Project 1: Solving the 1D Poisson equation

Erica Løken[†]
Department of Physics, University of Oslo
(Dated: September 10, 2025)

In this project I will numerically solve the one-dimensional Poisson equation with Dirichlet boundary conditions. This is done by discretizing the continuous equation and rewriting the equation as a set of linear equations represented in matrix form. Since the resulting matrix is tridiagonal, the Thomas algorithm will be used to solve the system. Calculations of the absolute and relative errors reveal the importance of choosing a sufficiently large number of grid points n. For a general tridiagonal matrix, this algorithm uses approximately $8n$ floating-point operations (FLOPs), whereas in the special case where the matrix is tridiagonal with signature $(-1, 2, -1)$, as in the Poisson problem, the number of FLOPs reduces to $6n$. This shows the difference in efficiency when having constant diagonals. Lastly, the computation time for the general and the special algorithms are compared.

The one-dimensional Poisson equation is a second order differential equation and can be written as

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

where $f(x) = 100e^{-10x}$ is the sourcing term and $x \in [0, 1]$. The given boundary conditions are $u(0) = 0$ and $u(1) = 0$.

PROBLEM 1

We want to verify analytically that

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

is an exact solution to the Poisson equation (Eq. (1)). This is done by calculating the second derivative $\frac{d^2u}{dx^2}$ of $u(x)$ and verifying that this equals $-f(x)$ and verifying the boundary conditions. Firstly, we calculate the derivatives:

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

Thus, we can see that $\frac{d^2u}{dx^2} = -f(x)$ which shows that $u(x)$ satisfies the differential equation. Further, we check the boundary conditions:

$$u(0) = 1 - (1 - e^{-10})0 - e^0 = 1 - 1 = 0$$
$$u(1) = 1 - (1 - e^{-10}) - e^{-10} = 0$$

Consequently, $u(x)$ is an exact solution to Eq. (1) since this function satisfies the differential equation and the given boundary conditions.

PROBLEM 2

For this problem, I evaluated the exact solution $u(x)$ at $n = 10^5$ evenly spaced points in the interval $x \in [0, 1]$ using the C++ program `problem2.cpp`. The corresponding plot of the analytical solution is shown in Figure 1.

[†] GitHub: https://github.uio.no/ericalo/FYS3150

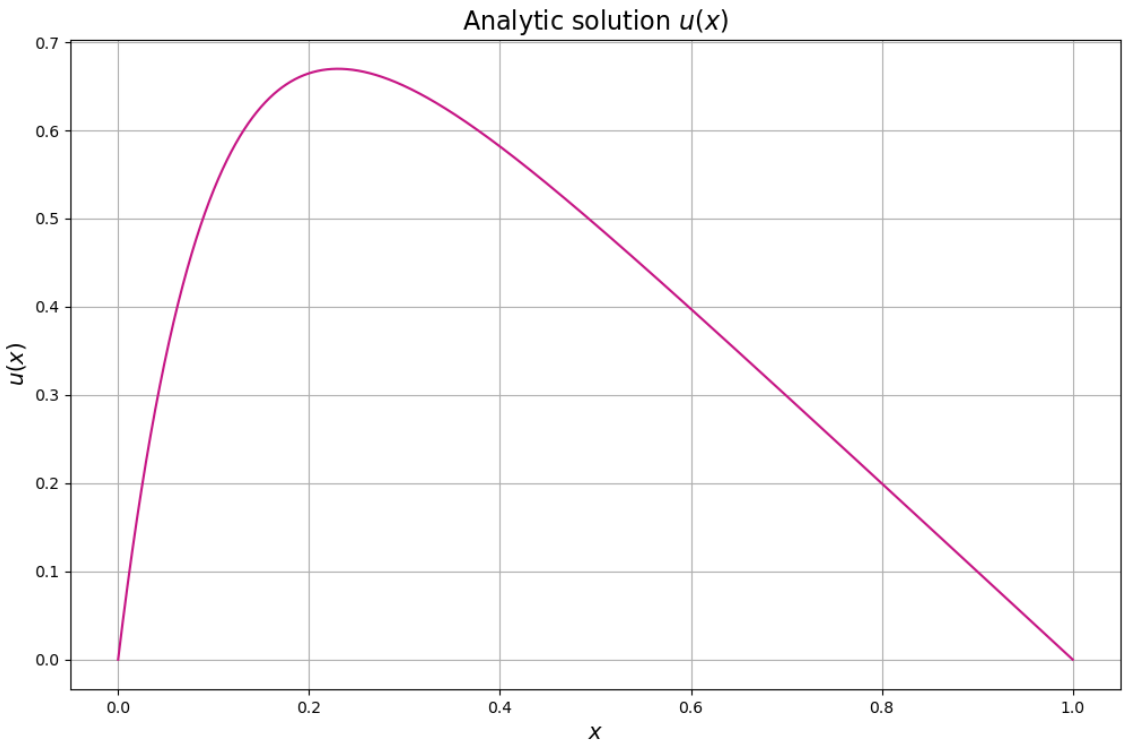

FIG. 1: Exact solution $u(x)$ from Eq. (2) evaluated at $n = 10^5$ points for $x \in [0, 1]$.

