Project 1

Viktor Braaten Gulbrandsen, Katchi Senthil Ganesh
(Dated: November 19, 2026)

https://github.uio.no/viktorgu/fys3150

PROBLEM 1

The one-dimensional Poisson equation is given as

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

We are given $f(x) = 100e^{-10x}$ for $x \in [0,1]$ with $u(0) = 0$ and $u(1) = 0$. We thus run the integral

$$-\int \frac{d^2u}{dx^2}dx = \int 100e^{-10x} \tag{2}$$

We know the exponential integration formula $\int e^{ax}dx = \frac{1}{a}e^{ax} + C$ so

$$\begin{aligned}
-\int \frac{d^2u}{dx^2}dx &= 100\int e^{-10x}dx \\
-\int \frac{d^2u}{dx^2}dx &= 100(-\frac{1}{10}e^{-10x} + C) \\
\frac{du}{dx} &= 10e^{-10x} - 100C \\
\int \frac{du}{dx}dx &= -e^{-10x} - 100C_1x + C_2
\end{aligned}$$

We set up the boundary conditions:

$$\begin{aligned}
u(0) &= -e^{-10(0)} - 100C_1(0) + C_2 = 0 \\
u(0) &= -1 + C_2 = 0 \iff C_2 = 1 \quad \square \\
u(1) &= -e^{-10} - 100C_1 + C_2 = 0 \\
u(1) &= -e^{-10} - 100C_1 + 1 = 0 \\
100C_1 &= 1 - e^{-10} \\
C_1 &= \frac{1 - e^{-10}}{100} \quad \square.
\end{aligned}$$

This gives the final solution:

$$u(x) = -e^{-10x} - \frac{(1 - e^{-10})}{100} \cdot 100x + 1 = -e^{-10x} + (e^{-10} - 1)x + 1 = 1 - (1 - e^{-10})x - e^{-10x} \tag{3}$$

PROBLEM 2

The solution from Problem one is shown on Figure 1

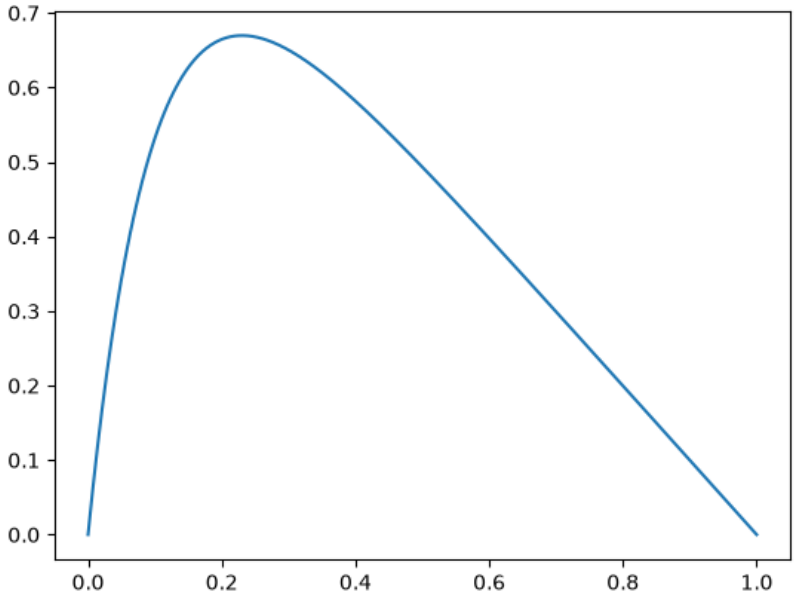

FIG. 1. Exact solution $u(x)$.

PROBLEM 3

We are now to get an expression for numerical differentiation.
The Taylor expansion up to order h^3 around x is given as:

$$u(x+h) = u(x) + u'(x)h + \frac{1}{2}u''(x)h^2 + \mathcal{O}(h^3) \tag{4}$$

$$u(x-h) = u(x) - u'(x)h + \frac{1}{2}u''(x)h^2 + \mathcal{O}(h^3) \tag{5}$$

Adding these two expansions, the first-derivative terms cancel:

$$u(x+h) + u(x-h) = 2u(x) + u''(x)h^2 + \mathcal{O}(h^4) \tag{6}$$

Solving for the second derivative gives the central difference approximation:

$$u''(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \tag{7}$$

PROBLEM 4

Consider those small intervals $x \pm h$ we use for the numerical approximation. These can also be indexed by i, forming the intuition of a grid $i = 1, \cdots, n-1$.

