

Solutions to Project 1

Lars Bogner

(Dated: September 4, 2025)

Code hosted at <https://github.com/larsbog/FYS4150>

PROBLEM 1

To prove that the given function is a solution to the one-dimensional Poisson equation, we are calculating the second derivative of $u(x)$ as

$$\frac{\partial u}{\partial x} = (1 - e^{-10}) + 10e^{-10x} \quad (1)$$

and therefore

$$\frac{\partial^2 u}{\partial x^2} = -100e^{-10x} \quad (2)$$

Therefore the main equation $-\frac{\partial^2 u}{\partial x^2} = f(x)$ is satisfied with $f(x) = 100e^{-10x}$. Now we have to check the remaining boundary conditions. There is no singularity within the x range: $x \in [0, 1]$, therefore the Poisson equation is well-defined in this interval. Lastly we check the boundary conditions:

$$\begin{aligned} u(0) &= 1 - 1 = 0 \\ u(1) &= 1 - 1 + e^{-10} - e^{-10} = 0 \end{aligned} \quad (3)$$

PROBLEM 2

The code for this problem can be found in the files `src/project1/poisson.cpp` and `src/project1/python/-poisson_plotter.py`. The resulting plot is displayed in [Figure 1](#).

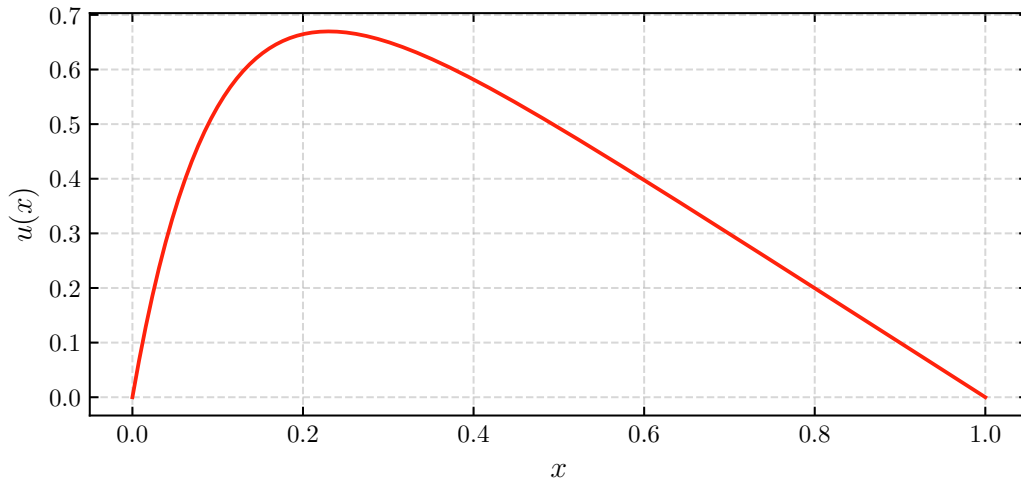


FIG. 1. Numerical values of the solution to the Poisson equation

PROBLEM 3

To discretize the Poisson equation, we start with discretizing derivatives. For this to work we will Taylor expand the function $u(x)$ around a point x_i :

$$u(x_j) = u(x_i) + (x_j - x_i)u'(x_i) + \frac{(x_j - x_i)^2}{2}u''(x_i) + \dots \quad (4)$$

If our $\{x_j\}$ are equally spaced, i.e. $x_{j+1} = x_j + \Delta$, $\forall j \in \mathbb{Z}$, we can write the Taylor expansion in terms of the spacing Δ :

$$u(x_j) = u(x_i) + (j - i)\Delta u'(x_i) + \frac{(j - i)^2}{2}\Delta^2 u''(x_i) + \dots \quad (5)$$

To find a discrete expression for the second derivative, we use the discrete version of derivatives:

$$\frac{\partial g(x)}{\partial x} = \lim_{\Delta \rightarrow 0} \frac{g(x + \Delta/2) - g(x - \Delta/2)}{\Delta} \quad (6)$$

and thus follows

$$\frac{\partial}{\partial x} \frac{\partial g(x)}{\partial x} = \lim_{\Delta \rightarrow 0} \frac{1}{\Delta^2} (g(x + \Delta) - g(x) - g(x) + g(x - \Delta)). \quad (7)$$

