Project 1

Adrian Gundersen, Casper Johnsen & Victor B. Johansen
(Dated: September 22, 2025)

https://github.uio.no/adriangg/FYS3150/

INTRODUCTION

In this project we will look at linear one-dimensional differential equations. We will also look at numerical solutions to matrix systems like the one analogue to the Poisson differential equation with Dirichlet boundary conditions.

PROBLEM 1

We start with the Poisson differential equation with Dirichlet boundary condition.

$$\begin{cases} -u''(x) = f(x), & x \in [0,1], \\ f(x) = 100e^{-10x}, \\ u(0) = 0, \quad u(1) = 0. \end{cases} \tag{1}$$

We want to find an analytical solution starting from Equation 1.

$$\begin{aligned} & \frac{d^2}{dx^2}u = -100e^{-10x} \\ \implies & \frac{d}{dx}u = -\int dx\, 100e^{-10x} \\ \implies & u(x) = \int dx\, (10e^{-10x} + C) \\ \implies & u(x) = -e^{-10x} + Cx + D \end{aligned}$$

Now we have a general expression. To find the exact solution we can put in the boundary conditions:

$$\begin{aligned} & u(x) = -e^{-10x} + Cx + D \\ & u(0) = -e^0 + D = -1 + D = 0 \quad \implies \quad D = 1 \\ & u(1) = -e^{-10} + C \cdot 1 + 1 = 0 \quad \implies \quad C = e^{-10} - 1 \end{aligned}$$

Thus, the exact solution is

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

PROBLEM 2

Now we want to plot the solution to the solution from Equation 2. We make a script in C++ that generates a txt-file with x-values and the corresponding u-values. We then use this to plot in Python.

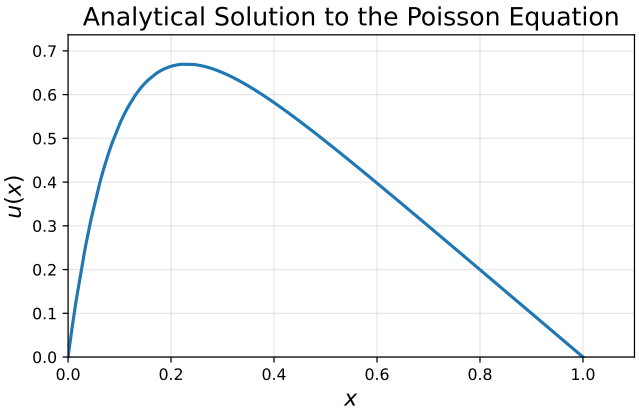

FIG. 1. Analytical solution to the Poisson equation. The x-axis is the domain and the y-axis denotes the solution given by Equation 2. The plot was made using Matplotlib-library in Python.

PROBLEM 3

We now want to look at a discretized version. We thus let $\mathbf{x} = (x_0, x_1, \ldots, x_{n+1})$. We want equal increments so we let $x_i = ih$ where $h = \frac{1}{n+1}$. Then we let $\mathbf{v}$ denote the approximate solution, where $v_i \approx u(x_i)$. Using the central finite difference for the double derivative this yields:

$$u''(x_i) \approx \frac{v_{i-1} - 2v_i + v_{i+1}}{h^2}, \quad \forall i \geq 1 \tag{3}$$

Putting this into the Poisson equation Equation 1 we get:

$$\frac{-v_{i-1} + 2v_i - v_{i+1}}{h^2} = f(x_i) \tag{4}$$

PROBLEM 4

Now we want to rewrite the equation in matrix form $A\mathbf{v} = \mathbf{x}$. We multiply Equation 4 by h^2 on both sides to get:

$$-v_{i-1} + 2v_i - v_{i+1} = f(x_i)h^2$$

As this holds for all i from 1 to N we can collect them to get:

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_n) \end{bmatrix} \tag{5}$$

This can also be written as:

$$A\mathbf{v} = \mathbf{g}, \quad \mathbf{g} = h^2\mathbf{f} \tag{6}$$

PROBLEM 5

Problem a and b

We now let $\mathbf{v}^* \in \mathbb{R}^m$ be the full solution to the problem. However the solution we give above is denoted as $\mathbf{v} \in \mathbb{R}^n$ As $\mathbf{v}^*$ is the complete solution with m points and $\mathbf{v}$ does not contain boundary points we have that $m = n+2$. The solution of $\mathbf{v}$ only contains the interior points of $\mathbf{v}$. This does not however matter as we can just apply the boundaries afterwards as they are fixed from Dirichlet boundary conditions.