Denote the numerical approximation $\tilde{u}(x)$. Substituting the central difference into $-u''(x) = f(x)$ and multiplying by $-h^2$:

$$h^2 f(x_i) = -\tilde{u}(x_{i+1}) + 2\tilde{u}(x_i) - \tilde{u}(x_{i-1})$$

This is a system of $n-1$ linear equations. Thus it can be written in matrix notation:

$$Av = g \tag{8}$$

Let us first consider this for $n = 4$. For $i = 1$, we have

$$h^2 f(x_1) = -\tilde{u}(x_2) + 2\tilde{u}(x_1) - \tilde{u}(x_0), \tag{9}$$

for $i = 2$,

$$h^2 f(x_2) = -\tilde{u}(x_3) + 2\tilde{u}(x_2) - \tilde{u}(x_1), \tag{10}$$

and for $i = 3$,

$$h^2 f(x_3) = -\tilde{u}(x_4) + 2\tilde{u}(x_3) - \tilde{u}(x_2). \tag{11}$$

We see a general presentation

$$h^2 \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \end{pmatrix} = \begin{pmatrix} -\tilde{u}(x_2) + 2\tilde{u}(x_1) - \tilde{u}(x_0) \\ -\tilde{u}(x_3) + 2\tilde{u}(x_2) - \tilde{u}(x_1) \\ -\tilde{u}(x_4) + 2\tilde{u}(x_3) - \tilde{u}(x_2) \end{pmatrix} \tag{12}$$

Or more compact:

$$h^2 \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} \tilde{u}(x_1) \\ \tilde{u}(x_2) \\ \tilde{u}(x_3) \end{pmatrix} - \begin{pmatrix} \tilde{u}(x_0) \\ 0 \\ \tilde{u}(x_4) \end{pmatrix} \tag{13}$$

For any $n \in \mathbf{N}$, the discretization yields a tridiagonal matrix with signature $(-1, 2, -1)$ and zero elsewhere:

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} \tag{14}$$

Since the boundary conditions give $u(0) = 0$ and $u(x_n) = 0$, the boundary terms vanish and g is $g_i = h^2 f(x_i)$

PROBLEM 5

a) The complete solution of the vector contains the value of $\tilde{u}$ at every grid point, so has length m. The matrix, however, encompasses only the unknown equations and since we know the boundary conditions:

$$n = m - 2 \tag{15}$$

b) The unknowns. To get a full representation of the solution we append the boundary conditions

PROBLEM 6

a) Let us write down a general algorithm for solving a general tridiagonal matrix. Let A be a general $n \times n$ tridiagonal matrix, with subdiagonal $\vec{a} = (a_2, \ldots, a_n)$, main diagonal $\vec{b} = (b_1, \ldots, b_n)$ and superdiagonal $\vec{c} = (c_1, \ldots, c_{n-1})$, indexed by row:

$$A = \begin{pmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & & \vdots \\ 0 & a_3 & b_3 & \ddots & 0 \\ \vdots & & \ddots & \ddots & c_{n-1} \\ 0 & \cdots & 0 & a_n & b_n \end{pmatrix}. \tag{16}$$

To solve $A\vec{v} = \vec{g}$ we follow the Thomas algorithm as presented in the lecture notes [1], pp. 26–27. Here is our vectorized implementation of the algorthim:

```
std::vector<double> thomas_algo(...){
    int n = diagonal.size();

    for(int i = 1; i < n; i++){
        diagonal[i] = diagonal[i] - (subdiagonal[i] / diagonal[i-1]) * superdiagonal[i-1];
        g[i] = g[i] - (subdiagonal[i] / diagonal[i-1]) * g[i-1];
    }
    g[n-1] = g[n-1] / diagonal[n-1];
    for(int i = n-2; i >= 0; i--){
        g[i] = (g[i] - superdiagonal[i] * g[i+1]) / diagonal[i];
    }
    return g;
}
```

b)
A FLOP is a floating-point operation. That is, addition, multiplication, subtraction or divison on `double`. There are two parts of this calculation for this algorithm. Concerning the forward loop we see diagonal update takes 3 operations, and respectively the update of g takes 3 operations. The backward loop uses 3 operations. It is important to note that both of these loop over $n-1$ rows, and the last divison for the last row. This gives