If we take this expression and apply it to $u(x_j)$, we get

$$\frac{\partial^2 u(x_j)}{\partial x_j^2} = \lim_{\Delta \rightarrow 0} \frac{u(x_{j+1}) - 2u(x_j) + u(x_{j-1}))}{\Delta^2} \quad (8)$$

The rest of the problem can be discretized by mapping the continuous domain of $x \in [0, 1]$ to a discrete spectrum of points, i.e. $f(x) \rightarrow f_j = f(x_j)$ and $u(x) \rightarrow u_j = u(x_j)$. If we discard the limit of $\Delta \rightarrow 0$ we move to an approximation of u_j which we call v_j . The problem can then be written as

$$-\frac{v_{j+1} - 2v_j + v_{j-1}}{\Delta^2} = f_j \quad \forall j \in \{1, \dots, n-1\} \quad (9)$$

PROBLEM 4

When slightly rearranging the discretized Poisson equation from Problem 3, we get

$$v_{j+1} - 2v_j + v_{j-1} = -\Delta^2 f_j \quad \forall j \in \{1, \dots, n-1\} \quad (10)$$

which can be directly interpreted as a matrix equation of the form $A\vec{v} = \vec{g}$. We need to pay special attention to the border regions of the matrix equation, as we can't compute v_0 and v_n . If we absorb those into $\vec{g}$, we get a correct matrix equation system. Therefore we define $g_j = -\Delta^2 f_j$, $\forall j \in \{2, \dots, n-2\}$ and $g_1 = -\Delta^2 f_1 - v_0$ and $g_{n-1} = -\Delta^2 f_{n-1} - v_n$. The matrix A is a tridiagonal matrix with the following structure:

- The main diagonal consists of -2 .
- The first superdiagonal consists of 1 .
- The first subdiagonal consists of 1 .

PROBLEM 5

Subproblem (a)

As our discrete solution always depends on one value before and one value after the current value, our matrix has to be smaller in either direction by one row and column. If we put this into an equation we can say:

$$n = m - 2 \quad (11)$$

Algorithm 1 General Algorithm (Thomas Algorithm)[1] for solving $A\vec{v} = \vec{g}$ where A is a tridiagonal matrix

```

function GENERAL_ALGORITHM( $\vec{a}, \vec{b}, \vec{c}, \vec{g}$ )
   $\vec{c}' \leftarrow \vec{0}$                                      ▷ Initialize temporary vectors
   $\vec{g}' \leftarrow \vec{0}$ 
   $c'_1 \leftarrow c_1/b_1$ 
  for  $i = 2, \dots, n-1$  do                             ▷ Recursively compute  $c'$ 
     $c'_i \leftarrow c_i/(b_i - a_i c'_{i-1})$ 
   $g'_1 \leftarrow g_1/b_1$ 
  for  $i = 2, \dots, n$  do                             ▷ Recursively compute  $g'$ 
     $g'_i \leftarrow (g_i - a_i g'_{i-1})/(b_i - a_i c'_{i-1})$ 
   $\vec{v} \leftarrow \vec{0}$                                      ▷ Initialize solution vector
   $v_n \leftarrow g'_n$ 
  for  $i = n-1, n-2, \dots, 1$  do                       ▷ Back substitution
     $v_i \leftarrow g'_i - c'_i v_{i+1}$ 
  return  $\vec{v}$ 

```

Subproblem (b)

We can find the inner part of $\vec{v}^*$, i.e. $v_i = v_{i+1}^* \forall i \in \{1, \dots, n\}$.

PROBLEM 6

Subproblem (a)

To solve a general tridiagonal matrix equation we can use the algorithm described in [1]. The algorithm is structured as described in Algorithm 1.

Subproblem (b)

The floating point operations (FLOPs) can be counted as follows:

- Calculate first element of c' : 1 FLOP
- Calculate rest of c' : 3 FLOPs per element ($3(n-2)$ FLOPs)
- Calculate first element of g' : 1 FLOP
- Calculate rest of g' : 5 FLOPs per element ($5(n-1)$ FLOPs)
- Back substitution: 2 FLOPs per element ($2(n-1)$ FLOPs)

In total this gives us

$$N = 1 + 3(n-2) + 1 + 5(n-1) + 2(n-1) = 10 \cdot n - 11 \sim \mathcal{O}(n) \quad (12)$$

floating point operations (FLOPs).

PROBLEM 7

Subproblem (a)

The implementation of the specialized algorithm can be found in the file `src/project1/src/solvers.cpp` with usage in `src/project1/poisson_solver.cpp`.

