# FYS4150 – Project 1

Aline, Even Tobias and Vilde
(Dated: September 13, 2022)

*GitHub:* https://github.com/ungvilde/FYS4150/tree/main/Project%201

## PROBLEM 1

We have the one-dimensional Poisson equation, given by

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

where $f(x) = 100e^{-10x}$ and $x \in [0,1]$. The boundary conditions of this problem are given by $u(0) = u(1) = 0$. We want to check analytically that an exact solution to equation (1) is given by

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

First, we find the second derivative of (2):

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

Second, we check if the boundary conditions hold:

$$\begin{aligned}u(0) &= 1 - \left(1 - e^{-10}\right)\cdot 0 - e^{-10\cdot 0} \\ &= 1 - 0 - 1 = 0, \\ u(1) &= 1 - \left(1 - e^{-10}\right)\cdot 1 - e^{-10\cdot 1} \\ &= 1 - 1 + e^{-10} - e^{-10} = 0\end{aligned}$$

From this, we see that the conditions for (1) are met by (2).

## PROBLEM 2

We developed a C++ program that evaluates the exact solution (2) and writes the values to a text file, letting us easily plot the resulting data set. Figure 1 shows a plot of the exact solution.

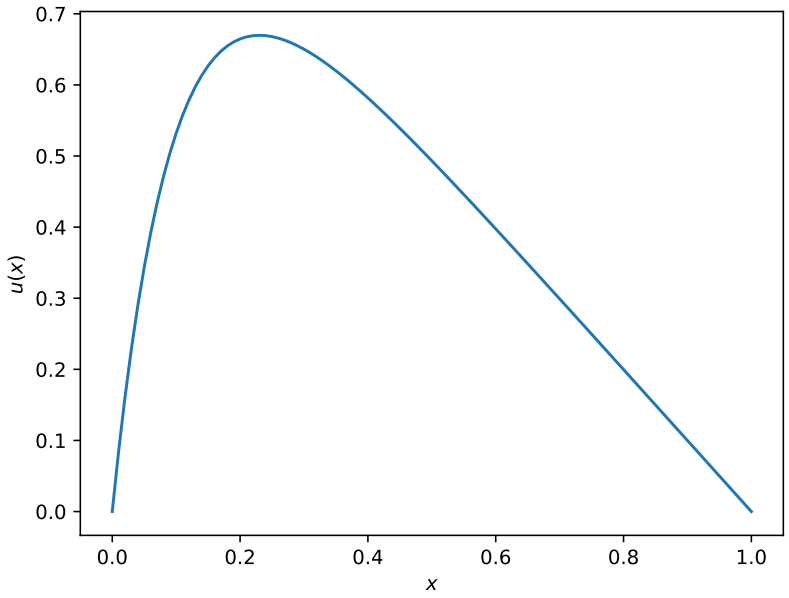


Figure 1. The exact solution (2) of the one-dimensional Poisson equation (1).

## PROBLEM 3

We want to derive a discretized approximation of (1). The Taylor expansion of $u$ at $x_0+h \in [0,1]$ and $x_0-h \in [0,1]$ is given by:

$$\begin{aligned} u(x_0+h) &= \sum_{n=0}^{\infty} \frac{h^n}{n!} u^{(n)}(x_0) \\ &= u(x_0) + hu'(x_0) + \frac{1}{2}h^2u''(x_0) + \frac{1}{6}h^3u'''(x_0) + \mathcal{O}(h^4) \\ u(x_0-h) &= \sum_{n=0}^{\infty} \frac{(-h)^n}{n!} u^{(n)}(x_0) \\ &= u(x_0) - hu'(x_0) + \frac{1}{2}h^2u''(x_0) - \frac{1}{6}h^3u'''(x_0) + \mathcal{O}(h^4) \end{aligned}$$

We can add the expressions above to get an expression for $u''(x_0)$:

$$\begin{aligned} u(x_0+h) + u(x_0-h) &= 2u(x_0) + h^2u''(x_0) + \mathcal{O}(h^4) \\ \iff u''(x_0) &= \frac{u(x_0-h) - 2u(x_0) + u(x_0+h)}{h^2} + \mathcal{O}(h^2) \end{aligned} \tag{3}$$