PROBLEM 3

In order to solve the Poisson equation numerically, we need to discretize and approximate the continuous differential equation. Since Eq. (1) is a continuous boundary value problem, we need to introduce some notation for our discretized variables:

$$\begin{aligned} x &\to x_i \\ f(x) &\to f(x_i) \equiv f_i \\ u(x) &\to u(x_i) \equiv u_i \\ u(x \pm h) &\to u(x_i \pm h) \equiv u_{i\pm 1} \end{aligned}$$

Where $h = \frac{x_{n+1}-x_0}{n+1}$ is the step size and $n+1$ is the number of steps. We will also define $x_i = x_0 + ih$, $i = 0, 1, 2, \ldots, n+1$. Since x goes from 0 to 1 in our problem, the step size is $h = \frac{1}{n+1}$ and $x = ih$. In order to distinguish our discretized approximation u_i from the exact value $u(x)$, we will define the numerical approximation $v_i \approx u_i$.

Since the Poisson equation contains a second derivative, we need a discrete approximation of the second derivative. This is done by taylor expanding $u(x)$ around a point x and solving for the derivative. I will use the expansion of u around the two points $x \pm h$:

$$u(x+h) = \sum_{n=0}^{\infty} \frac{1}{n!} u^{(n)}(x) h^n = u(x) + u'(x)h + \frac{1}{2}u''(x)h^2 + \frac{1}{6}u'''(x)h^3 + \mathcal{O}(h^4) \tag{3}$$

$$u(x-h) = \sum_{n=0}^{\infty} \frac{1}{n!} u^{(n)}(x)(-h)^n = u(x) - u'(x)h + \frac{1}{2}u''(x)h^2 - \frac{1}{6}u'''(x)h^3 + \mathcal{O}(h^4) \tag{4}$$

where $\mathcal{O}(h^4)$ is the truncation error proportional to h^4. Now adding the two expansions in Eq. (3) and (4) to get rid of the first and third derivative:

$$u(x+h) + u(x-h) = 2u(x) + u''(x)h^2 + \mathcal{O}(h^4)$$

and rearranging to get an expression for the centered finite difference approximation of the second derivative:

$$u''(x) = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} + \mathcal{O}(h^2) \tag{5}$$

where the order of the truncation error is reduced to be proportional to h^2. We discretize our continuous function $u(x)$ by introducing our new notation $u(x) \to u_i$ in Eq. (5):

$$u_i'' = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \mathcal{O}(h^2) \tag{6}$$

Thus, the Poisson equation can be written

$$-\left[\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \mathcal{O}(h^2)\right] = f_i, \qquad f_i \equiv f(x_i) \tag{7}$$

Now we approximate our discretized equation (Eq. (7)) by leaving out $\mathcal{O}(h^2)$ terms and changing notation to $v_i \approx u_i$. Finally, the discretized version of the Poisson equation is

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i \tag{8}$$

PROBLEM 4

From the discretized Poisson equation (Eq. (8)) and the boundary conditions $v_0 = v_{n+1} = 0$, we get n unknowns $v_1, v_2, \ldots, v_n$. To visualize, I will write down the first 3 equations:

$$\begin{aligned}
(i=1) \quad & -v_0 + 2v_1 - v_2 = h^2 f_1 \\
(i=2) \quad & -v_1 + 2v_2 - v_3 = h^2 f_2 \\
(i=3) \quad & -v_2 + 2v_3 - v_4 = h^2 f_3 \\
& \vdots \\
(i=n) \quad & -v_{n-1} + 2v_n - v_{n+1} = h^2 f_n
\end{aligned}$$

For $i = 1$ and $i = n$ the boundary terms v_0 and v_{n+1} are zero from the boundary conditions. Since these are known values, I will move them over to the right side and introduce some new notation. Explicitly, I will rewrite the three first equations once more:

$$\begin{aligned}
(i=1) \quad & 2v_1 - v_2 = h^2 f_1 + v_0 && \equiv g_1 \\
(i=2) \quad & -v_1 + 2v_2 - v_3 = h^2 f_2 && \equiv g_2 \\
(i=3) \quad & -v_2 + 2v_3 - v_4 = h^2 f_3 && \equiv g_3 \\
& \vdots \\
(i=n) \quad & -v_{n-1} + 2v_n = h^2 f_n + v_{n+1} && \equiv g_n
\end{aligned}$$

We recognize from linear algebra that this system of equations can be translated into the well known matrix equation

$$\mathbf{A}\vec{v} = \vec{g}, \qquad \vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{R}^n, \qquad \vec{g} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} \in \mathbb{R}^n, \tag{9}$$

where $\vec{v}$ is a vector containing the n unknowns and $\vec{g}$ is a vector containing the known values g_i. Writing this system on matrix form yields:

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_n \end{bmatrix} \tag{10}$$

where the matrix $\mathbf{A}$ is a $n \times n$ tridiagonal matrix with 2 on the main diagonal and -1 on the sub- and superdiagonal. Equivalently, $\mathbf{A}$ has a tridiagonal signature $(-1, 2, -1)$.

