{1}{n+1} \tag{4}$$

Furthermore, the discretized approximation to u is denoted as v_i and $f(x) \to f(x_i) = f_i$.

2.2. Second derivative approximation

To calculate the second derivative numerically, two slightly tweaked taylor-expansions are used. Equation (5) shows the general expression of a taylor-expansion around a point a:

$$T_a(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{5}$$

Now, knowing that $f(x)$ is discretized, one step forward and one step backwards can be calculated using equation (5).

$$\begin{gathered} f(x+h) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!}(h)^n \\ = f(x+h) = f(x) + h\,f'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + O(h^4) \end{gathered} \tag{6}$$

$$\begin{gathered} f(x-h) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!}(-h)^n \\ = f(x-h) = f(x) - h\,f'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + O(h^4) \end{gathered} \tag{7}$$

The discrete values are denoted as follows:

$$f(x+h) \to f(x_i + h) = f_{i+1}$$

$$f(x-h) \to f(x_i - h) = f_{i-1}$$

To find an expression for the 2nd derivative equation (6) and (7) are added together and equation (8) is formed:

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + 2O(h^4)$$

$$f_i'' = \frac{f_{i+1} + f_{i-1} - 2f_i}{h^2} + O(h^2) \tag{8}$$

Equation (8) gives the possibility to calculate the 2nd derivative by knowing the function values at each step i. Furthermore; the error in this approximation will run like h^2, which will be explained later from a logarithmic point of view.

2.3. Tridiagonal matrix equation

Now, using equation (8) to describe equation (2), suggests:

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \tag{9}$$

Which can be rewritten as:

$$-v_{i+1} - v_{i-1} + 2v_i = h^2 f_i \tag{10}$$

This equation consists of three trailing v-values for every i used. By the following set of equations, it can be showed what Equation (10) looks like when i evolves from 1 to n:

$$\begin{gathered} -v_2 - v_0 + 2v_1 = h^2 f_1 \\ -v_3 - v_1 + 2v_2 = h^2 f_2 \\ \vdots \\ -v_{n+1} - v_{n-1} + 2v_n = h^2 f_n \end{gathered} \tag{11}$$

Further, there is the fact that $v_0 = v_{n+1} = 0$ and that in every equation there is one v_i multiplied with 2 and one v_{i+1} and one v_{i-1} multiplied with -1. This gives rise to a tridiagonal matrix of the type:

$$A = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & \cdots & 0 & \cdots \\ \cdots & \cdots & \cdots & \ddots & \cdots & \cdots \\ 0 & \cdots & \cdots & -1 & 2 & -1 \\ 0 & \cdots & \cdots & \cdots & -1 & 2 \end{bmatrix}$$

Such that the n-dimensional equation system can be written as a matrix equation as follows:

$$Av = b \tag{12}$$

Where:

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ \vdots \\ v_n \end{bmatrix} \quad , \quad b = \begin{bmatrix} b_1 \\ b_1 \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{bmatrix} = h^2 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ \vdots \\ f_n \end{bmatrix}$$

If the diagonals in matrix A is written as vectors a, d and c, this can be expressed in a general way as:

$$Av = \begin{bmatrix} d_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_1 & d_2 & c_2 & 0 & \cdots & \cdots \\ 0 & a_2 & d_3 & \ddots & 0 & \cdots \\ \cdots & \cdots & \ddots & \ddots & \ddots & \cdots \\ 0 & \cdots & \cdots & a_{n-2} & d_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & \cdots & a_{n-1} & d_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_1 \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{bmatrix} \tag{13}$$

It’s important to note that the endpoints of the vectors are not included in this matrix, because they are already known from the Dirichlet boundary conditions.

2.4. The analytical solution

The source term used in this report is $f(x) = 100e^{-10x}$, which gives the solution of equation (2) as

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{14}$$

By derivation this can be shown to be correct:

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

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

$$-u''(x) = f(x)$$

This function will be used to compare the results of the numerical solutions.