PROBLEM 6

Now we want to utilize an algorithm for solving general tridiagonal linear problems as the one described by Equation 5, but with arbitrary diagonals. Now we let the matrix A have diagonals consisting of three vectors: $\mathbf{a}, \mathbf{b}, \mathbf{c}$ as such:

$$A\mathbf{v} = \mathbf{g}$$

where:

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_1 & b_2 & c_2 & \cdots & 0 \\ 0 & a_2 & b_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \cdots & 0 & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_n \end{bmatrix}$$

Writing this out we get:

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = g_i, \qquad i = 1, \ldots, n, \tag{7}$$

Doing row reduction one ends up with the Thomas Algorithm as described in the lecture notes [1]. We also want to find the floating-point operations (FLOPs). Each time we use one of the operations $+, -, \times, \div$, we count this as one FLOP and add to the total.

Algorithm 1 Thomas Algorithm for $A\mathbf{v} = \mathbf{g}$, where A is tridiagonal

Input: $\mathbf{a} = (a_1, \ldots, a_{n-1})$, $\mathbf{b} = (b_1, \ldots, b_n)$, $\mathbf{c} = (c_1, \ldots, c_{n-1})$, $\mathbf{g} = (g_1, \ldots, g_n)$
Output: $\mathbf{v} = (v_1, \ldots, v_n)$ such that $A\mathbf{v} = \mathbf{g}$

Forward elimination:

$\tilde{b}_1 = b_1$ ▷ 0 FLOPs
$k_i = \frac{a_{i-1}}{\tilde{b}_{i-1}}, \quad i = 2, 3, \ldots, n$ ▷ 1 FLOP per i. Total: $(n-1)$
$\tilde{b}_i = b_i - k_i \cdot c_{i-1}, \quad i = 2, 3, \ldots, n$ ▷ 2 FLOPs per i. Total: $2(n-1)$
$\tilde{g}_1 = g_1$ ▷ 0 FLOPs
$\tilde{g}_i = g_i - k_i \cdot g_{i-1}, \quad i = 2, 3, \ldots, n$ ▷ 2 FLOPs per i. Total: $2(n-1)$

Back substitution:

$v_n = \frac{\tilde{g}_n}{\tilde{b}_n}$ ▷ 1 FLOP for iteration n. Total: 1
$v_i = \frac{\tilde{g}_i - c_i v_{i+1}}{\tilde{b}_i}, \quad i = n-1, n-2, \ldots, 1$ ▷ 3 FLOPs per i. Total: $3(n-1)$

Handling boundary points:
Add the boundary conditions at x_0 and x_{n+1} given in the problem. ▷ 0 FLOPs

Now we want to find the total FLOPs:

$$\text{FLOP}_{\text{total}} = (n-1) + 2(n-1) + 2(n-1) + 1 + 3(n-1) = 8(n-1) + 1 = 8n - 7$$

PROBLEM 7

To implement Algorithm 1 we write it in C++ and run it for $n = 10, 10^2, 10^3, 10^4, 10^5$. We will later look at what happens when we increase the number of mesh points from $n = 10^5$.