PROBLEM 5

We now define a vector $\vec{v}^*$ of length m that represents a complete solution to the discretized Poisson equation. The matrix $\mathbf{A}$ is an $n \times n$ matrix.

Problem a

From Problem 4 we know that the vector $\vec{v}$ containing the unknowns $v_1, v_2, \ldots, v_n$ satisfies $\mathbf{A}\vec{v} = \vec{g}$. That is, $\vec{v}$ holds the unknowns, but not the boundary values $v_0 = 0$ and $v_{n+1} = 0$. Since $\vec{v}^*$ holds the complete solution, the endpoints are included and it's length must be $m = n + 2$. Explicitly,

$$\vec{v}^* = \begin{bmatrix} 0 \\ v_1 \\ \vdots \\ v_n \\ 0 \end{bmatrix} \tag{11}$$

Problem b

Solving $\mathbf{A}\vec{v} = \vec{g}$ yields the unknowns $v_1, \ldots, v_n$, but not the complete solution. To obtain the complete solution $\vec{v}^*$, we include the boundary values $v_0 = v_{n+1} = 0$.

PROBLEM 6

In this problem $\mathbf{A}$ is a general tridiagonal matrix where the vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$ represents the subdiagonal, main diagonal and superdiagonal:

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & 0 \\ 0 & a_3 & b_3 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_{n-2} & 0 \\ \vdots & \ddots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & 0 & 0 & a_n & b_n \end{bmatrix} \tag{12}$$

Problem a

The alogrithm for solving $\mathbf{A}\vec{v} = \vec{g}$ is called the Thomas algorithm and is derived by Gaussian elimination. The algorithm is given by a forward substitution:

$$\begin{aligned} \tilde{b}_1 &= b_1 \\ \tilde{b}_i &= b_i - \frac{a_i}{\tilde{b}_{i-1}} c_{i-1} \qquad i = 2, \ldots, n \\ \tilde{g}_1 &= g_1 \\ \tilde{g}_i &= g_i - \frac{a_i}{\tilde{b}_{i-1}} \tilde{g}_{i-1} \qquad i = 2, \ldots, n \end{aligned}$$

and a back substitution:

$$\begin{aligned} v_n &= \frac{\tilde{g}_n}{\tilde{b}_n} \\ v_i &= \frac{\tilde{g}_i - c_i v_{i+1}}{\tilde{b}_i} \qquad i = n-1, \ldots, 1 \end{aligned}$$

I will refer to this algorithm as the general algorithm. The general algorithm is summarized in Algorithm 1.

Algorithm 1 General algorithm

$\tilde{b}_1 = b_1$
$\tilde{g}_1 = g_1$

Forward substitution:
for $i = 2, 3, ..., n$ **do**
  $d = a_i / \tilde{b}_{i-1}$
  $\tilde{b}_i = b_i - d * c_{i-1}$
  $\tilde{g}_i = g_i - d * \tilde{g}_{i-1}$

Back substitution:
$v_n = \tilde{g}_n / \tilde{b}_n$
for $i = n-1, \ldots 1$ **do**
  $v_i = (\tilde{g}_i - c_i v_{i+1}) / \tilde{b}_i$

Problem b

We want to count the number of FLOPs for the general algorithm. For the forward substitution, $\tilde{b}_1$ and $\tilde{g}_1$ has 0 FLOPs. For d there is 1 FLOP from the division. Further, $\tilde{b}_i$ and $\tilde{g}_i$ there are 2 FLOPs each from multiplying and subtracting. For each i this adds up to $1 + 2 + 2 = 5$ FLOPs $n - 1$ times, so $5(n-1)$ FLOPs. It is worth mentioning

that by defining $d = \frac{a_i}{b_{i-1}}$, we save n FLOPs in the forward substitution.

In the back substitution v_n has 1 FLOP from the division. The computation of v_i has 3 FLOPs repeated $n-1$ times, resulting in $3(n-1)$ FLOPs. In total, the back substitution has $1+3(n-1)$ FLOPs. In total this results in $5(n-1)+1+3(n-1) = 8n-7 \approx 8n$ FLOPs for the general algorithm.