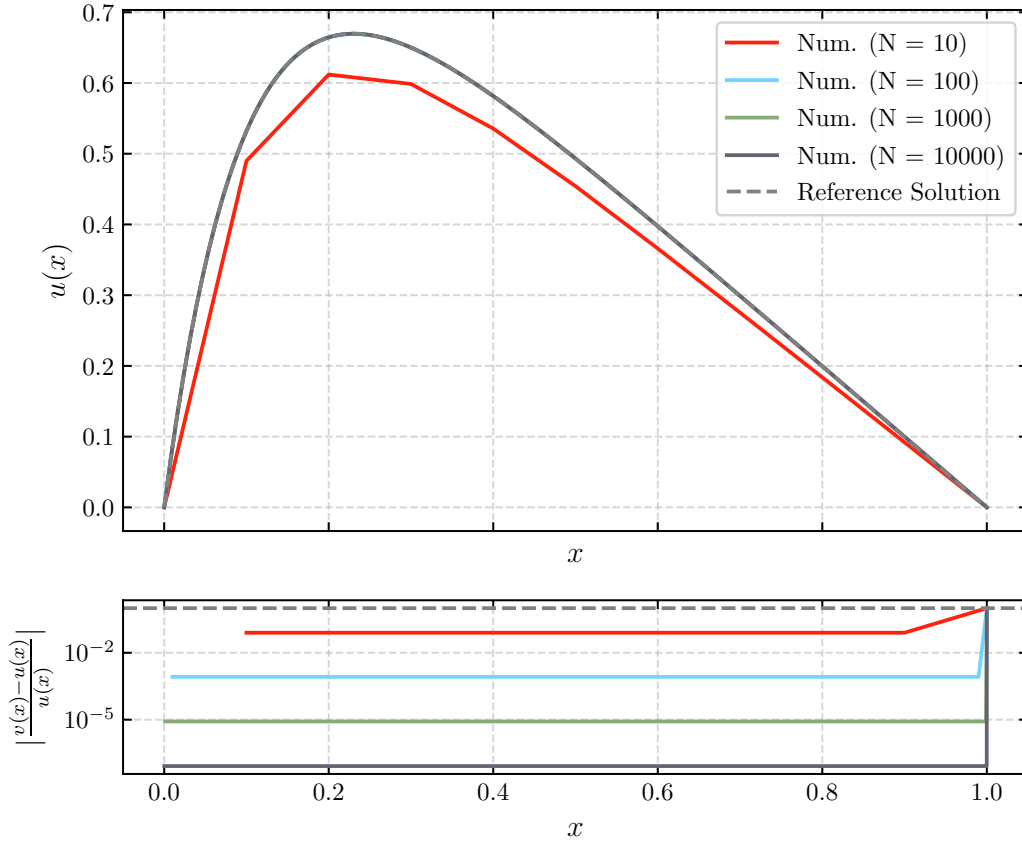


FIG. 2. Numerical values of the solution to the Poisson equation for different values of N

Subproblem (b)

Comparing the solutions for different values of N we can clearly see the increase in accuracy as N is increased. The numerical solutions converge towards the analytical solution. The results are summarized in Figure 2. To assess the quality of the results, the solutions are plotted in comparison to the numerical solution in the upper plot, while the lower plot shows the relative error. The relative error is defined as

$$\frac{\delta v_i}{u_i} = \frac{v_i - u_i}{u_i} \quad (13)$$

PROBLEM 8

Subproblem (a)

Plotting the absolute error for different values of N reveals the said convergence behavior. As N increases, the absolute error decreases, indicating that the numerical solution is approaching the analytical solution. The results are summarized in Figure 3.

Subproblem (b)

Plotting the relative error for different values of N reveals a constant value for the relative error across the entire domain of x . The relative error decreases by two orders of magnitude for every order of magnitude increase in N .

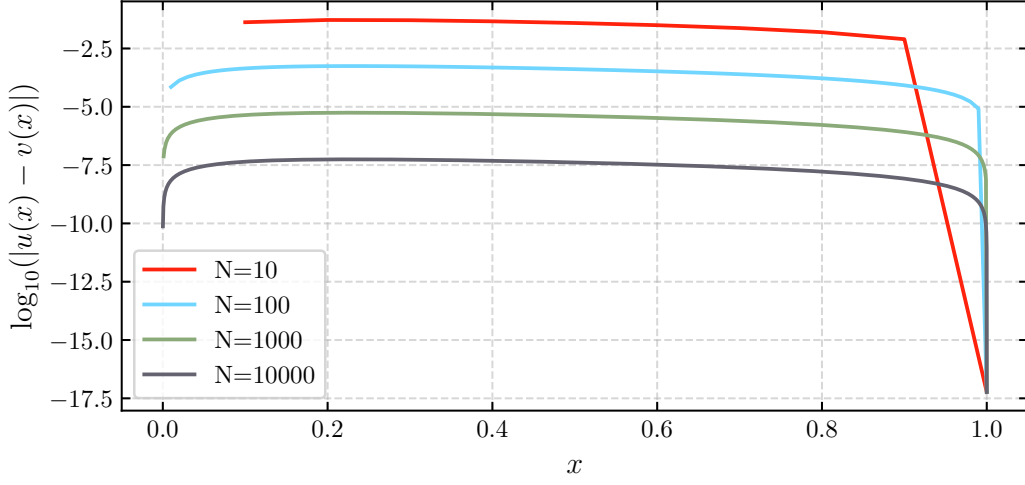


FIG. 3. Absolute error of the numerical solution for different values of N

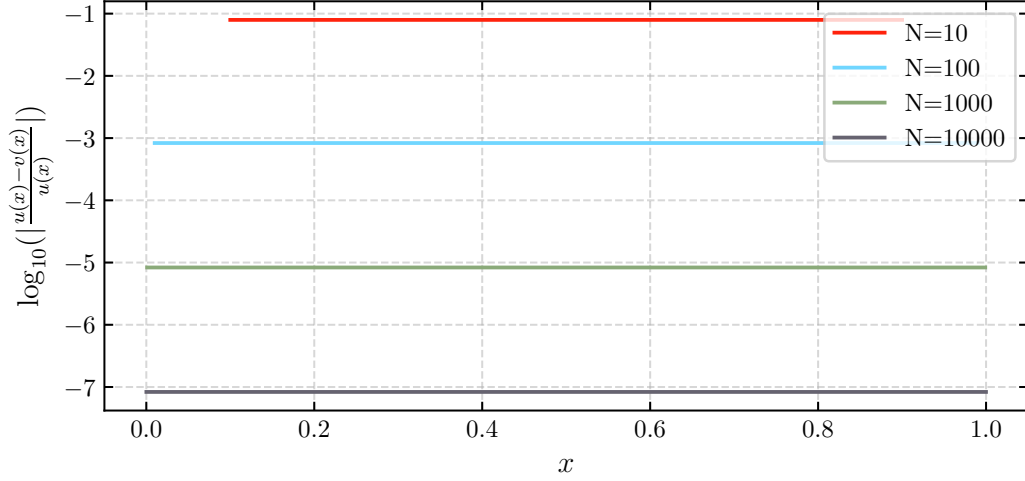


FIG. 4. Relative error of the numerical solution for different values of N

The results are summarized in [Figure 4](#). To avoid issues with divisions by zero, the points where $u(x) \leq 10^{-10}$ were omitted from the plot.

Subproblem (c)

The maximum relative errors are calculated directly using C++ as the output size scales linearly with N and gets overwhelmingly large for bigger N . The results are output using the `std::cout` command and retrieved using GNU/Linux command line utilities. The code for calculating the maximum relative error $\max_i \frac{\delta v_i}{u_i}$ (see (13)) is included in the C++ implementation. The results are summarized in [Table I](#). We see a steady decrease of the relative error up to $N = 100\,000$, after which the relative error increases again. This will most likely be due to the limited numerical precision of floating point numbers in C++ and associated numerical errors.

Algorithm 2 Special Algorithm for computing $\vec{v}$ from $\vec{g}$ using constant tridiagonal elements

function OPTIMAL_ALGORITHM($\vec{g}$)

 $\vec{v} \leftarrow \vec{0}$
 $\triangleright$ Initialize solution vector

 $\tilde{g} \leftarrow \vec{0}$
 $\triangleright$ Initialize modified input vector

 $\tilde{g}_0 \leftarrow g_0/2$
for $i = 1, \dots, n-1$ **do**
 $\triangleright$ Forward substitution