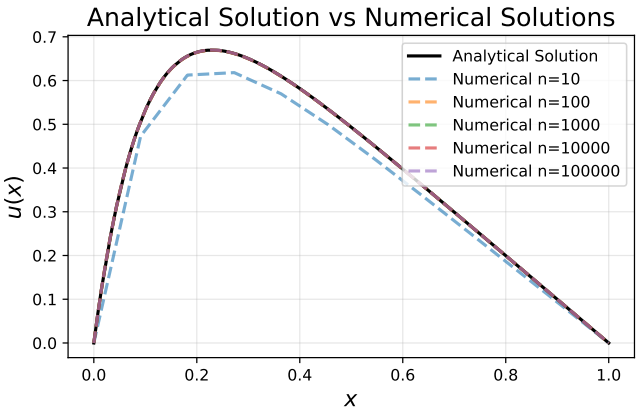

FIG. 2. Analytical solution from Equation 2 plotted against numerical solution for different mesh points n. The x-axis depicts the domain and the y-axis depicts the analytical/numerical value $u(x)$ or $v(x)$

PROBLEM 8

Visually we see in Figure 2 that for higher n the numerical solution approaches the exact one. However, we also want exact numbers to verify convergence. Thus we want to look at the difference between the exact and numerical solution for each n. Due to the big difference in magnitude we use a logarithmic-plot. We also remove the boundary points as they are fixed and not interesting to look at as they will always be equal.

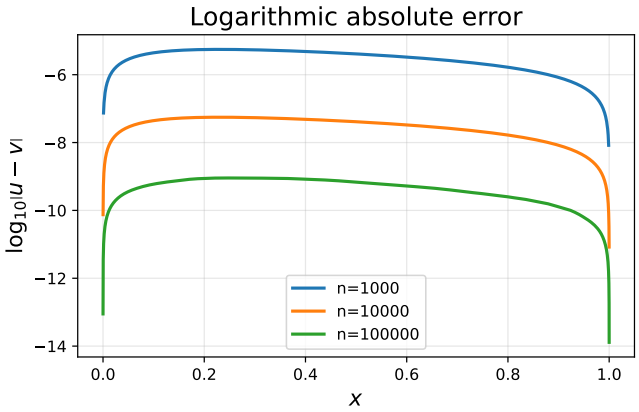

FIG. 3. Logarithm of the absolute error $|u - v|$ as a function of x for different grid resolutions n. For more grid points n we see a trend towards less error across the domain. Boundary points are omitted as they are fixed.

From Figure 3 we see that the difference becomes smaller for larger n, but also nearer the endpoints. Therefore, it is interesting to look at the relative error.

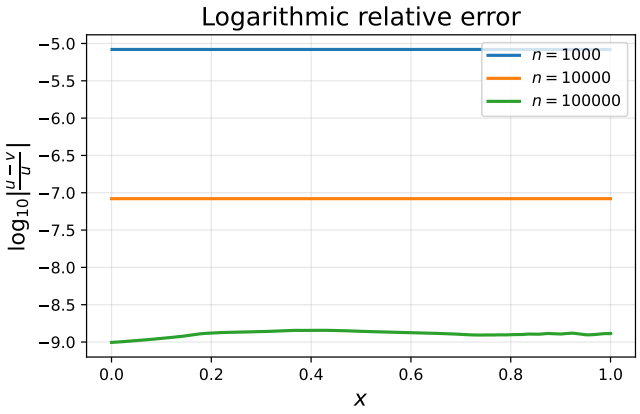

FIG. 4. Logarithm of the relative error $|(u - v)/u|$ as a function of x for different grid resolutions n. Higher resolutions yields lower relative error across the domain. Boundary points are omitted as they are fixed.

From Figure 4, we see that the relative error stays somewhat constant for each n. We also want to look at the maximum relative difference for higher values for n.

n	$\log_{10}(h)$	$\log_{10}\big(\max\lvert(u-v)/u\rvert\big)$
10	-1.041393	-1.179698
100	-2.004321	-3.088037
1000	-3.000434	-5.080052
10000	-4.000043	-7.079285
100000	-5.000004	-8.842973
1000000	-6.000000	-6.075474
10000000	-7.000000	-5.525230

TABLE I. Maximum relative error $\max|(u - v)/u|$ (logarithmic scale) as compared to the number of grid points n and step length h. Data obtained from `max_relative_error.txt`.

We see from Table I that it starts to follow the expected trend. Higher n yields lower absolute error. However, as we go above $n = 100000$ suddenly it goes in the other direction. This we can also see in a plot.