PROBLEM 7

Problem a

I have implemented the general algorithm from Problem 6 in the C++ program `problem7.cpp` to solve $\mathbf{A}\vec{v} = \vec{g}$, where $\mathbf{A}$ is the tridiagonal matrix in Eq. (10). The program constructs the matrix $\mathbf{A}$ and the right hand side $\vec{g}$. The solution $\vec{v}$ is found by forward and back substitution and written to file along with the corresponding x-values. The code for plotting is found in the file `problem7.py`.

Problem b

Figure 2 shows the analytical solution $u(x)$ and the numerical solution $v(x)$ from using the general algorithm to solve Eq. (9) with $n_{steps} = [10, 10^2, 10^3, 10^5, 10^7]$. A zoomed in version of the plot is shown in Figure 3 to better show how the algorithm and the exact solution match up. We see that when n increases, the numerical solution gets closer to the analytical solution. When n exceeds 10^5, the numerical solution overshoots the analytical, but it is also of importance that I used $n = 10^5$ for the analytical solution. Either way, the algorithm and the analytical solution is a good match when calculated for the same number of points.

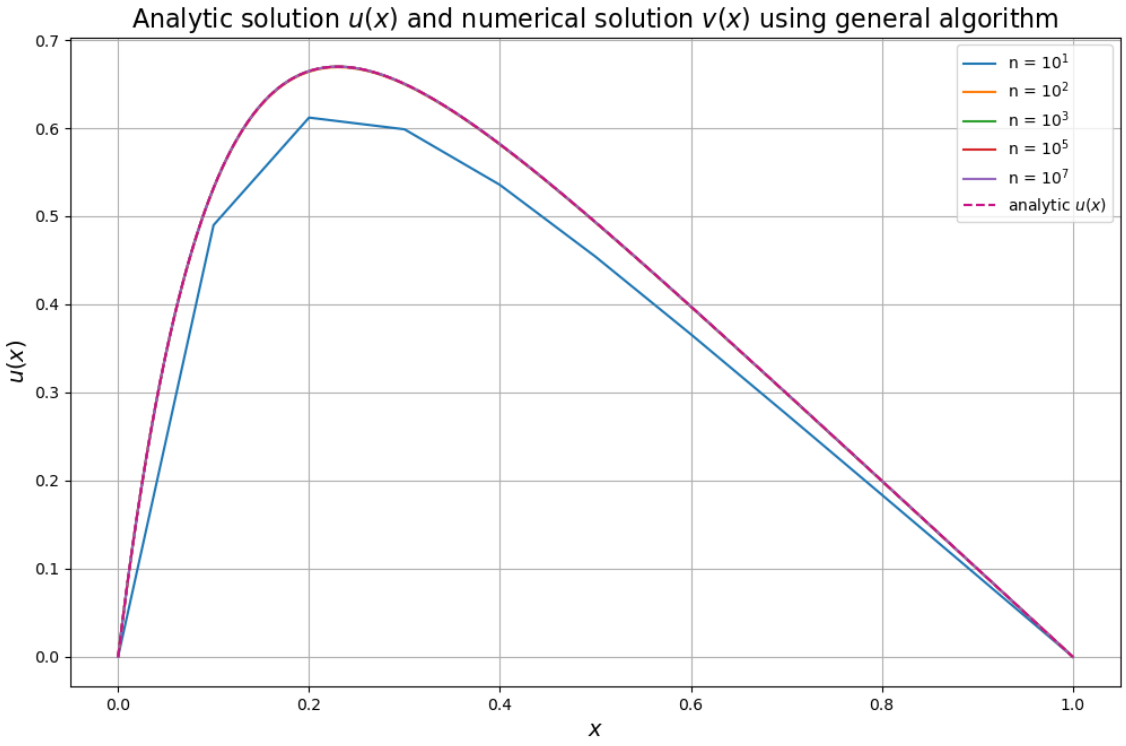

FIG. 2: Analytic solution $u(x)$ from Eq. (2) for $x \in [0,1]$ and $n = 10^5$, along with the numerical solution $v(x)$ computed with the general algorithm.

