Project 1

Johan Carlsen, Mads Saua Balto, Anton Brekke, Ali Resa

September 13, 2022

Problem 1

We are interested in the solution for

$$-\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = 100e^{-10x}$$

It is claimed that $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$ is a solution. Inserting this into the lefthand side of the original expression will show that this is indeed a solution if both sides of the equation still match:

$$\begin{aligned} &- \frac{\mathrm{d}^2}{\mathrm{d}x^2}\left(1 - (1 - e^{-10})x - e^{-10x}\right) \\ &= -(-100e^{-10x}) \\ &= 100e^{-10x} \end{aligned}$$

As such, we've shown that $u(x) = 1-(1-e^{-10})x-e^{-10x}$ is an exact solution.

Problem 2

We are looking to write a program that defines a vector of x-values, and a function that evaluates the excact solution over these values, the results of which will be stored in 2 columns, with a fixed amount of decimals, in scientific notation. This data will also be plotted using a separate script.

The function

we've implemented a function that returns a double $u(x)$ for an argument x, who's type is double, such that:

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

The function "analytic_sol" is declared as the type double, and so is it's (only) parameter "x" too. the function body evaluates $u(x)$ as shown in (1), and returns the result. Vectors x and u were initialized to contain 101 elements doubles, and x was fully defined by assigning each element within a for-loop, iterating through the elements the elements and assigning each one a value. (each element is assigned the value of the previous element plus 1/100, starting at 0, such that $x_{i+1} = x_i$). u was defined using a for loop, calling "analytic_sol" with x_i as the input, such that $u_i = \text{analytic\_sol}(x_i)$.

x is a vector containing 101 linearly spaced doubles ranging from 0 to 1, and u is similarily a vector of doubles, such that $u_i = u(x_i)$. The elements of these vectors are stored in a .dat file, and loaded using python, for which the data is used to initialize numpy arrays, for the purpose of displaying u(x). The results of which can be seen in the figure:

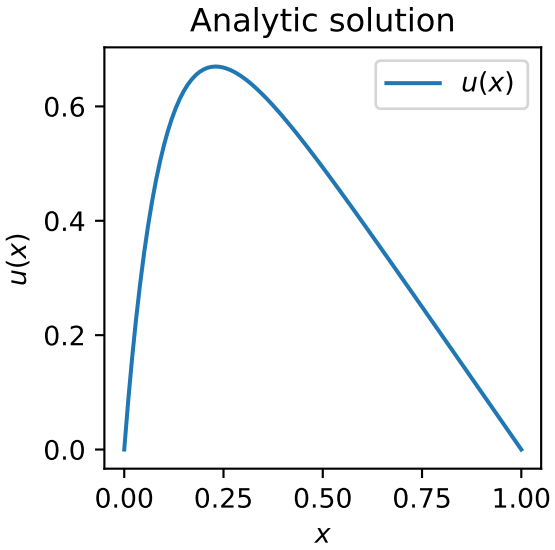

Fig. 1: Shows 101 linearly spaced points starting at 0 and ending at 1, evaluated on the excact solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$.

Problem 3

A taylor series for a function $u(x)$ around $x = a$ is defined as

$$u(x) = \sum_{n=0}^{\infty} \frac{u^{(n)}(x-a)^n}{n!}$$

$u(x+h)$ and $u(x-h)$ may both be expanded around x such that:

$$u(x+h) = u(x) + \frac{\mathrm{d}u}{\mathrm{d}x}h + \frac{1}{2}\frac{\mathrm{d}^2u}{\mathrm{d}x^2}h^2 + \sum_{n=3}^{\infty}\frac{u^{(n)}h^n}{n!}$$
$$u(x-h) = u(x) - \frac{\mathrm{d}u}{\mathrm{d}x}h + \frac{1}{2}\frac{\mathrm{d}^2u}{\mathrm{d}x^2}h^2 + \sum_{n=3}^{\infty}\frac{u^{(n)}h^n(-1)^n}{n!}$$