$$N_{\text{FLOPs}} = 6(n-1) + 3(n-1) + 1 = 9n - 8 \approx 9n. \tag{17}$$

PROBLEM 7

a) The general algorithm from Problem 6 is used to solve $\mathbf{A}\vec{v} = \vec{g}$ for the tridiagonal matrix. The algorithm is implemented under the folder named `problem7` in the Github repository. There is a more robust version using vectors, and a more interpretable using `armadillo`. With n_{steps} steps the grid spacing is $h = 1/n_{\text{steps}}$ and there are $n = n_{\text{steps}} - 1$ interior points. The program writes the solution $\vec{v}$ and the corresponding $\vec{x}$ to file, including the two boundary points $u(0) = u(1) = 0$.
b) The numerical solution for $n_{\text{steps}} = 10, 100, 1000, 10000$ and the exact solution are shown in Figure 2.

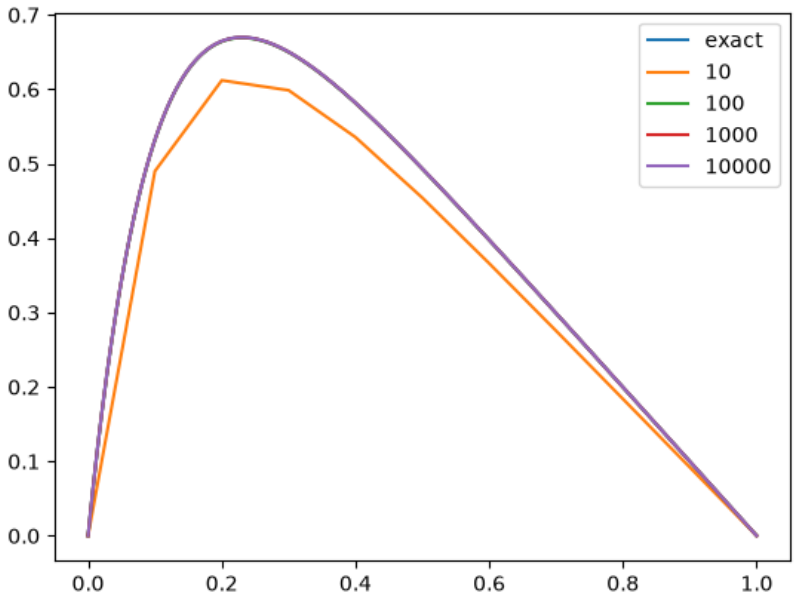

FIG. 2. Numerical solution for $n_{\text{steps}} = 10, 100, 1000, 10000$ and the exact solution.

PROBLEM 8

The errors are computed in `plot_errors.py` from the data files, using 15 digits in the output. The boundary points are left out.

a)

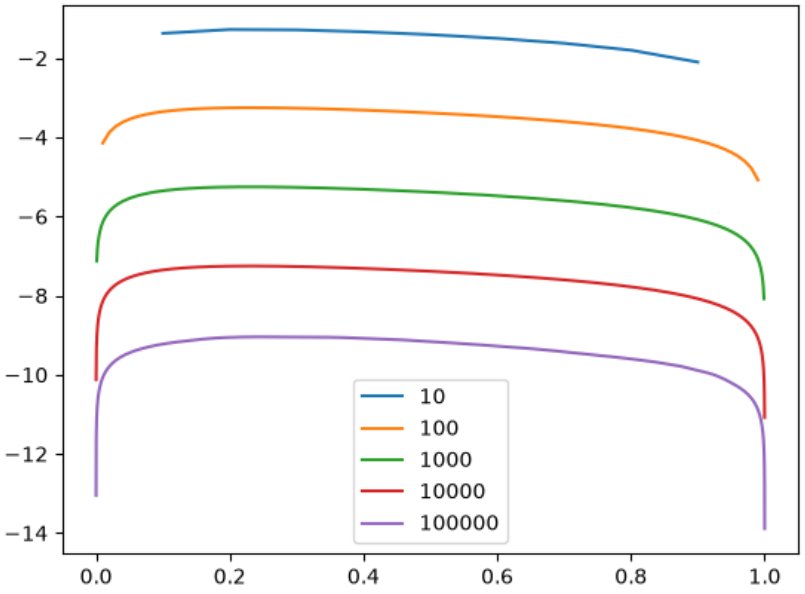

FIG. 3. $\log_{10}|u_i - v_i|$ for different n_{steps}.

b)

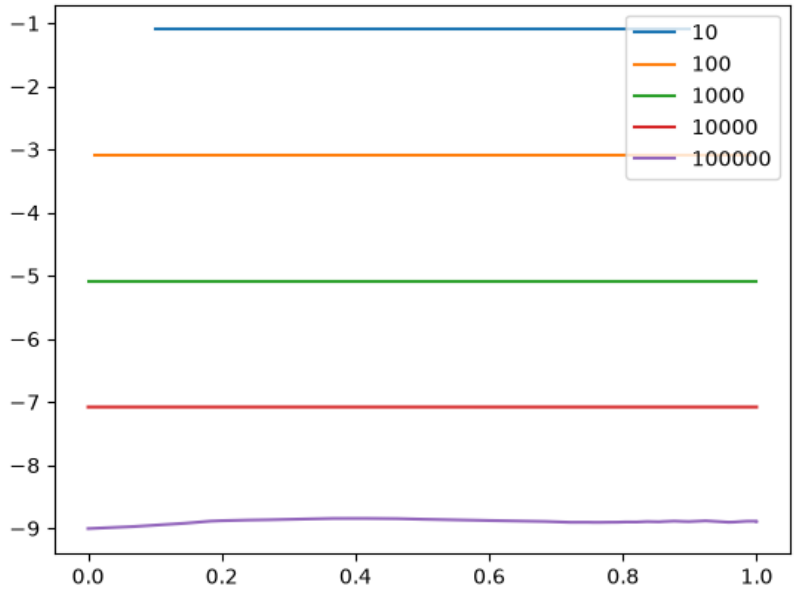

FIG. 4. $\log_{10}|(u_i - v_i)/u_i|$ for different n_{steps}.

As seen in Figure 3, the absolute error follows the shape of $u(x)$, while the relative error is almost constant in x. Both drop by a factor 100 each time n_{steps} increases by 10, as expected from the h^2 truncation error.

c)

The table is given in Table I.

TABLE I. Maximum relative error $\max_i \epsilon_i$ as a function of n_{steps}.

n_{steps}	$\max_i \epsilon_i$
10	7.93×10^{-2}
10^2	8.33×10^{-4}
10^3	8.33×10^{-6}
10^4	8.33×10^{-8}
10^5	1.44×10^{-9}
10^6	8.40×10^{-7}
10^7	2.98×10^{-6}

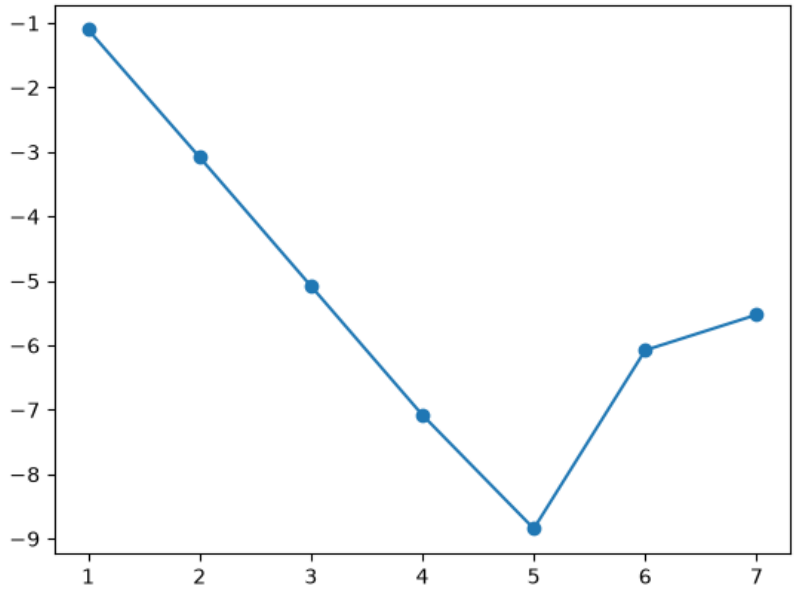

FIG. 5. $\log_{10}$ of the maximum relative error against $\log_{10} n_{\text{steps}}$.

In Figure 5, the error decreases as h^2 up to $n_{\text{steps}} = 10^5$, then increases again. For smaller h the round-off error from dividing by h^2 becomes larger than the truncation error, so the best result is at $n_{\text{steps}} \approx 10^5$.

PROBLEM 9

a) For the Poisson equation the matrix has signature $(-1, 2, -1)$, so $a_i = c_i = -1$ and $b_i = 2$ for all i. Inserting this into the general algorithm, the multiplications by -1 become sign changes and the subdiagonal and superdiagonal vectors are no longer needed. The forward substitution becomes

$$\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}}, \tag{18}$$

$$\tilde{g}_i = g_i + \frac{\tilde{g}_{i-1}}{\tilde{b}_{i-1}}, \tag{19}$$

with $\tilde{b}_1 = 2$, and the backward substitution becomes

$$v_i = \frac{\tilde{g}_i + v_{i+1}}{\tilde{b}_i}, \tag{20}$$

with $v_n = \tilde{g}_n / \tilde{b}_n$.

b) The forward loop now uses 2 operations for the diagonal and 2 for g, so 4 FLOPs per row over $n-1$ rows. The backward loop uses 2 per row over $n-1$ rows, plus one division for the last row. In total

$$N_{\text{FLOPs}} = 4(n-1) + 2(n-1) + 1 = 6n - 5 \approx 6n, \tag{21}$$

compared to $9n$ for the general algorithm.

c) The implementation only takes $\vec{g}$ as input:

```
std::vector<double> thomas_algo_special(std::vector<double> g){
    int n = g.size();
    std::vector<double> diagonal(n);
```

```
    diagonal[0] = 2.0;
    for(int i = 1; i < n; i++){
        diagonal[i] = 2.0 - 1.0 / diagonal[i-1];
        g[i] = g[i] + g[i-1] / diagonal[i-1];
    }
    g[n-1] = g[n-1] / diagonal[n-1];
    for(int i = n-2; i >= 0; i--){
        g[i] = (g[i] + g[i+1]) / diagonal[i];
    }
    return g;
}
```

We checked that it gives the same solution as the general algorithm for $n_{\text{steps}} = 100$ (maximum difference 0).

PROBLEM 10

Both algorithms are timed in `timing.cpp` using `<chrono>`, averaged over 20 runs for each n_{steps},

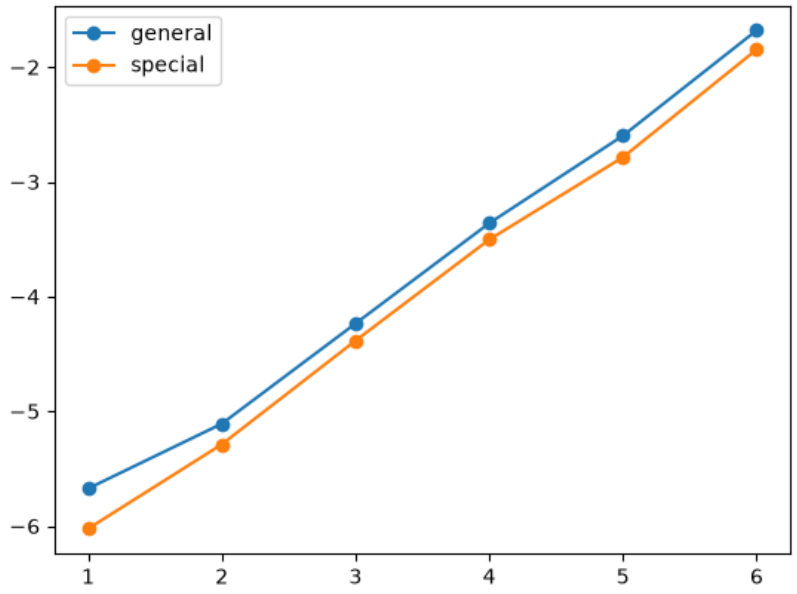

FIG. 6. Average run time of the general and special algorithm against n_{steps}, log-log.

Both algorithms scale linearly with n_{steps}. The two curves are approximately parallel, so ten times the steps yield ten time the time. The special algorithm is about 1.5 times faster, which matches the ratio of FLOP counts.

---

[1] A. Kvellestad, *FYS3150 Lecture Notes*, v1.20.1 (2026), https://github.com/anderkve/FYS3150/blob/master/lecture_notes/2026/lecture_notes.pdf