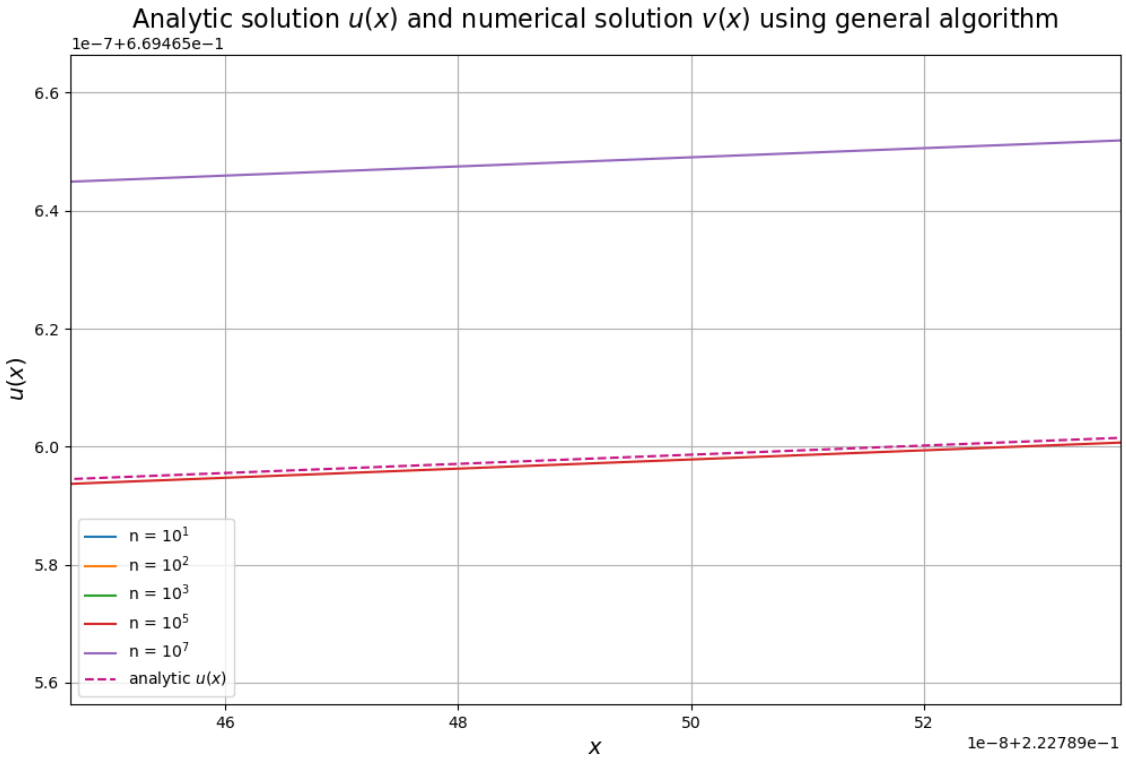

FIG. 3: Zoomed version of Figure 2. We see that the analytical and numerical solutions match very well for the same n-value.

PROBLEM 8

Problem a

Figure 4 shows the logarithm of the absolute error between the analytical solution $u(x)$ and the numerical solution $v(x)$, $\log_{10}(\Delta_i) = \log_{10}(|u_i - v_i|)$ for different grid sizes n. I have excluded the boundary points $u(0) = u(1) = 0$, since these should have zero error. As expected, we see that the absolute error decreases as n grows. Explicitly, $n = 10$ has an absolute error of order $10^{-1.5}$, while $n = 10^5$ has an absolute error of order 10^{-9}, which is a significant reduction. The sharp drops at the edges of the plot is due to the error being small near the endpoints, and 0 at $x = 0$ and $x = 1$ due to the boundary conditions. The code for plotting and error calculations is found in the file `problem8.py`.

For $n = 10^7$ the absolute error is larger than for $n = 10^5$, and we observe a dip at around $x \approx 0.25$. This behaviour is likely due to floating-point round off-errors. In general, increasing the number of grid points n improves accuracy, but at some point n will be too big and round-off errors will dominate. Therefore, we can conclude from the data that $n = 10^5$ is suitable for this problem by balancing accuracy and computational errors. Increasing n further will cost computationally and reduce accuracy.

Problem b

Figure 5 shows a plot of the logarithm of the relative error between the analytical solution $u(x)$ and the numerical solution $v(x)$, $\log_{10}(\epsilon_i) = \log_{10}\left(\left|\frac{u_i - v_i}{u_i}\right|\right)$. As expected, and consistent with the analysis of the absolute error, the relative error decreases when n increases. For $n = 10$ the relative error is large $\sim 10^{-1.5}$ while $n = 5$ gives and relative error of order 10^{-9}. We confirm our conclusion that 10^5 is a suitable choice of n for this problem, since $n = 10^7$ gives