The sum of the expansions is useful for deriving an expression for $\frac{\mathrm{d}^2u}{\mathrm{d}x^2}$

$$u(x+h) + u(x-h) = 2u(x) + \frac{\mathrm{d}^2u}{\mathrm{d}x^2}h^2 + \sum_{n=3}^{\infty}\frac{u^{(n)}h^n(1+(-1)^n)}{n!}$$
$$\Rightarrow \frac{\mathrm{d}^2u}{\mathrm{d}x^2} = \frac{u(x+h)+u(x-h)-2u(x)}{h^2} + \sum_{n=3}^{\infty}\frac{u^{(n)}h^{n-2}(1+(-1)^n)}{n!}$$

For the purpose of discretizing, the rest of the sum will not be included, which leads to an error in the approximation of order h^2. As such: $O(h^2) = \frac{u^{(n)}h^{n-2}(1+(-1)^n)}{n!}$. h^2 is by far the largest element of this sum when h approaches 0, and will be the dominating part of the truncation error, as such:

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = \frac{u(x+h)+u(x-h)-2u(x)}{h^2} + O(h^2)$$

Discretizing:

$$u''(x) \mapsto u_i'',\ u(x-h) \mapsto u_{i-1},\ u(x) \mapsto u_i,\ u(x+h) \mapsto u_{i+1}$$
$$u_i'' = \frac{u_{i+1}-2u_i+u_{i-1}}{h^2} + O(h^2)$$
$$\Rightarrow -u_i'' = \frac{-u_{i-1}+2u_i-u_{i+1}}{h^2} + O(h^2) \tag{1}$$
$$\approx v_i'' = \frac{-v_{i-1}+2v_i-v_{i+1}}{h^2}$$

Where $v_i \approx u_i$. This discretization of $\frac{\mathrm{d}^2u}{\mathrm{d}x^2}$ may be applied to discretize the poisson equation:

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

Where $f(x) \equiv f_i$.

Problem 4

Let $\mathbf{v}$ be the approximation of (1), that is

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

In addition, the discrete solution of the sourcing term, $f(x)$, will be represented by $\mathbf{g}$. This leads to the following equation.

$$\begin{aligned} v_i'' &= g_i \\ \frac{-v_{i-1} + 2v_i - v_{i+1}}{h^2} &= f(x_i) \\ -v_{i-1} + 2v_i - v_{i+1} &= h^2 f(x_i) = h^2 g_i \end{aligned} \tag{3}$$

We recognize that (3) can be written as a matrix equation. The following derivation shows how this matrix equation works, step by step, for $i = 1, 2, 3, 4$

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = h^2 \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix}$$

$$\begin{aligned} b_1 v_1 + c_1 v_2 &= h^2 g_1 \\ a_2 v_1 + b_2 v_2 + c_2 v_3 &= h^2 g_2 \\ a_3 v_2 + b_3 v_3 + c_3 v_4 &= h^2 g_3 \\ a_4 v_3 + b_4 v_4 &= h^2 g_4 \end{aligned}$$

Now, we know that the Dirichlet conditions are satisfied, meaning the solution is zero for the first and last element. In our matrix equation, this corresponds to $a_1 = 0$ and $c_4 = 0$.

In general, this can written as

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = h^2 g_i \tag{4}$$

We notice that (4) is identical to (3), for $a_i = -1$, $b_i = 2$ and $c_i = -1$, i.e. the subdiagonal, main diagonal and superdiagonal respectively. Thus, the matrix equation $\mathrm{A}\mathbf{v} = \mathbf{g}$, in the general case looks like this:

$$\begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & -1 & 2 & -1 \\ 0 & \dots & 0 & -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_n \end{bmatrix}$$

An element in **g** is thus related to the original differential equation as stated in (3), by $f(x_i) = h^2 g_i$.

Problem 5

If the vector **v*** of length m represents the complete solution to the Poisson equation, solved for x-values in the vector **x** of length m, and A= $N \times N$, then $m = N + 2$. This is the result of the boundary conditions, that (in our case) both $v(0)$ and $v(1)$ are known (to be 0).

When we solve the matrix equation, we will then find the solution vector **v**, with elements corresponding to the elements in **v*** minus the first and last element. That is

$$v_i = v_i^*$$

for $i = 1, 2, 3, \ldots, m-2$, assuming that **v***= v_k^* for $k = 0, 1, 2, \ldots, m-1$.

Problem 6

The algorithm

For a general matrix equation A**v**=**g**, with the vectors **a**, **b** and **c** representing the sub-, main- and super-diagonal, respectively, we can derive an algorithm to solve the equation. Note that we want to solve the equation for **v**.

$$\begin{bmatrix} b_1 & c_1 & \ldots & 0 \\ a_2 & b_2 & c_2 & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & a_n & b_n \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}$$

The first step is called the forward substitution in the Gauss elimination method. We want the matrix to be upper triangular. We do this by eliminating all the a_i's in A. For a given row, we can eliminate the a_i by subtracting $\frac{a_i}{b_{i-1}}$ times the previous row from it.

For row number two this would look as follows.

$$\begin{aligned}(a_2,\, b_2,\, c_2,\, 0,\, \ldots,\, 0) &\mapsto \left(a_2 - \frac{a_2 b_1}{b_1},\, b_2 - \frac{a_2 c_1}{b_1},\, c_2,\, 0,\, \ldots,\, 0\right) \\ &= \left(0,\, b_2 - \frac{a_2 c_1}{b_1},\, c_2,\, 0,\, \ldots,\, 0\right)\end{aligned}$$

This also applies to the solution vector $\mathbf{g}$, where

$$g_2 \mapsto g_2 - \frac{a_2}{b_1} g_2$$

Let us now define three variables, w_i, $\tilde{b}_i$ and $\tilde{g}_i$.

$$\begin{aligned} w_i &\equiv \frac{a_i}{\tilde{b}_{i-1}} \\ \tilde{b}_1 &= b_1 & \tilde{g}_1 &= g_1 \\ \tilde{b}_i &\equiv b_i - w_i\, c_{i-1} & \tilde{g}_i &\equiv g_i - w_i\, \tilde{g}_{i-1} \end{aligned}$$

The matrix equation will now look like this:

$$\begin{bmatrix} \tilde{b}_1 & c_1 & \dots & 0 \\ 0 & \tilde{b}_2 & c_2 & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & \tilde{b}_n \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{g}_1 \\ \tilde{g}_2 \\ \vdots \\ \tilde{g}_n \end{bmatrix}$$

As we can see, the last row in A consists of all zeros except the last element. Thus, the solution to the last equation is

$$\tilde{b}_n\, v_n = \tilde{g}_n$$

$$\Rightarrow v_n = \frac{\tilde{g}_n}{\tilde{b}_n}$$

The next-to-last row will have solution

$$\tilde{b}_{n-1}\, v_{n-1} + \tilde{b}_n\, v_n = \tilde{g}_{n-1}$$

$$\Rightarrow v_{n-1} = \frac{\tilde{g}_{n-1} - \tilde{b}_n\, v_n}{\tilde{b}_{n-1}}$$

This leads to the algorithm

$$v_i = \frac{\tilde{g}_i - \tilde{b}_{i+1}\, v_{i+1}}{\tilde{b}_i}\,, \quad \text{for } i = n-1,\, n-2,\, n-3,\, \dots,\, 1$$

FLOPs

Now we will take a look at how many FLOPs (floating point operations) this general algorithm will have to do to solve the equation. Each operation, (+, -, $\times$, $\div$), will count as 1 FLOP.

Lets list all of the operations we have to do.

$$
\begin{aligned}
w_i &= \frac{a_i}{\tilde{b}_{i-1}} \,, && \text{for } i = 2,\, 3,\, \ldots,\, n \\
\tilde{b}_i &= b_i - w_i\, c_{i-1} \,, && \text{for } i = 2,\, 3,\, \ldots,\, n \\
\tilde{g}_i &= g_i - w_i\, \tilde{g}_{i-1} \,, && \text{for } i = 2,\, 3,\, \ldots,\, n \\
v_n &= \frac{\tilde{g}_n}{\tilde{b}_n} \\
v_i &= \frac{\tilde{g}_i - \tilde{b}_{i+1}\, v_{i+1}}{\tilde{b}_i} \,, && \text{for } i = n-1,\, n-2,\, \ldots,\, 1
\end{aligned}
$$

In Table 1 all the FLOPs per variable are listed and summed up.

Variable	**FLOPs**	**Accumulated**
w_i	$n-1$	$n-1$
$\tilde{b}_i$	$2(n-1)$	$3(n-1)$
$\tilde{g}_i$	$2(n-1)$	$5(n-1)$
v_n	1	$5(n-1)+1$
v_i	$3(n-1)$	$8(n-1)+1$
	Total:	$8n-7$

Table 1: All the FLOPs required per variable listed together with the accumulated number in the rightmost column.

The total number of FLOPs required to solve the matrix equation A**v**=**g**, where A is a $N \times N$-matrix, is $8N-7$, i.e. of $O(N)$.

Problem 7

Now we're getting back to the Poisson-equation:

$$-\frac{d^2u}{dx^2} = f(x) \tag{3}$$

Our goal for this task is to write the Thomas Algorithm in its most general form, in order to solve the matrix equation

$$A\vec{v} = \vec{g} \tag{4}$$

where A is the tridiagonal matrix from Problem 4 in order to solve the Poisson equation. The link for the Github to the implemented code will be given

in the appendix.

Now that we have implemented this algorithm, we want to test it for different amount of discretization steps along the x-axis. We solved the matrix equation for $n_{\text{steps}} = \{10,\ 10^2,\ 10^3,\ 10^4,\ 10^5,\ 10^6\}$ and plotted it together with the analytical solution $u(x)$ found in equation (1), and the result can be seen in Figure (2)

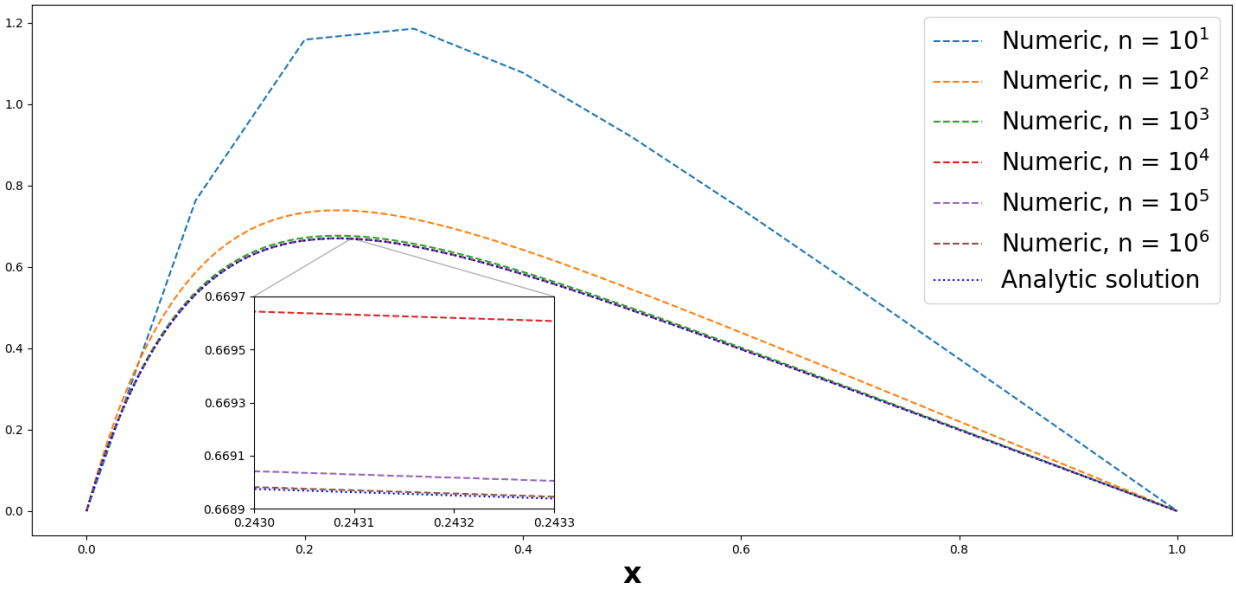

Fig. 2: Solutions of the poisson equtaion over different amount of discretization steps ranging from 10 to 10^6 compared to the analytical solution.

Problem 8

While solving the equation in **??** it could be useful to look at the errors of the algorithm. Therefore, we will find the absolute error

$$\Delta_i = |u_i - v_i|$$

where u_i corresponds to data-element i of the exact solution, and v_i the data-element i of the approximated solution. We will also find the relative error

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

and plot the logarithm of these for different choices of n_{steps}. We still restrict ourselves to look at $n_{\text{steps}} = \{10,\ 10^2,\ 10^3,\ 10^4,\ 10^5,\ 10^6\}$. We choose to plot the absolute error and relative error, but scale the axis to a logarithmic scale with base 10. Therefore its also important to note that we will leave out the boundary points of the data set, since they have the value 0 (and logarithms do not like zeros). The result for the absolute error can be seen in Figure (3)

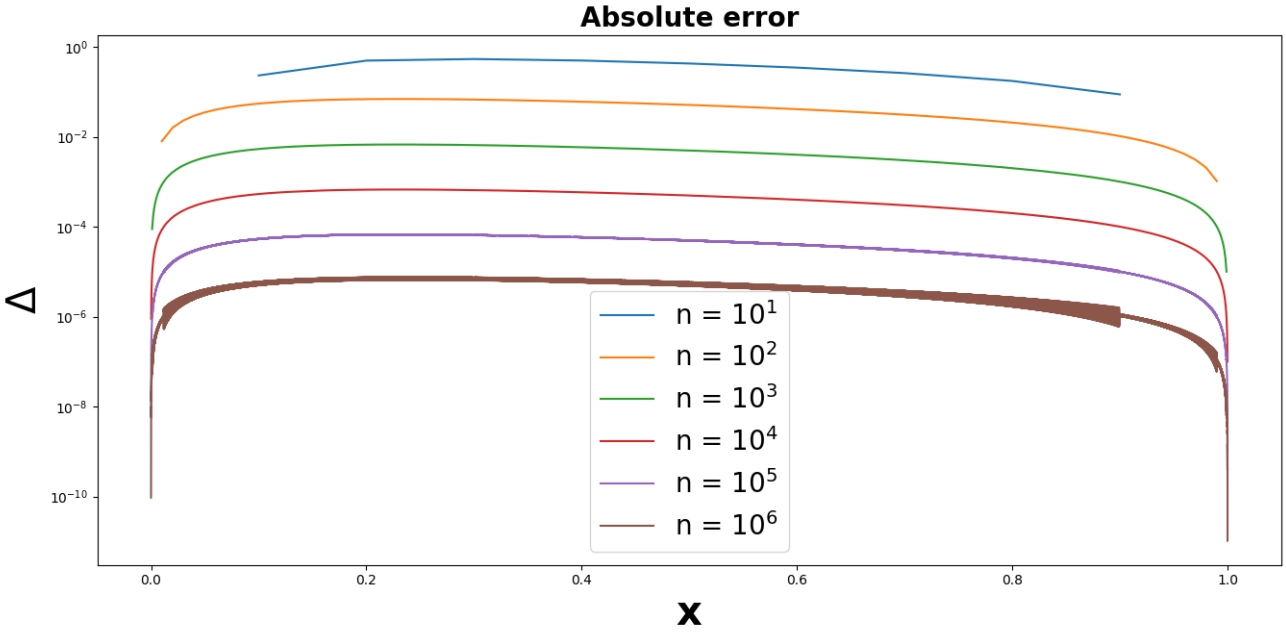

Fig. 3: The absolute error on every different x-value for different choices of n_{steps}.

and we to the exact same thing for the relative error seen in Figure (4).

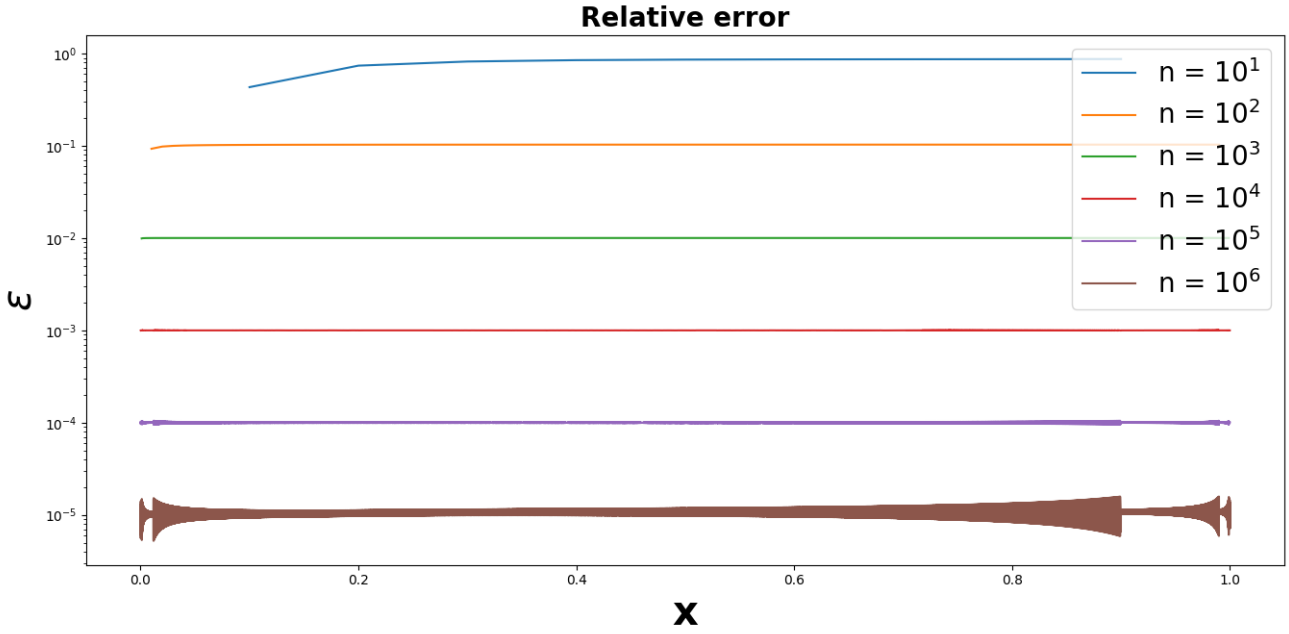

Fig. 4: The relative error on every different x-value for different choices of n_{steps}.

These plots do not give us that much of information other than how the errors vary over the x-axis. It would be more interesting to look at how they vary over the number of discretization steps n_{steps}, as this would tell us how big the maximum relative error becomes as we try to make the simulation more precise. Therefore we take the maximum relative error for each n_{steps}, and put them in a table for us to analyze. We will also plot them logarithmically over n_{steps}. The result can be seen in Table (2) and Figure (5)

n_{steps}	relative error
10^1	$8.752 \cdot 10^{-1}$
10^2	$1.033 \cdot 10^{-1}$
10^3	$1.004 \cdot 10^{-2}$
10^4	$1.005 \cdot 10^{-3}$
10^5	$1.049 \cdot 10^{-4}$
10^6	$1.581 \cdot 10^{-5}$
10^7	$8.487 \cdot 10^{10}$

Table 2: Table containing the maximum relative error for each choice of n_{steps}.

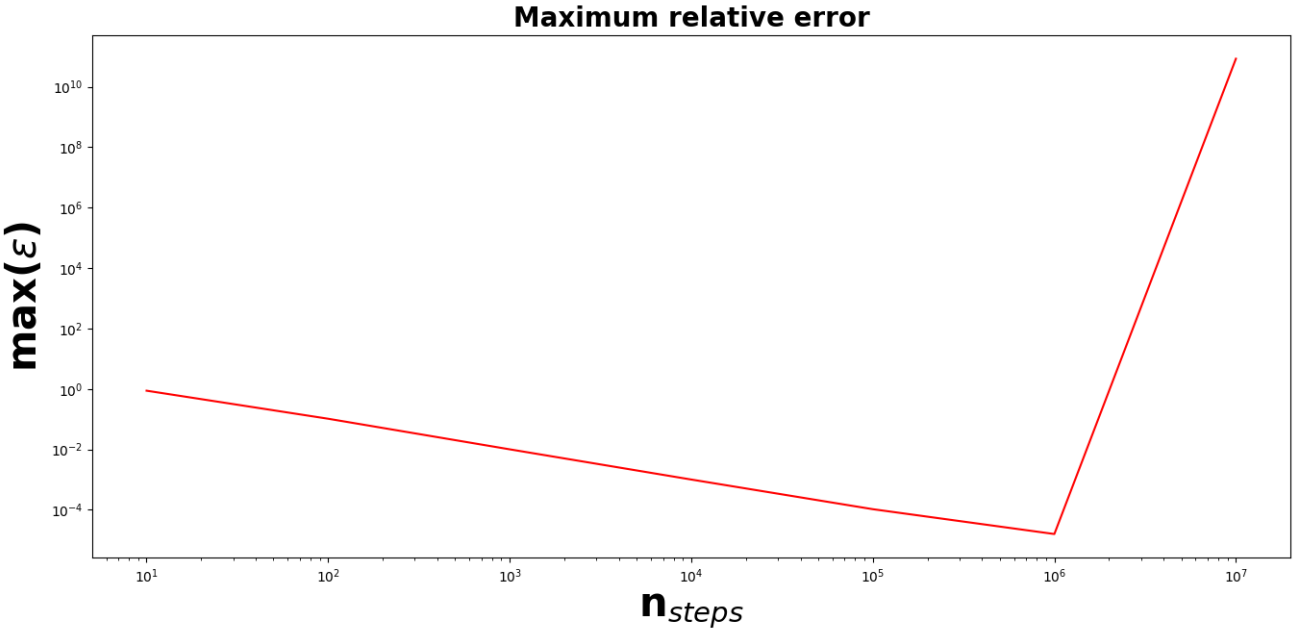

Fig. 5: The maximum relative error of each choice of n_{steps} plotted over n_{steps}.

As we would expect, an increased amount of discretization steps gives us a smaller relative error. However, as we can see in Figure (5), the maximum relative error spikes up to approximately 10^{11} as we reach $n_{\text{steps}} = 10^7$ which proves to us that the biggest number of discretization steps isn't necessarily the best.

Problem 9

To specialize our Tridiagonal matrix algorithm for special cases where the sub- and super diagonal have values -1 while the main diagonal has the values 2, we expand the the general algorithm from Problem 6. Knowing the initial values for a, b and c, we can reduce some algorithmic operations into constants. Since a and c equals -1, we can rewrite the operations as follows

$$\tilde{b}_i = b_i - \frac{a_i}{b_{i-1}} c_{i-1} = b_i - \frac{1}{b_{i-1}} \quad \text{for } i = 2, 3, \ldots, n$$
$$\tilde{g}_i = g_i + \frac{g_{i-1}}{b_{i-1}} \quad \text{for } i = 2, 3, \ldots, n$$

$$\mathrm{v}_n = \frac{\tilde{g}_n}{\tilde{b}_n}$$

$\mathrm{v}_i = \frac{\tilde{g}_i + v_{i+1}}{\tilde{b}_i}$, for $i = n-1, n-2, \ldots, 1$

We can write down the first few $\tilde{b}$ values: $\tilde{b}_1 = 2 \quad \tilde{b}_2 = 2 - \frac{1}{2} = \frac{3}{2} \quad \tilde{b}_3 = 2 - \frac{2}{3} = \frac{4}{3} \quad \tilde{b}_4 = 2 - \frac{3}{4} = \frac{5}{4} \quad . \quad . \quad . \quad \tilde{b}_i = \frac{i+1}{i}$ We ended up with an easier operation. In the previous operation, we had to subtract the inverse of b_{i-1} from b_i, this would have included one more arithmetic operation. Not only is the new formula n times faster, but its implementation would also be faster: in computing, indexing an array counts as a primitive operation. The new formula will avoid this, since it only operates with the value i.

FLOPs

With a and c being constants, we avoided 1 arithmetic operation in the calculation of $\tilde{b}$ and $\tilde{g}$.Further we wrote $\tilde{b}_i$ as an easier formula. In the table Table 3 we can see the amount of FLOPs for each operation.

Variable	**FLOPs**	**Accumulated**
$a_i.c_{i-1}$	1	1
$\tilde{b}_i$	$1(n-1)$	$n+2$
$\tilde{g}_i$	$2(n-1)$	$3n$
v_n	1	$3n+1$
v_i	$2(n-1)$	$5n-1$
	Total:	$5n-1$

Table 3: All the FLOPs required for the special algorithm per variable listed together with the accumulated number in the right-most column.

Problem 10

Now we want to compare the general algorithm from **??** with the special algorithm from Problem 9. The way we are going to do this is to use a timer inside the code to measure how long the algorithm takes to run. We do this for every $n_{\text{steps}} = \{10,\ 10^2,\ 10^3,\ 10^4,\ 10^5,\ 10^6\}$, and repeat the measure 6 times to get some data. Then we use the mean of the data to give us a good value, and the standard deviation to measure the fluctiations. The results can be found in Table (4)

n_{steps}	Special algorithm [s]	General algorithm [s]
10^1	0.000 ± 0.000	0.000 ± 0.000
10^2	0.000 ± 0.000	$7.717 \cdot 10^{-5} \pm 1.186 \cdot 10^{-4}$
10^3	$5.337 \cdot 10^{-4} \pm 1.035 \cdot 10^{-4}$	$7.160 \cdot 10^{-4} \pm 2.263 \cdot 10^{-4}$
10^4	$3.108 \cdot 10^{-3} \pm 4.239 \cdot 10^{-4}$	$5.546 \cdot 10^{-3} \pm 8.719 \cdot 10^{-4}$
10^5	$3.029 \cdot 10^{-2} \pm 2.327 \cdot 10^{-3}$	$5.733 \cdot 10^{-2} \pm 4.608 \cdot 10^{-3}$
10^6	$3.032 \cdot 10^{-1} \pm 2.257 \cdot 10^{-2}$	$5.745 \cdot 10^{-1} \pm 3.297 \cdot 10^{-2}$

Table 4: This table contains the time the two algorithms used for different choices of n_{steps}. Note that the time 0 does not mean that the algorithm runs instantly, but that it ran so fast that the timer wasn't able to measure the time.

Note that the time 0.000 does not mean that the algorithm runs instantly, but that it ran so fast that the timer wasn't able to measure the time. It would also be useful to us to compare the algorithms by making a table containing the ratios between the time-measurements. We will only make the table with means values, as computing the fluctuations will make us do extra calculations. We will take the ratio of the special algorithm over the general algorithm, or more specific

$$\text{ratio} = \frac{t_{\text{special}}}{t_{\text{general}}}$$

We will exclude the case of $n = 10$ as it will give us a $0/0$ expression. The ratios can be found in Table (5).

n	Ratio (special / general) [−]
10^2	0.000
10^3	0.745
10^4	0.560
10^5	0.528
10^6	0.527

Table 5: This table contains the ratios of the mean values for the special and general algorithms.

As we can read from Table (5) the special algorithm runs almost twice as fast as the general one, just by reducing some FLOPs.

Where to find the sourcode

All source code can be found in their respective folders in our github repository.