 $\tilde{g}_i \leftarrow (g_i + \tilde{g}_{i-1}) \cdot \frac{i+1}{i+2}$
 $v_{n-1} \leftarrow \tilde{g}_{n-1}$
for $i = n-2, n-3, \dots, 0$ **do**
 $\triangleright$ Backward substitution

 $v_i \leftarrow \tilde{g}_i + \frac{i+1}{i+2} \cdot v_{i+1}$
return $\vec{v}$

N	Max Relative Errors
10	7.933×10^{-2}
100	8.329×10^{-4}
1000	8.333×10^{-6}
10 000	8.333×10^{-8}
100 000	1.436×10^{-9}
1 000 000	8.404×10^{-7}
10 000 000	2.984×10^{-6}

 TABLE I. Summary of maximum relative errors for different values of N
PROBLEM 9
Subproblem (a)

To specialize the general algorithm to our specific problem, we take a look at how our general algorithm modifies the input vectors during forward and backward substitution. As a first step, we can calculate the values c'_i from Algorithm 1 explicitly as $c'_i = \frac{-i}{i+1}$. This explicit formulation also allows for a more straightforward definition of $g'_i \equiv \tilde{g}_i$. Combining these two results we can rewrite the algorithm as described in Algorithm 2.

Subproblem (b)

The floating point operations (FLOPs) can be counted as follows:

- Calculate first element of $\tilde{g}$: 1 FLOP
- Calculate remaining elements of $\tilde{g}$: $3(n-1)$ FLOPs
- Calculate last element of v : 0 FLOPs
- Calculate remaining elements of v : $3(n-1)$ FLOPs

If we dismiss integer additions and subtractions, the total number of FLOPs can be simplified to:

$$N = 1 + 3(n-1) + 0 + 3(n-1) = 6 \cdot n - 5 \sim \mathcal{O}(n) \quad (14)$$

Subproblem (c)

The implementation of the specialized algorithm can be found in the file `src/project1/src/solvers.cpp` with usage in `src/project1/poisson_timing.cpp`. Secondly a variant with further optimizations on the memory access patterns is implemented in the same location. From now on we will reference this version as the *Optimized Algorithm*.

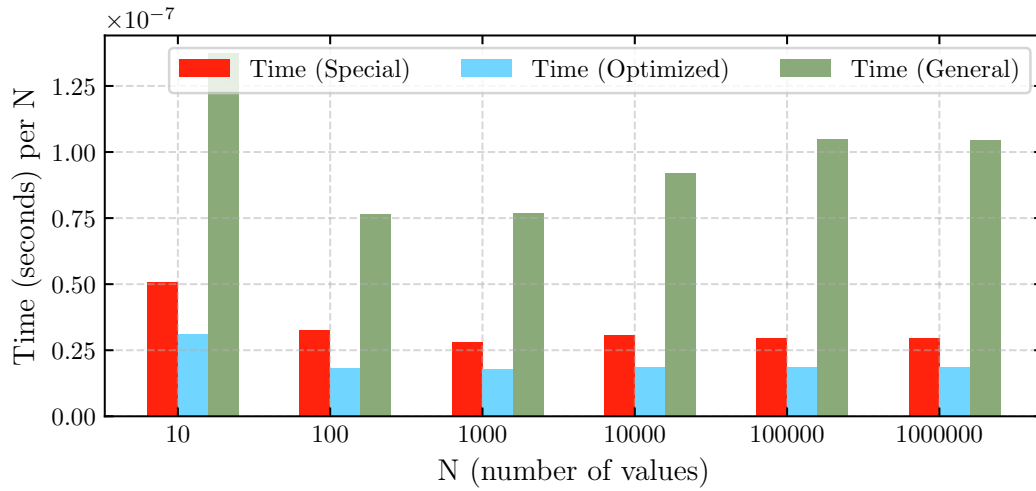


FIG. 5. Normalized execution times for different algorithms and problem sizes

PROBLEM 10

To evaluate the performance of the different algorithms, the algorithms were executed 100 times for varying problem sizes N . The execution times were recorded and analyzed to compare the efficiency of the algorithms. To compare the effect of different N on the execution time, the execution times were normalized by dividing them by N . The results are summarized in [Figure 5](#). We can observe, that the specialized algorithm outperforms the general algorithm consistently across all problem sizes by more than a factor of 2. The optimized algorithm shows a further improvement over the specialized algorithm with very consistent execution times across all problem sizes.

[1] “Tridiagonal matrix algorithm,” Page Version ID: 1306920126.