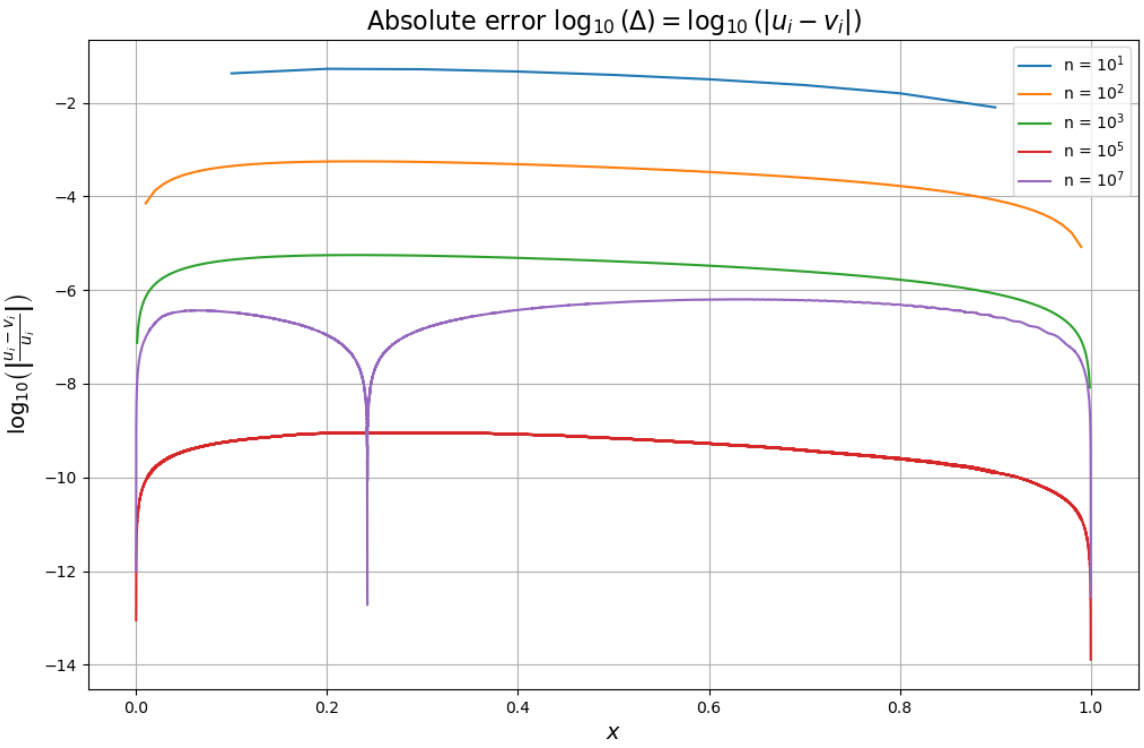

FIG. 4: Absolute error $\log_{10}(\Delta_i) = \log_{10}(|u_i - v_i|)$ of analytical value $u(x)$ and numerical value $v(x)$, excluding endpoints with 0 error.

an increased relative error and round-off errors dominates, while accuracy is reduced.

Problem c

Table I shows the maximum relative error $\max(\epsilon_i)$ for each choice of n_{steps}. We see that as n increases, the relative error decreases. $n = 10^5$ has the lowest maximum relative error of order 10^{-9}, while $n = 10^7$ has a maximum relative error of order 10^{-6}, reflecting the balance between discretization error and round-off errors at high values of n.

n_{steps}	$\max(\epsilon_i)$
10	7.93×10^{-2}
10^2	8.33×10^{-4}
10^3	8.33×10^{-6}
10^5	1.44×10^{-9}
10^7	2.98×10^{-6}

TABLE I: Maximum relative error $\max(\epsilon_i)$ for each n_{steps}.