2.5. The General Algorithm

To solve the matrix equation (13) the algorithm called the Thomas Algorithm[1] will be used. The Thomas algorithm consists of a decomposition of the matrix into three arrays, a forward substitution and then a backwards substitution. It will look something like this (using Python notation):

Forward substitution

```
for i in range(1, n):
    d[i] = d[i] − c[i − 1] ∗ (a[i − 1]/d[i − 1])
    b[i] = b[i] − b[i − 1] ∗ (a[i − 1]/d[i − 1])
```

Backwards substitution

```
v[n − 1] = b[n − 1]/d[n − 1]
for i in range(n − 2,0, −1):
    v[i] = (b[i] + v[i + 1])/d[i]
```

The number of FLOPS required for this algorithm is approximately $6n$ FLOPS in the forward substitution and $3n$ FLOPS in the backwards substitution. Which gives a total of $9n$ FLOPS. There are also 10 memory reads and 3 memory writes. These

2.6. The Special Algorithm

In the case when all the elements at each diagonal is the same, the calculations can be simplified to have less FLOPS. In this case, we can use the fact that the arrays a and c are just -1. Also, in the forward substitution section, the d array can be precalculated. This makes the algorithm (in Python notation):

[1] **Thomas, L.H. (1949)**, *Elliptic Problems in Linear Differential Equations over a Network*, Watson Sci. Comput. Lab Report, Columbia University, New York.

Forward substitution

```
for i in range(1, n):
    b[i] = b[i] − 1b[i − 1]/d[i − 1])
```

Backwards substitution

```
v[n − 1] = b[n − 1]/d[n − 1]
for i in range(n − 2,0, −1):
    v[i] = (b[i] + v[i + 1])/d[i]
```

Which gives a total number of FLOPS of $4n$. The d-values in this special algorithm are calculated prior to the for-loop with the algorithm derived by entering the values of $d_i = 2$ and $a = c = -1$ in the expression of $d[i]$. This results in the expression:

$$d_i = 2 - \frac{1}{d_{i-1}} \quad , \quad i = 2,3, \dots n$$

For increasing values of i this can be written:

$$d_1 = 2$$

$$d_2 = 2 - \frac{1}{2} = \frac{3}{2}$$

$$\vdots$$

$$d_n = 2 - \frac{n+1}{n}$$

But by examining the these equations, the expression for the d_i-elements can be derived as:

$$\rightarrow d_i = \frac{i+1}{i} \quad , \quad i = 2,3, \dots , n$$

While $d_1 = 2$. This is a faster way of computing the d-values, rather than doing it recursively inside the for-loop. In other words this algorithm is expected to run 9/4 times faster than the general algorithm (based only on FLOPS).

2.7. LU-Decomposition

The last way to calculate and analyse the 2nd derivative in this report will be by using the LU-decomposition of the matrix A. The LU-decomposition of a matrix consists of a lower (L) and an upper (U) triangular matrix respectively. Pivoting is not necessary, since the matrix A is a non-singular matrix and therefore division by zero is no risk. An LU-decomposition of A can be expressed as:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & \cdots \\ \cdots & \cdots & \ddots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & \cdots & 0 \\ l_{21} & l_{22} & \cdots & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \ddots & 0 \\ l_{n1} & \cdots & \cdots & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & \cdots & \cdots \\ \cdots & \cdots & \ddots & \cdots & \cdots \\ 0 & \cdots & \cdots & \ddots & \cdots \\ 0 & \cdots & \cdots & 0 & u_{nn} \end{bmatrix}$$

This way of solving the matrix equation $Av = b$ will run with approximately $2/3\ n^3$ FLOPS. This can be shown by writing out the whole set of n equations and counting the FLOPS. So, this will (based on FLOPS) be a much slower way of computing the approximation v, than both the general and the special method. However, the memory reads/writes in the previous algorithms also play a significant role for some values of n. The LU-Decomposition method will be performed by the *lu_factor()* and *lu_solve()* functions in python.

2.8. Relative error

The relative error at the position x_i in the approximation of the 2nd derivative is calculated with the expression:

$$\epsilon_i = \left|\frac{v_i - u_i}{v_i}\right| \tag{15}$$

The error used in the error-plot will be the average of all the ϵ_i's for each step size h. By plotting these values as a log-log-plot ($\log_{10}$) the result should in theory be a linear graph with a slope of 2. This slope comes from the fact that the error in the second derivative approximation is a function of h^2. This is described in Eq. (8) by $O(h^2)$.

3. Results and discussion

3.1 General algorithm

Figure 3.1.1 shows the approximation $v(x)$ calculated with the general algorithm for values $n = 10, 10^2, 10^3$ compared to the analytical solution $u(x)$ from Eq. (14).