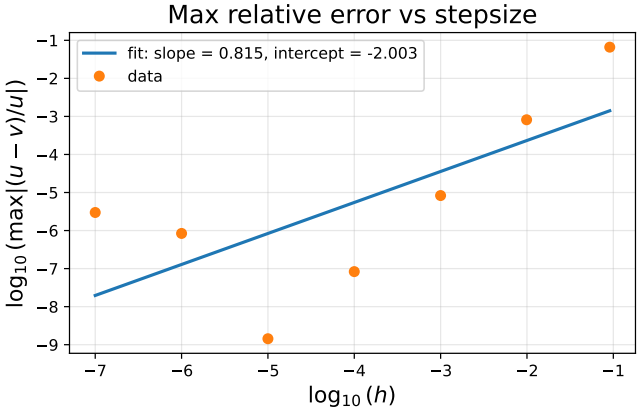

FIG. 5. The logarithm of maximum relative error $\max|(u - v)/u|$ as a function of the logarithm of stepsize h. A linear fit is included with slope ≈ 0.815.

Now, we see in Figure 5 that for higher n or smaller h we get a near linear trend, that suddenly stops at $h \approx 10^{-5}$. To see the linear trend we omit the points $h = 10^{-7}$ and $h = 10^{-6}$ to fit a linear trend where it may apply.

FIG. 6. The logarithm of maximum relative error $\max|(u-v)/u|$ as a function of the logarithm of stepsize h. A linear fit is included with slope ≈ 1.949. The datapoints are the same as in Figure 5

Now we see in Figure 6 that a linear model seems to fit the points very well up until $h = 10^{-5}$. So in addition to taking up enormous amount of storage (upwards of 1 GB) and a lot of time, too high n also gives worse results.

PROBLEM 9

However, due to the simplicity of the A-matrix in our problem, we can further reduce our problem. We know that the three diagonal vectors are all proportional to the 1-vector $\mathbf{1}_n = (1, 1, \ldots, 1) \in R^n$. We have the sub- and superdiagonal $\mathbf{a} = \mathbf{c} = (-1) \cdot \mathbf{1}_{n-1}$ and the diagonal vector $\mathbf{b} = 2 \cdot \mathbf{1}_n$. This means that we can reduce our number of FLOPs as we can make some variables constant from Algorithm 1. For instance $\frac{a_i}{c_{i-1}} = 1$ for all $i \geq 2$. Further, as $a_i = c_i = -1$ we do not need to use a FLOP to multiply this in but rather hardcode it. Doing this the algorithm reduces to:

Algorithm 2 Thomas Algorithm for $A\mathbf{v} = \mathbf{g}$, where A is tridiagonal

Input: $\mathbf{a} = (-1, \ldots, -1)$, $\mathbf{b} = (2, \ldots, 2)$, $\mathbf{c} = (-1, \ldots, -1)$, $\mathbf{g} = (g_1, \ldots, g_n)$
Output: $\mathbf{v} = (v_1, \ldots, v_n)$ such that $A\mathbf{v} = \mathbf{g}$

Forward elimination:
$\tilde{b}_1 = 2$ ▷ 0 FLOPs
$\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}}, \quad i = 2, 3, \ldots, n$ ▷ 2 FLOPs per i. Total: $2(n-1)$
$\tilde{g}_1 = g_1$
$\tilde{g}_i = g_i + \frac{g_{i-1}}{\tilde{b}_{i-1}}, \quad i = 2, 3, \ldots, n$ ▷ 2 FLOPs per i. Total: $2(n-1)$

Back substitution:
$v_n = \frac{\tilde{g}_n}{\tilde{b}_n}$ ▷ 1 FLOP for iteration n. Total: 1
$v_i = \frac{\tilde{g}_i + v_{i+1}}{\tilde{b}_i}, \quad i = n-1, n-2, \ldots, 1$ ▷ 2 FLOPs per i. Total: $2(n-1)$

Handling boundary points:
Add the Dirichlet boundary conditions at x_0 and x_{n+1} given in the problem. ▷ 0 FLOPs

Summing up we get the total FLOPs:

$$\text{FLOP}_{\text{total}}^{\text{reduced}} = 2(n-1) + 2(n-1) + 1 + 2(n-1) = 6n - 5$$

In total there are 6 FLOPs per iteration i, thus giving a 25% reduction from Algorithm 1. As n grows $\tilde{b}_i$ converges to 1, and there will be a limit where you can reduce the number of FLOPs when $\tilde{b}_i \approx 1$.