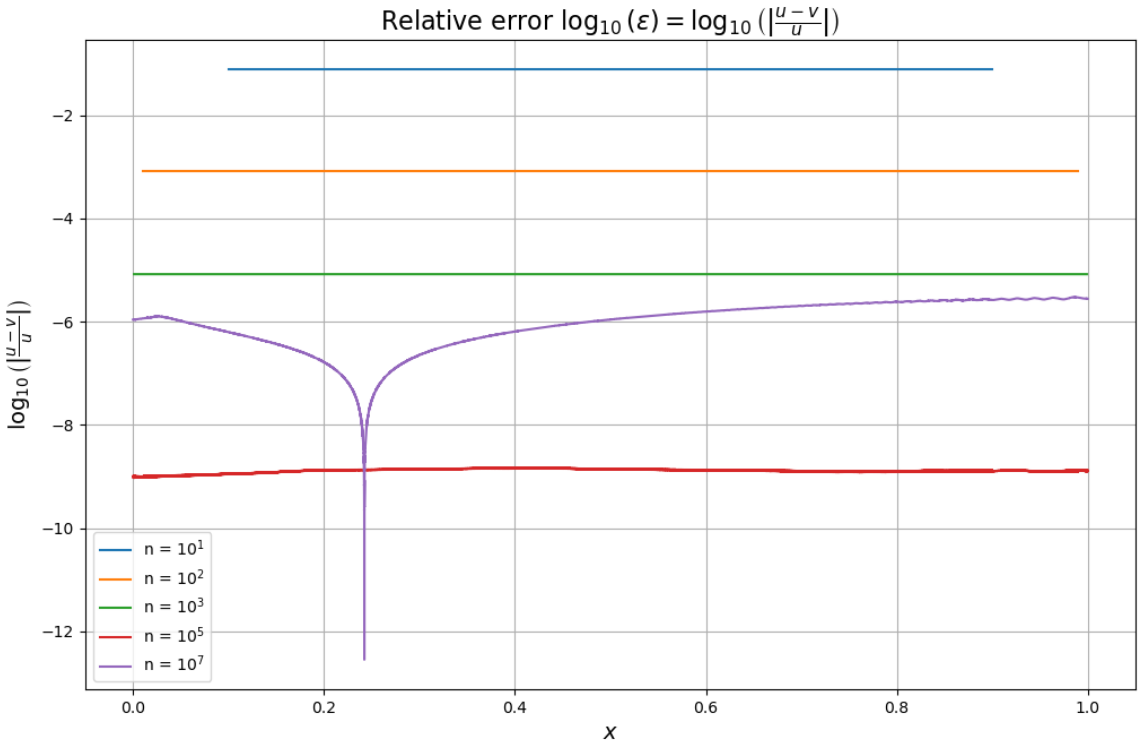

FIG. 5: Relative error $\log_{10}(\epsilon_i) = \log_{10}\left(\left|\frac{u_i - v_i}{u_i}\right|\right)$ of analytical value $u(x)$ and numerical value $v(x)$.

PROBLEM 9

Problem a

I will now specialize the general algorithm from Problem 6 to the special case where $\mathbf{A}$ is a tridiagonal matrix with signature $(-1, 2, -1)$. Explicitly, this corresponds to $\vec{a} = (-1, -1, \ldots, -1)$, $\vec{b} = (2, 2, \ldots, 2)$ and $\vec{c} = (-1, -1, \ldots, -1)$.

The general algorithm is listed in Algorithm 1. For this special case, $a_i = -1$, $b_i = 2$ and $c_i = -1$, which simplifies the forward substitution to:

$$\begin{aligned}
\tilde{b}_1 &= 2 \\
\tilde{b}_i &= 2 - \frac{1}{\tilde{b}_{i-1}} \qquad i = 2, \ldots, n \\
\tilde{g}_1 &= g_1 \\
\tilde{g}_i &= g_i + \frac{\tilde{g}_{i-1}}{\tilde{b}_{i-1}} \qquad i = 2, \ldots, n
\end{aligned}$$

and the back substitution:

$$v_n = \frac{\tilde{g}_n}{\tilde{b}_n},$$
$$v_i = \frac{\tilde{g}_i + v_{i+1}}{\tilde{b}_i}, \qquad i = n-1, \ldots, 1.$$

The special algorithm is summarized in Algorithm 2.

Algorithm 2 Special algorithm

$\tilde{b}_1 = 2$
$\tilde{g}_1 = g_1$

Forward substitution:
for $i = 2, 3, ..., n$ **do**
  $\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}}$
  $\tilde{g}_i = g_i + \frac{\tilde{g}_{i-1}}{\tilde{b}_{i-1}}$

Back substitution:
$v_n = \tilde{g}_n / \tilde{b}_n$
for $i = n-1, \ldots 1$ **do**
  $v_i = \frac{\tilde{g}_i + v_{i+1}}{\tilde{b}_i}$

Problem b

Now we want to count the number of FLOPs for the specialized algorithm. For the forward substitution, $\tilde{b}_i$ has 2 FLOPs from subtraction and division, and $\tilde{g}_i$ has 2 FLOPs from addition and division. This adds up to 2+2=4 FLOPs repeated $n-1$ times, in total $4(n-1)$ FLOPs.

The back substitution has 1 FLOP from the division in v_n. Inside the loop, there are 2 FLOPs from addition and division repeated $n-1$ times resulting in $1 + 2(n-1)$. In total for the whole algorithm, there are $4(n-1) + 1 + 2(n-1) = 6n - 5 \approx 6n$ FLOPs.

Problem c

I have implemented the special algorithm for solving the system $\mathbf{A}\vec{v} = \vec{g}$ in the C++ program `problem9.cpp`.

PROBLEM 10

Lastly, I will do run time tests for the general algorithm and the special algorithm to compare the computation time of the two algorithms. For reliable timing results, I will run each algorithm 100 times for each value of n_{steps} up to $n_{steps} = 10^6$. These test were done by the C++ program `problem10.cpp` and plotted by `problem10.py`.

Figure 6 shows the runtime of each algorithm across 100 runs for different values of n_{steps}. For small values of n_{steps} ($n = 10, 10^2, 10^3$), the runtime is short. When n_{steps} increases, the runtime increases. For $n = 10^6$, the runtime is significantly larger, and the special algorithm is consistently faster than the general algorithm. At run number ~ 60, a negative time was measured, but in the plot I have clipped it to be greater than zero. This result can be due to measurement noise or other factors.

For both algorithms, the first run is the slowest, and when n_{steps} increases, the variability in runtime increases. This can be due to CPU scheduling and factors related to the hardware itself, not the algorithm. In short, the special algorithm achieves a consistently shorter runtime than the general algorithm due to fewer FLOPs.

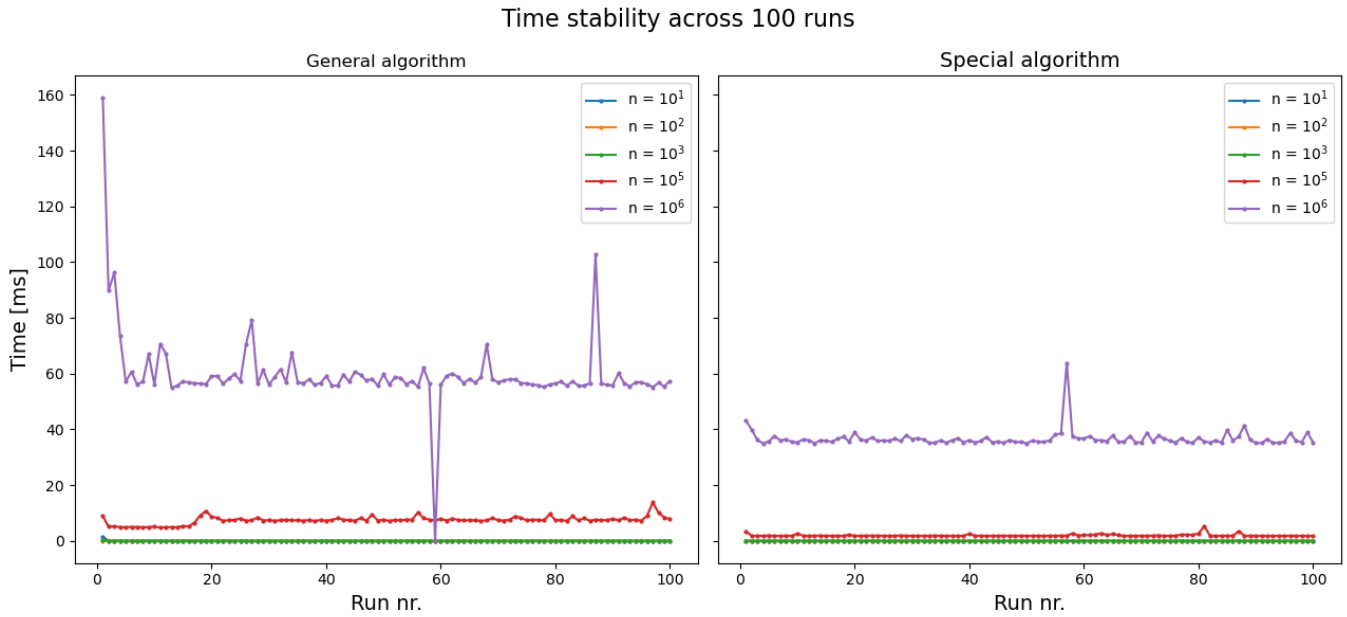

FIG. 6: Time stability as a function of run number across 100 runs for each value of n_{steps} for the general and the special algorithm. Negative times have been clipped away.

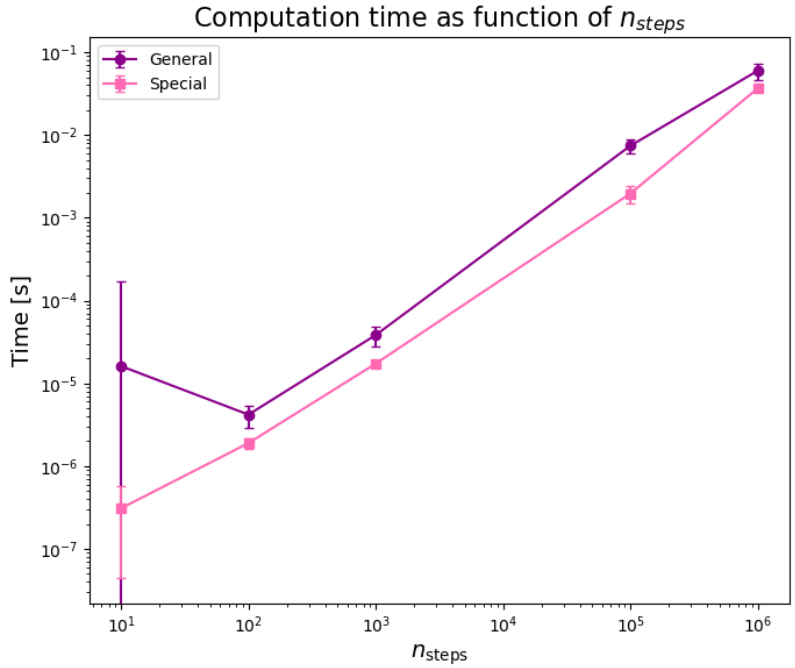

FIG. 7

Figure 7 shows an additional plot of the mean computation time as function of grid size n_{steps} for both algorithms. In agreement with Figure 6, the special algorithm is consistently faster than the general algorithm for all values of n_{steps}. The plot includes errorbars revealing the irregular behaviour in the first run and variability in runtime. Overall, the timing tests of both algorithms show that the special algorithm perfoms better than the general algorithm

due to lower computational complexity.