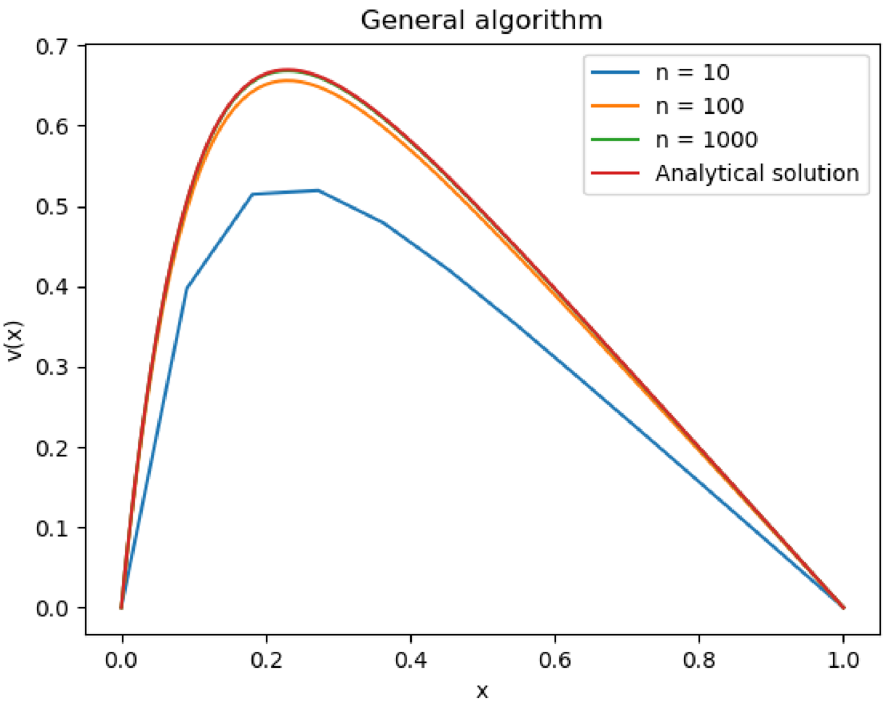

Figure 3.1.1: Solution with general algorithm for different values of n compared to the closed-form solution.

From the plot, it's easy to see the increasing precision with increasing n. Already when $n = 1000$ (green curve), the calculated curve is hidden underneath the analytical curve (at this magnification).

3.2 Special algorithm vs. General algorithm

Table 3.2.1 contains average CPU-time usage in both the general and the special algorithm. This average CPU-time is calculated from Table 5.1.1 and Table 5.1.2.

Table 3.2.1 – Average CPU-time usage in the General and Special algorithm

	General algorithm CPU-time	Special algorithm CPU-time	Speed fraction
n=10	$8.816 \cdot 19^{-5}$	$4.236 \cdot 10^{-5}$	2.08
n=10^2	$4.587 \cdot 10^{-4}$	$3.529 \cdot 10^{-4}$	1.30
n=10^3	$5.651 \cdot 10^{-3}$	$3.987 \cdot 10^{-3}$	1.42
n=10^4	$5.535 \cdot 10^{-2}$	$3.681 \cdot 10^{-2}$	1.50
n=10^5	0.5175	0.2601	1.99
n=10^6	5.429	3.377	1.61

From 2.5 and 2.6, it is expected that the special algorithm is $9/4 = 2.25$ times faster than the general algorithm. From the speed fraction in Table 3.2.1 it's clear that this is not the case, but the assumption that the special algorithm is faster, still holds quite well. The special algorithm is in fact the fastest for every value of n, but how much faster varies a lot. The reason for this may be that the program is not run isolated on the CPU. On a computer, the operation system (in this case Windows) will always run other tasks on the CPU simultaneously as the program runs. Depending on the heaviness of these task, the CPU-time of the algorithms may vary.

3.3 Comparing LU-Decomposition

The approximation done by LU-Decomposition is compared to the general and the special algorithm in Table 3.3.1. The fastest and slowest algorithms are shown with green and red colours respectively.