By letting $h$ be a sufficiently small number, we can use this to approximate the value of $u''(x_0)$. To discretize the problem, we partition the interval $[0,1]$ into $N+1$ equidistant points, and set $h = 1/(N+2)$. Let $x_i = i \cdot h$ for $i = 0, 1, ..., N+1$. Now let $v_i \approx u(x_i)$ and $f_i = f(x_i)$. Note that the boundary conditions for this problem imply that $v_0 = v_{N+1} = 0$. From (3), we have:

$$-\left(\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + \mathcal{O}(h^2)\right) = f_i,$$

which we approximate as

$$-\left(\frac{v_{i-1}-2v_i+v_{i+1}}{h^2}\right)=f_i.$$

Rearranging this expression finally gives us:

$$-v_{i-1}+2v_i-v_{i+1}=h^2f_i \tag{4}$$

for $i=1,\ldots,N$.

## PROBLEM 4

The results from the previous exercise show that the problem can be expressed as a set of $N$ linear equations. We let $g_i=h^2f_i$ and set our boundary conditions to be 0. Equation (4) can then be written as equation (5).

$$\begin{aligned}
-v_0+2v_1-v_2&=g_1\\
-v_1+2v_2-v_3&=g_2\\
&\vdots\\
-v_{N-2}+2v_{N-1}-v_N&=g_{N-1}\\
-v_{N-1}+2v_N-v_{N+1}&=g_N.
\end{aligned} \tag{5}$$

Note again that the $v_0$ and $v_{N+1}$ terms are zero. The above set of linear equations can be expressed as a matrix equation, where there are $N$ columns, and the elements of column $i$ contain the coefficients of $v_i$ for $i=1,2,...,N$. That is, let $\mathbf{A}$ be an $N\times N$ matrix given by:

$$\mathbf{A}=\begin{bmatrix}
2 & -1 & 0 & 0 & \dots & 0 & 0 & 0 & 0\\
-1 & 2 & -1 & 0 & \dots & 0 & 0 & 0 & 0\\
0 & -1 & 2 & -1 & \dots & 0 & 0 & 0 & 0\\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots\\
0 & 0 & 0 & 0 & \dots & -1 & 2 & -1 & 0\\
0 & 0 & 0 & 0 & \dots & 0 & -1 & 2 & -1\\
0 & 0 & 0 & 0 & \dots & 0 & 0 & -1 & 2
\end{bmatrix} \tag{6}$$

This is a tri-diagonal matrix, where the elements of the upper and lower diagonals are all $-1$, and the elements of the main diagonal are all 2. Let $\mathbf{v}=(v_1,v_2,...,v_N)$ and $\mathbf{g}=(g_1,g_2,...,g_N)$ be column vectors. The set of linear equations can be written in matrix form as:

$$\mathbf{Av}=\mathbf{g}. \tag{7}$$

## PROBLEM 5

a) The matrix $\mathbf{A}$ is an $n\times n$-matrix, which means there are $n$ solutions to equation (7). However, in order to find the complete solution of the discretized equation, we have to take into account the boundary conditions $u(0)=u(1)=0$, which will add two solutions. Therefore, the complete solution $\vec{v}^*$ is a vector of lenght $m=n+2$.

b) When we solve equation (7) for $\vec{v}$, we only get $n$ solutions. By inserting the boundary conditions as well we find the complete solution $\vec{v}^*$ of length $m=n+2$.

## PROBLEM 6

For this problem we will consider a *general* tridiagonal matrix $\mathbf{A}$ defined by vectors $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$ representing the lower, main, and upper diagonal of $\mathbf{A}$, respectively. The fact that $\mathbf{A}$ is a general tridiagonal matrix means that the elements of $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$ can be nonequal. Explicitly, the matrix will have the form:

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & a_{N-2} & b_{N-2} & c_{N-2} & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & a_N & b_N \end{bmatrix}.$$

a) A general algorithm for solving a matrix equation of the form:

$$\mathbf{Av} = \mathbf{g},$$

where $\mathbf{A}$ is tridiagonal, can be found by applying Gaussian elimination. Let $\mathbf{A}$ be an $N \times N$-matrix. Then $\mathbf{a}$ and $\mathbf{c}$ will each have $N-1$ elements, while $\mathbf{b}$ will have $N$ elements. In our algorithm, we will not include the entire matrix $\mathbf{A}$, but rather arrays of $\mathbf{a} = (a_2, a_3, \dots, a_N)$, $\mathbf{b} = (b_1, b_2, \dots, b_N)$ and $\mathbf{c} = (c_1, c_2, ...c_{N-1})$.

**Algorithm 1** General algorithm for solving $\mathbf{Av} = \mathbf{g}$

**for** $i = 2, 3, \dots, N$ **do** ▷ First we do forward substitution
  $\alpha \leftarrow \frac{a_i}{b_{i-1}}$ ▷ 1 FLOP
  $b_i \leftarrow b_i - \alpha \cdot c_{i-1}$ ▷ 2 FLOPs
  $g_i \leftarrow g_i - \alpha \cdot g_{i-1}$ ▷ 2 FLOPs
**end for**
$v_N \leftarrow \frac{g_N}{b_N}$ ▷ 1 FLOP
**for** $i = N-1, N-2, \dots, 1$ **do** ▷ Then we do backward substitution
  $v_i \leftarrow \frac{g_i - c_i \cdot v_{i+1}}{b_i}$ ▷ 3 FLOPs
**end for**

b) In Algorithm 1 presented above, we do a total number of $5(N-1) + 3(N-1) + 1 = 8N - 7$ floating-point operations (FLOPs). See the comments in Algorithm 1 for details.

## PROBLEM 7

a) Using the general algorithm found in problem 6, we can solve the matrix equation $\mathbf{Av} = \mathbf{g}$ as found in (7) by writing a C++ program that writes the output $\vec{v}$ and the corresponding $\vec{x}$ value to a file.

b) Defining $n_{\text{steps}}$ as the number of discretization steps along the full x-axis, we can run our program for $n_{\text{steps}} = 10, 100, 1000$ and compare these to the exact solution in a plot, as shown in figure 2.

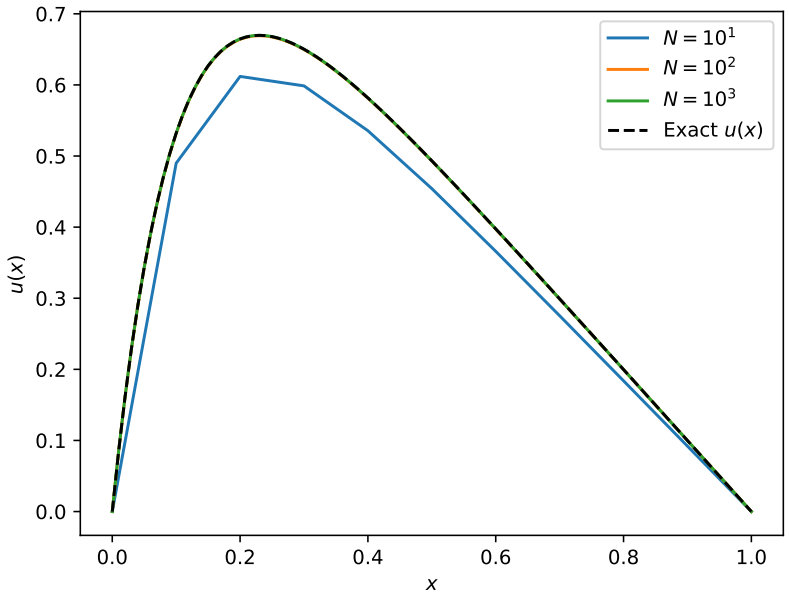


Figure 2. The figure shows the numerical approximation together with the exact solution of $u(x)$. The approximated values where computed using the general algorithm presented in Algorithm 1. We have included the numerical approximation found using $N = 10, 100, 1000$ steps along the $x$-axis.

## PROBLEM 8

a) We can make a plot of the logarithm of the absolute error

$$\log_{10}(\Delta_i) = \log_{10}(|u_i - v_i|)$$

as a function of $x_i$. The results are shown in figure 3.

b) Similarly we can plot the relative error

$$\log_{10}(\epsilon_i) = \log_{10}\left(\left|\frac{u_i - v_i}{u_i}\right|\right)$$

for different $x_i$ and $n_{\text{steps}}$, as shown in figure 4.

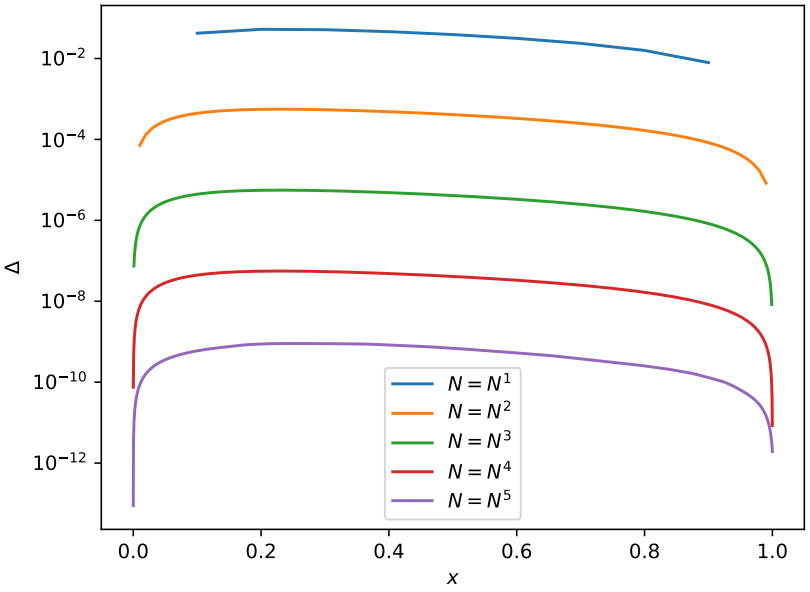


Figure 3. The logarithm of the absolute error $\Delta_i$ as a function of $x_i$, plotted for different $n_{\text{steps}}$.

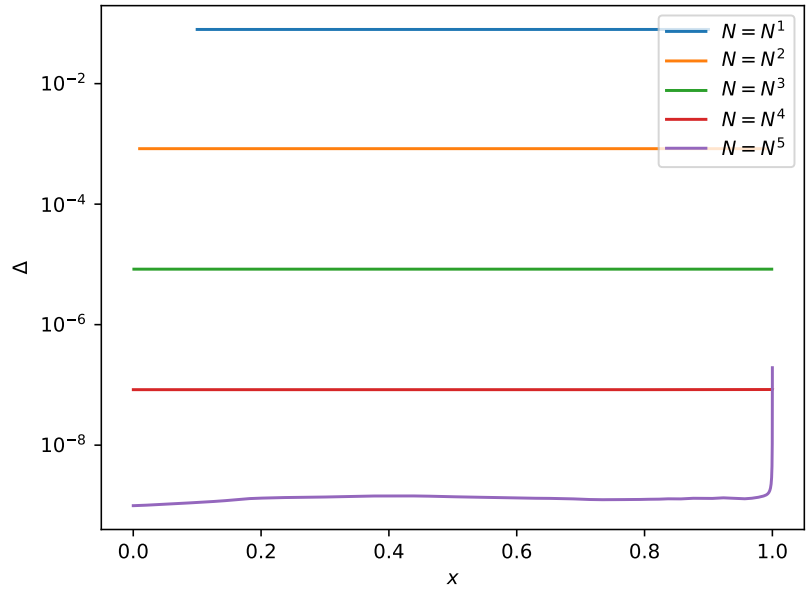


Figure 4. The relative error $\epsilon_i$ as a function of $x_i$, plotted for a selection of different $n_{\text{steps}}$.

c) Table (I) shows the maximum values for the relative error $\epsilon_i$ for different $n_{\text{steps}}$.

We see that as $n_{\text{steps}}$ increases, the relative error decreases since truncation error becomes less of a problem. At $n_{\text{steps}} = 10^4$, we have the lowest value for the maximum relative error, which indicates that a $n_{\text{steps}} = 10^4$ is the ideal amount of steps for this numerical calculation. Further down the table, the error begins to increase again, which is due to the step size being too small, thus creating bigger round-off errors.

| Num. steps | Max. relative error |
|---|---|
| 10 | $7.9 \cdot 10^{-2}$ |
| $10^2$ | $8.3 \cdot 10^{-4}$ |
| $10^3$ | $8.3 \cdot 10^{-6}$ |
| $10^4$ | $8.0 \cdot 10^{-8}$ |
| $10^5$ | $1.9 \cdot 10^{-7}$ |
| $10^6$ | $8.8 \cdot 10^{-6}$ |
| $10^7$ | $2.5 \cdot 10^{-3}$ |

Table I. Table showing max. relative error for varying number of steps along the $x$-axis.

## PROBLEM 9

For this problem, we want to find an algorithm that is specialized to solve the matrix equation $\mathbf{Av} = \mathbf{g}$ for $\mathbf{v}$, where $\mathbf{A}$ is an $N \times N$ matrix given by (6). To do this, we will exploit the fact that lower, main, and upper diagonals each are defined by a single value, $-1$, 2, and $-1$, respectively. Let $\tilde{b}_i$ denote the updated value of the $i$th element in the $\mathbf{b}$ array. Then:

$$\begin{aligned}
\tilde{b}_1 &= b_1 = 2 \\
\tilde{b}_2 &= b_2 - \frac{a_2}{\tilde{b}_1} c_1 = 2 - \frac{1}{2} = \frac{3}{2} \\
\tilde{b}_3 &= b_3 - \frac{a_3}{\tilde{b}_2} c_2 = 2 - \frac{2}{3} = \frac{4}{3} \\
\tilde{b}_4 &= b_4 - \frac{a_4}{\tilde{b}_3} c_3 = 2 - \frac{3}{4} = \frac{5}{4} \\
&\dots
\end{aligned}$$

The pattern that emerges is that $\tilde{b}_i = \frac{i+1}{i}$. This means that we can pre-compute these values up to $\tilde{b}_N$, which reduces the number of floating-point operations in the algorithm.

**Algorithm 2** Special algorithm for solving $\mathbf{Av} = \mathbf{g}$

```
for i = 2, 3, ..., N do                          ▷ First do forward substitution
    g_i ← g_i + g_{i-1} / b_{i-1}                ▷ 2 FLOPs
end for
v_N ← g_N / b_N                                  ▷ 1 FLOP
for i = N − 1, N − 2, ..., 1 do                  ▷ Then do backward substitution
    v_i ← (g_i + v_{i+1}) / b_i                  ▷ 2 FLOPs
end for
```

The number of FLOPs in Algorithm 2 is $2(N-1)+1+2(N-1) = 4N-3$, which is approximately half the number of FLOPs in Algorithm 1.

## PROBLEM 10

Having implemented the special algorithm as found in (2) in addition to the general algorithm (1), we can then use both the general and special algorithms and compare the time it takes for each program to run. We ran tests of the algorithms for $n_{\text{steps}} = 10, 10^2, ..., 10^6$ and compared them in table (II).

Here we see that while the special algorithm undoubtedly runs a little faster, the runtimes for both algorithms is of the same order of magnitude in all the cases we test for. This essentially means that for very large step sizes (small $n_{\text{steps}}$), the runtime difference between the general and special algorithm is very small, and there is not much use for a specialized algorithm in these cases. However, as the amount of steps increases, the difference in performance for the general and special algorithm also increases, thus making it more efficient and therefore beneficial to specialize the algorithm in order to achieve a shorter runtime.

| Num. steps | Time general (s) | Time special (s) |
|---|---|---|
| 10 | $2.1 \cdot 10^{-7}$ | $1.7 \cdot 10^{-7}$ |
| $10^2$ | $2.4 \cdot 10^{-6}$ | $1.9 \cdot 10^{-6}$ |
| $10^3$ | $2.4 \cdot 10^{-5}$ | $1.9 \cdot 10^{-5}$ |
| $10^4$ | $2.4 \cdot 10^{-4}$ | $2.0 \cdot 10^{-4}$ |
| $10^5$ | $2.4 \cdot 10^{-3}$ | $2.0 \cdot 10^{-3}$ |
| $10^6$ | $2.4 \cdot 10^{-2}$ | $2.0 \cdot 10^{-2}$ |

Table II. A table showing the average time it took to run the general algorithm and the special (optimized) algorithm, computed as the average time over 10 iterations.