PROBLEM 10

Even though we have calculated a reduction of $\approx 25\%$ in FLOPs, we are also interested in finding the actual runtime-difference. We therefore time the different methods for different n-values and compare. However, we may have fluctuations in runtime so therefore we do repeated runs and compare the total time after 1000 runs.

TABLE II. Shows the total runtime after 1000 repeated runs for different n-values, comparing time between the Thomas-algorithm 1 and the reduced Thomas-algorithm 2.

n	Time original Thomas-Algorithm [s]	Time reduced Thomas-Algorithm [s]	Reduction [%]
10^1	$1.58 \cdot 10^{-4}$	$1.33 \cdot 10^{-4}$	15.46
10^2	$1.51 \cdot 10^{-3}$	$1.32 \cdot 10^{-3}$	12.81
10^3	$1.53 \cdot 10^{-2}$	$1.36 \cdot 10^{-2}$	10.55
10^4	$1.51 \cdot 10^{-1}$	$1.35 \cdot 10^{-1}$	11.01
10^5	$1.55 \cdot 10^{0}$	$1.35 \cdot 10^{0}$	12.72
10^6	$1.55 \cdot 10^{1}$	$1.33 \cdot 10^{1}$	13.77

For reproducibility you may need a powerful computer. We ran the same script on multiple computers and noticed that the reduction percentage varied from computer to computer in addition to from run to run. This is expected. However, something strange often happened around $n = 10^3$ or $n = 10^4$ for the weaker computers, where we suddenly got a negative reduction percentage implying that the optimized algorithm was slower. We are however unsure of the cause.

TABLE III. Shows the total runtime after 1000 repeated runs for different n-values, comparing time between the Thomas-algorithm 1 and the reduced Thomas-algorithm 2. This run was done on a weaker computer.

n	Time original Thomas-Algorithm [s]	Time reduced Thomas-Algorithm [s]	Reduction [%]
10^1	$8.11 \cdot 10^{-4}$	$7.08 \cdot 10^{-4}$	12.68
10^2	$2.18 \cdot 10^{-3}$	$4.58 \cdot 10^{-3}$	-110.14
10^3	$1.90 \cdot 10^{-2}$	$1.68 \cdot 10^{-2}$	11.62
10^4	$1.92 \cdot 10^{-1}$	$1.69 \cdot 10^{-1}$	11.95
10^5	$2.19 \cdot 10^{0}$	$1.74 \cdot 10^{0}$	20.53
10^6	$2.42 \cdot 10^{1}$	$2.07 \cdot 10^{1}$	14.37

Our main hypothesis is due to the structure of the processor and how it handles different types of FLOPs. This might lead to the vectors changing from L1 to L2 cache for $n = 10^3$. The original solver might line this better up with the CPU while the division in the reduced algorithm might be more heavy due to having to cache-transition. However on the stronger computer L2 is bigger and some FLOPs might thus be cheaper- This is however beyond the scope of our knowledge, but this is our best guess.

Without regarding this anomaly we see that the reduced algorithm has a tendency to be between $10\% - 20\%$ faster.

DECLARATION OF USE OF GENERATIVE AI

In this scientific work, generative artificial intelligence (AI) has been used. All data and personal information have been processed in accordance with the University of Oslo's regulations, and we, as the authors of the document, take full responsibility for its content, claims, and references. An overview of the use of generative AI is provided below.

Summary

- **Tool(s) used:** OpenAI ChatGPT (GPT-5), https://chatgpt.com/
- **Use:**
  - Generating boilerplate code like plotting with Matplotlib.
  - Checking language for clarity and grammar and general proof reading.

– Generating brief code documentation/comments.

– Creating tables in proper tex-format.

---

[1] A. Kveim and collaborators, "Fys3150 lecture notes," https://github.com/anderkve/FYS3150/tree/master/lecture_notes, accessed: 2025-09-10.