Table 3.3.1 – Average CPU-time usage in the General, Special and LU-Decomposition

	General algorithm CPU-time	Special algorithm CPU-time	LU-Decomposition CPU-time
n=10	$8.816 \cdot 19^{-5}$	$4.236 \cdot 10^{-5}$	$2.528 \cdot 10^{-4}$
n=10^2	$4.587 \cdot 10^{-4}$	$3.529 \cdot 10^{-4}$	$2.818 \cdot 10^{-3}$
n=10^3	$5.651 \cdot 10^{-3}$	$3.987 \cdot 10^{-3}$	$2.601 \cdot 10^{-2}$
n=10^4	$5.535 \cdot 10^{-2}$	$3.681 \cdot 10^{-2}$	10.796
n=10^5	0.5175	0.2601	Error
n=10^6	5.429	3.377	Error

From section 2.7, the LU-Decomposition method is expected to run with approximately $2/3\ n^3$ FLOPS. For large values of n, this would imply a much slower calculation time than both the general and special algorithm ($9n$ and $4n$ FLOPS respectively). This is correct, as the LU-Decomposition is the slowest one for all n-values. Though, if the FLOPS of the algorithms were to be the only thing determining the calculation time, the speeds would not be this "close". Depending on n, the special algorithm is varying from ~ 5 to ~ 300 times faster than de LU-Decomposition.

In the cases of $n = 10^5, 10^6$, the program will not calculate the LU-Decomposition method and returns an error. This error is due to the fact that a $10^5 \times 10^5$-matrix contains 10^{10}-elements. Every element in an array, matrix or variable on a computer, uses 8 bytes of random access memory (RAM). The computer must then allocate $8 \cdot 10^{10}$ bytes $\simeq 80\ GB$ of RAM. This much more than average people have on their computers.

One remark to note is the fact that the actual LU-decomposition of the matrix A is included when calculating the speed of the method. This may seem a bit unfair, because in the special algorithm, the array elements on the diagonal was precalculated. By inspecting how much time this actually takes (in Table 5.1.2 under Diagonal calculation) the time used (depending on n) has a maximum of 40% of the time used when pre calculation is excluded. The significance of the pre calculation does actually become smaller by increasing n. In other words; adding the pre calculation time to the time in Table 3.1.1 does not change the main results considerably.

3.4 Relative error

The relative error is described by the log-log plot in Figure 3.4.1 and shows how different values of the step size may impact the experimental results. The data behind the plot is listed in Table 5.2.1.

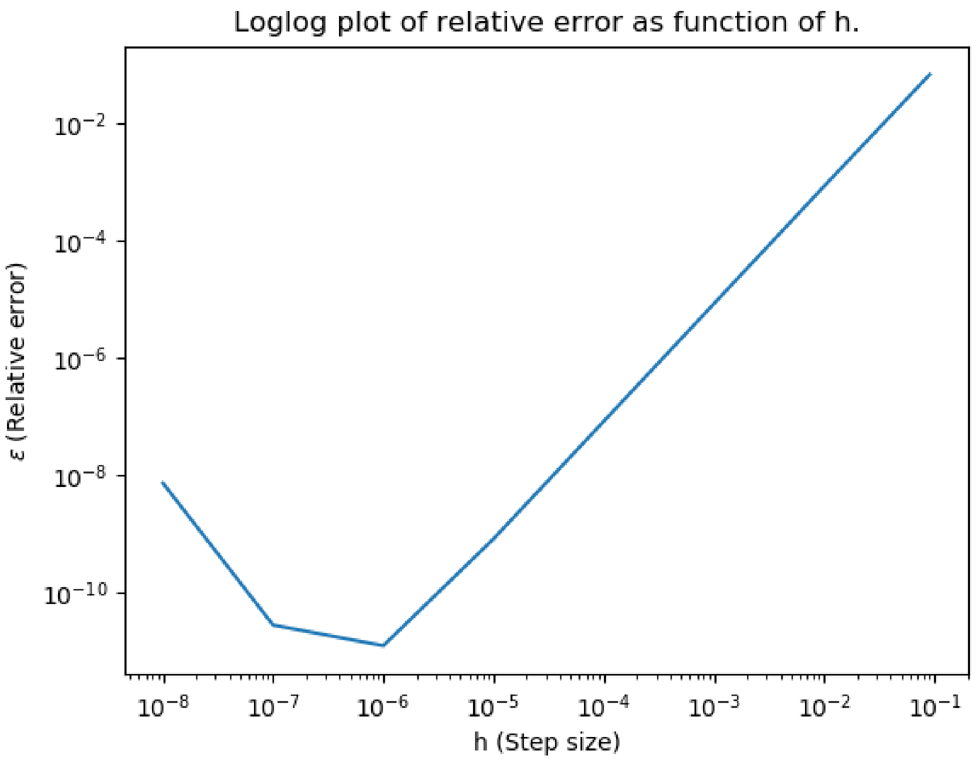

Figure 3.4.1: Log-log plot of the relative error as a function of step size h.

As expected from the last plot (Figure 3.1.1), there is an increasing precision as h decreases (n increases) and there is a linear curve in the beginning. But as the error reaches its minimum (around $h = 10^{-6}$), the error increases rapidly. There is a loss of numerical precision due to truncation and round-off errors which appear when working with small numbers ($\sim 10^{-16}$ with double precision) on a computer. This is how a rather small step size may make large loss of numerical precision because the error run as h^2.

The slope of the graph is obtained by use of the *numpy.polyfit()* method in python, which uses the least square method. This gives a slope of ~ 1.99, which coincide with the expected value from Equation (8).

4. Conclusion

This report has shown that a matrix equation with a tridiagonal matrix can be sped up to use less CPU time and less memory. This special algorithm can ultimately be used with a much larger number of iterations, while a standard matrix multiplication simply won't solve at all. The report also give a reminder that its always important to know the limits of the numerical precision of a computer, because not accounting for this may lead to large truncation/round-off errors. This is an encouragement that one should always strive to seek better ways to solve problems that may not be fully optimized.

5. Appendix

5.1 CPU-time data

Table 5.1.1 and 5.1.2 contains 5 sets of CPU-time usage in both the General and the Special algorithm for values of $n = 10^1, 10^2, \dots, 10^6$. The CPU-time usage presented in Table 5.1.3 contains CPU-time usage in the LU-Decomposition.

Table 5.1.1 – CPU-time in General algorithm from 5 different runs.

					General algorithm
Run:	1	2	3	4	5
n=10	0.0001170	0.0001228	0.0000609	0.0000477	0.0000924
n=10^2	0.0003081	0.0002983	0.0005668	0.0005648	0.0005556
n=10^3	0.0040404	0.0038759	0.0072516	0.0062470	0.0068394
n=10^4	0.0483207	0.0326118	0.0699969	0.0639499	0.0618664
n=10^5	0.3871088	0.2814706	0.6291117	0.6299004	0.6598148
n=10^6	3.6926919	4.7178563	6.1608409	6.6048769	5.9710214

Table 5.1.2 – CPU-time in Special algorithm from 5 different runs and CPU-time of calculation of the diagonal elements.

						Special algorithm
Run:	1	2	3	4	5	Diagonal calculation
n=10	0.0000403	0.0000433	0.0000194	0.0000854	0.0000234	$9.6 \cdot 10^{-6}$
n=10^2	0.0003088	0.0001563	0.0002143	0.0008692	0.0002160	$2.4 \cdot 10^{-5}$
n=10^3	0.0020578	0.0018289	0.0033968	0.0088157	0.0038368	$3.7 \cdot 10^{-4}$
n=10^4	0.0204579	0.0197546	0.0377723	0.0693642	0.0367184	$6.3 \cdot 10^{-3}$
n=10^5	0.1669740	0.2652897	0.2802256	0.3940636	0.1939927	$3.7 \cdot 10^{-2}$
n=10^6	2.2580291	3.8945884	3.7121895	4.0959852	2.9244365	$2.6 \cdot 10^{-1}$

Table 5.1.3 – CPU-time in LU-Decomposition from 5 different runs.

					LU-Decomposition
Run:	1	2	3	4	5
n=10	0.0002393	0.0002920	0.0002796	0.0002238	0.0002291
n=10^2	0.0020839	0.0020134	0.0035307	0.0018647	0.0045986
n=10^3	0.0257322	0.0275712	0.0256663	0.0243680	0.0267002
n=10^4	10.211161	10.085833	12.274161	10.730455	10.679242

5.2 Relative error

Table 5.2.1 contains the $\log_{10}$-values of the step size for $n = 10^1, 10^2, \dots, 10^8$ with corresponding $\log_{10}$-values of the relative error in the special algorithm.

Table 5.2.1 – Log-log values of relative error as function of step size h

Step size, log(h)	**Relative error, log(ϵ)**
-1.04139	-1.1797
-2.00432	-3.08804
-3.00043	-5.08005
-4.00004	-7.07927
-5	-9.0791
-6	-10.8953
-7	-10.5446
-8	-8.12941

References

[1] **Thomas, L.H. (1949)**, *Elliptic Problems in Linear Differential Equations over a Network*, Watson Sci. Comput. Lab Report, Columbia University, New York.

[2] **Hjorth-Jensen, M. (2015)**, *Computational Physics, Lecture Notes Fall 2015*, Department of Physics, University of Oslo. https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf