Project 1

Christophe Kristian Blomsen
c.k.blomsen@astro.uio.no

Eloi Richard Martaillé
e.m.richard@astro.uio.no

Vebjørn Øvereng
vebjoro@uio.no

Vetle Henrik Hvoslef
vetlehh@uio.no

September 13, 2022

Contents

Corresponding github repository: https://github.com/christopheblomsen/project1

Problem 1

The 1D Poisson equation is given by:

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

where $f(x)$ is a known source term. For $f(x) = 100e^{-10x}$, with the boundary conditions $u(0) = 0$ and $u(1) = 0$, we propose the solution $u(x)$:

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

We check that this is a solution to the Poisson equation in this case by calculating the double derivative of $u(x)$:

$$\begin{gathered} \frac{\mathrm{d}u}{\mathrm{d}x} = 1 - e^{-10} + 10e^{-10x} \\ \Downarrow \\ -\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = -100e^{-10x} = f(x) \end{gathered}$$

We have found that the function u defined is indeed a solution to our ODE and the last verification we need is if $u(x)$ satisfy our boundary conditions.

We can see for $x = 0$, we have $u(0) = 1 - 0 - 1 = 0$ and for $x = 1$ we have $u(1) = 1 - 1 + e^{-10} - e^{-10} = 0$. Our function $u(x)$ respect both boundaries conditions and the second derivative is equal to $-f(x)$, hence we have shown that $u(x)$ is a solution to our ODE.

Problem 2

The function can be found in `src/utils.cpp`. Where we are going to be using the `armadillo` header. In this we first use the `arma::vec` type to fill a x array from $0, 1$, with 100 steps. Then making the function $f(x)$ is trivial with `armadillo`. Furthermore we have a function that prints this out to a `prob2.txt` file. Then using `Python` to plot it we get the following figure.

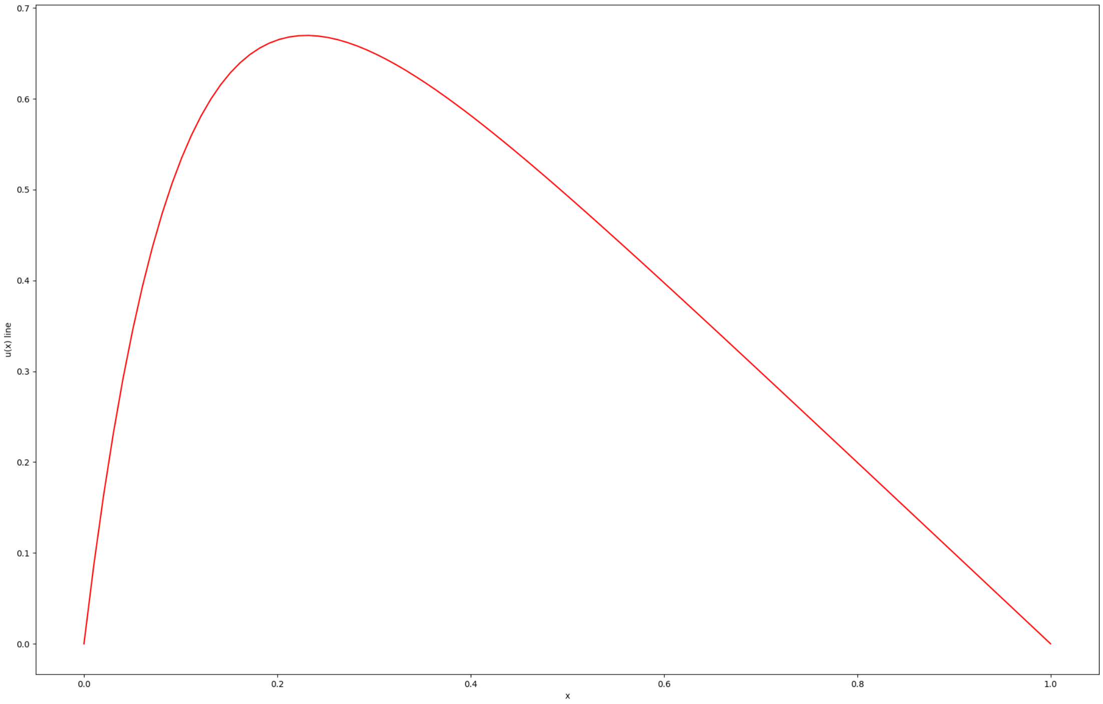

Figure 1: Graph for the analytical solution

Problem 3

We now want to solve the Poisson equation numerically, in order to achieve this we will start by dicretizing the second derivative of $u(x)$ using the a Taylor expansion, this gives:

$$\frac{\mathrm{d}^2u(x)}{\mathrm{d}x^2} = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} + \mathcal{O}(h^2) \tag{3}$$

with h being the step size, which we consider constant. Now that we have our expansion we can approximate $v_i(x) \approx u_i(x)$ giving us the approximation of $\mathcal{O}(h^2)$ and the discretized Poisson equation:

$$-\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} \approx f_i$$

Problem 4

As seen in Problem 3 we have come with a discretized version of our ODE with a step h constant creating $n_{\text{step}} + 1$ discrete values of v and f. Here we have this relation between number of steps and number of unknowns that our matrix equation has, $n = n_{\text{step}} - 1$ so $n + 1 = n_{\text{step}}$

We can remark that the Taylor expansion we used to define a discrete version of the second derivative for a step i requires the previous and future step value of our function v. Causing a problem for the boundaries values v_0 and $v_{n_{\text{step}}}$ since for the first case v_0 we do not have a previous step v_{-1} and for v_{n+1}, v_{n+2} is not defined as well. However we may not be able to numerically compute the boundaries values of v but we are able to calculate all the intermediates values.

In consequence, we will have the relations for all v_i $i \in \{1, ..., n\}$:

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

From this we can see that the left side of every equation is known because $f(x)$ is given, h is a controlled parameter and v_0 and v_{n+1} are our boundaries conditions hence we know their value. From this we define $g_i = h^2 f_i$ for $i \in \{2, ..., n\}$, $g_1 = h^2 f_1 + v_0$, $g_n = h^2 f_n + v_{n+1}$ and the column vector

$$\vec{g} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}$$

For the right side, we define $\vec{v}$ a $1 \times n$ vector containing all the approximated solution except for our two bondaries points which are known and by defining $\mathbf{A}$ a $n \times n$ matrix

$$\mathbf{A}\vec{v} = \begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & 2 & -1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} \tag{4}$$

With these definitions, we have rewritten our system of discretized equations to a matrix equation: $\mathbf{A}\vec{v} = \vec{g}$.

Problem 5

Problem 5.1 a)

if we define $\vec{v}^*$ the complete solution of length m and $\mathbf{A}$ the matrix of size $n \times n$ define in a similar way as problem 4. In order to transform our system of equations we had to ignore the boundaries conditions as they were ill-defined due to the discretization of the second derivative. With that in mind, the complete solution must contain the boundaries conditions so if we have m discrete values and a square matrix of size n without two values, we have the following relation: $n = m - 2$.

Problem 5.2 b)

In a) and problem 4 we established a matrix equation to solve $\vec{v}$ containing all points except for the boundary points so solving this matrix equation gives us all values of v_i with $i \in \{1, \dots, n\}$.

Problem 6

Problem 6.1 a)

We have the matrix equation $\boldsymbol{A}\vec{v} = \vec{g}$ with $\boldsymbol{A}$ being a general tridiagonal matrix. In order to solve this equation we will proceed in two steps. The first one, known as forward substitution consist in doing matrix operation to obtain an upper triangular matrix and the last step, backwards substitution, which will give us our approximation $\vec{v}$. The algorithm works by transforming $\boldsymbol{A}$ to the identity matrix, modifying $\vec{g}$ along the way.In the end we see that the solution to $\vec{v}$ will equal the modified $\vec{g}$. Now let's see in detail how we can perform the algorithm.

We have:

$$\boldsymbol{A} = \begin{bmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_3 & b_3 & c_3 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & 0 & a_n & b_n \end{bmatrix}$$

We perform the operation: $R_2 \rightarrow R_2 - \frac{a_2}{b_1} R_1$ which implies that the second row becomes $R_2 = [0, b_2 - \frac{a_2}{b_1} c_1, 0, \dots, 0]$ and the second element of $\vec{g}$ becomes $\tilde{g_2} := g_2 - \frac{a_2}{b_1} c_1$. We define $\tilde{b_2} := b_2 - \frac{a_2}{b_1} c_1$, $\tilde{b_1} := b_1$ and $\tilde{g_1} := g_1$.

We can see that performing the general operation: $R_i = R_i - \frac{a_i}{\tilde{b}_{i-1}} R_{i-1}$ for $i \in \{2, ..., n\}$ and defining $\tilde{b_i} := b_i - \frac{a_i}{\tilde{b}_{i-1}} c_{i-1}$, $\tilde{g_i} := g_i - \frac{a_i}{\tilde{b}_{i-1}} \tilde{g}_{i-1}$ will lead us to:

$$\begin{bmatrix} \tilde{b_1} & c_1 & 0 & 0 & \dots & 0 \\ 0 & \tilde{b_2} & c_2 & 0 & \dots & 0 \\ 0 & 0 & \tilde{b_3} & c_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & \dots & 0 & \tilde{b}_{n-1} & c_{n-1} \\ 0 & \dots & \dots & 0 & 0 & \tilde{b_n} \end{bmatrix} \vec{v} = \begin{bmatrix} \tilde{g_1} \\ \tilde{g_2} \\ \tilde{g_3} \\ \vdots \\ \tilde{g}_{n-1} \\ \tilde{g_n} \end{bmatrix}$$

The first step is now complete and we start the last step. First we observe that all coefficient from $\vec{a}$, $\vec{b}$ and $\vec{c}$ is known, therefore we can see that the last row yields the following relation: $\tilde{b}_n v_n = \tilde{g}_n \Rightarrow v_n = \frac{\tilde{g}_n}{\tilde{b}_n}$

By making the row operation $R_n \rightarrow \frac{R_n}{\tilde{b}_n}$ we 'clear' the nth row and we have found the first value of our numerical solution v_n. Now that we know v_n we can go back one rank to find the value of v_{n-1} and we repeat the process in order to find v_{n-2} until we find the last value v_1. So we can generalize the row operation needed:

$$R_{n-1} \rightarrow \frac{R_{n-1} - c_{n-1}R_n}{\tilde{b}_{n-1}}$$

which allows us to write an expression for v_{n-1}

$$v_{n-1} = \frac{\tilde{g}_{n-1} - c_{n-1}v_n}{\tilde{b}_{n-1}}$$

By repeating the operation enough time to find all values of v_i giving us the relation for v_i:

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

Problem 6.2 b)

To count the FLOPs for this algorithm we start by looking at how many FLOPs is needed for the forwards substitution. To start of we determine $\tilde{b}_1$ as:

$$\tilde{b}_1 = b_1 \tag{5}$$

Then we proceed to do the following calculations for every $n \in \{2, \dots n\}$:

$$d = \frac{a_i}{\tilde{b}_{i-1}} \tag{6}$$

$$\tilde{b}_i = b_i - d \cdot c_{i-1} \tag{7}$$

$$\tilde{g}_i = g_i - d \cdot \tilde{g}_{i-1} \tag{8}$$

the first line uses 1 FLOPs, and each of the lines below uses 2 FLOPs. They are repeated $(n-1)$ times. Hence we have $2(n-1)+2(n-1)+(n-1) = 5(n-1)$ FLOPs for the Forward substitution.

For the backwards substitution we start off by calculating the last element of vector v.

$$v_n = \frac{\tilde{g}_n}{\tilde{b}_n} \tag{9}$$

next we calculate the remaining elements of v by:

$$v_i = \frac{\tilde{g}_i - c_i v_{i+1}}{\tilde{b}_i} \quad i \in \{n-1, \dots, 1\} \tag{10}$$

The first step in the backwards substitution contributes a single FLOP, the remaining iterations yield $3(n-1)$ FLOPs to the total FLOP-count.

Thus the total number of FLOPs for the Thomas-algorithm is $8 \times (n-1) + 1 = 8n - 7$

Problem 7

Problem 7.1 a)

main.cpp calls the function `gen_thoman_alg` from `src/utils.cpp` to solve the matrix equation $\mathbf{A}\vec{v} = \vec{g}$, where $\mathbf{A}$ is a general tridiagonal matrix. The solution is written to a file with name `general_thomas_<N>.txt`, where N is the number of points.

We define the vectors `a`, `b`, `c` , `g` and `v` with `n` where the first three of these define the signature $(-1, 2, -1)$ of the three diagonals in our matrix. the `g`-vector is initialized by `g = h*h*f(x)`.

We explicitly force the boundary values at $v_{n+1} = 0$ and $v_0 = 0$ and iterate over the inner points. The vectors $\tilde{b}$ and $\tilde{g}$ are defined as $b\_$ and $g\_$ respectively. We then implement the algorithm as defined in task 6. (In our program the variable n_{steps} is the total number of points.)

Problem 7.2 b)

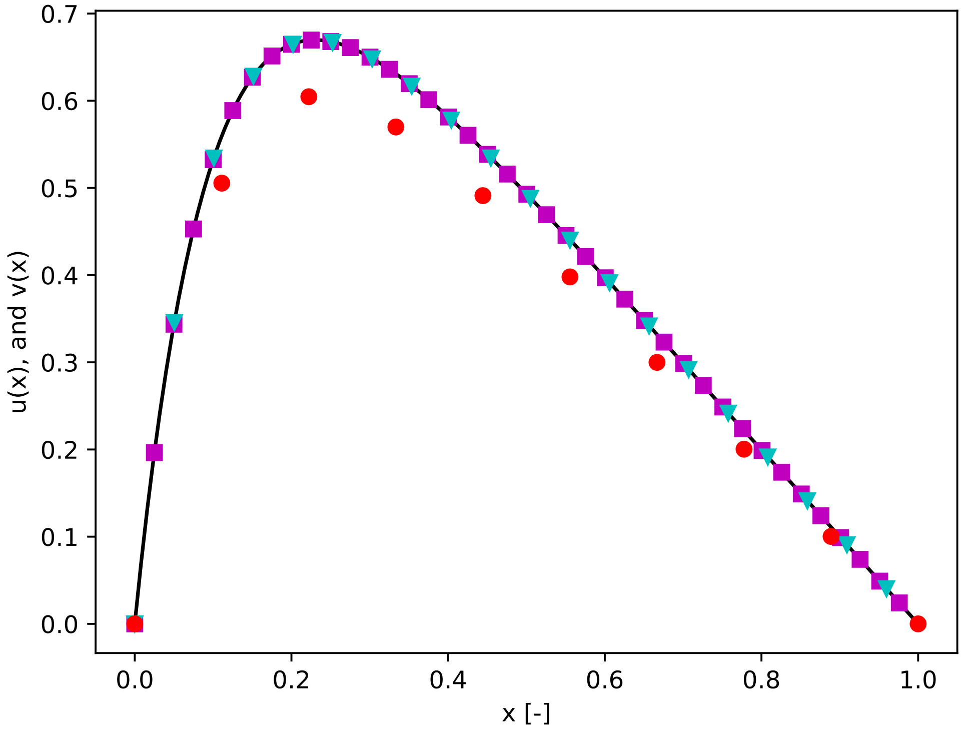

Figure 2: Plot of numerical approximation and excact solution (black line) for different number of steps, red circle is $n_{\text{steps}} = 10$, cyan triangle is $n_{\text{steps}} = 100$, and magenta square is $n_{\text{steps}} = 1000$. We see a signinficant improvement in the accuracy when we increase from 10 to 100 steps. By eye we don't see a significant difference between 100 and 1000 steps.

In **??** we can see the exact solution $u(x)$ and the numerical approximation v_i. By eye we can see a significant improvement when we increase the number of steps from 10 to 100, it is however difficult to see a difference in accuracy between 100 and 1000 steps.

Problem 8

Problem 8.1 a)

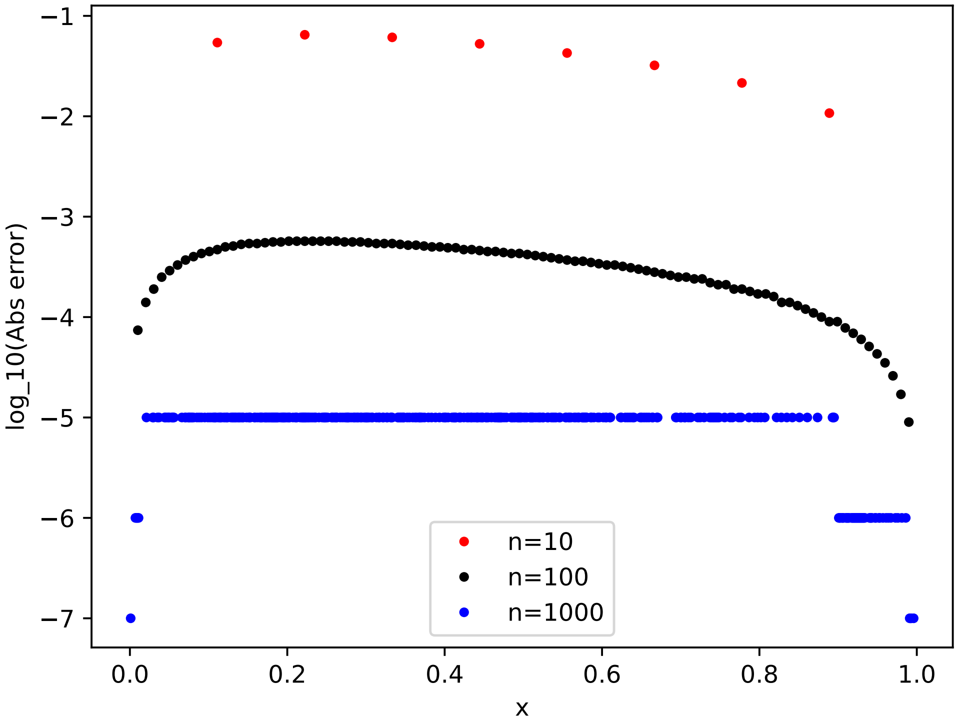

Figure 3: Plot of $\log_{10} \Delta_i$

We make plots **??** of the absolute error, $\Delta_i = |u_i - v_i|$ in every inner point using `plot8.py`. We see that the errors are smaller closer to the boundaries for all.

Problem 8.2 b)

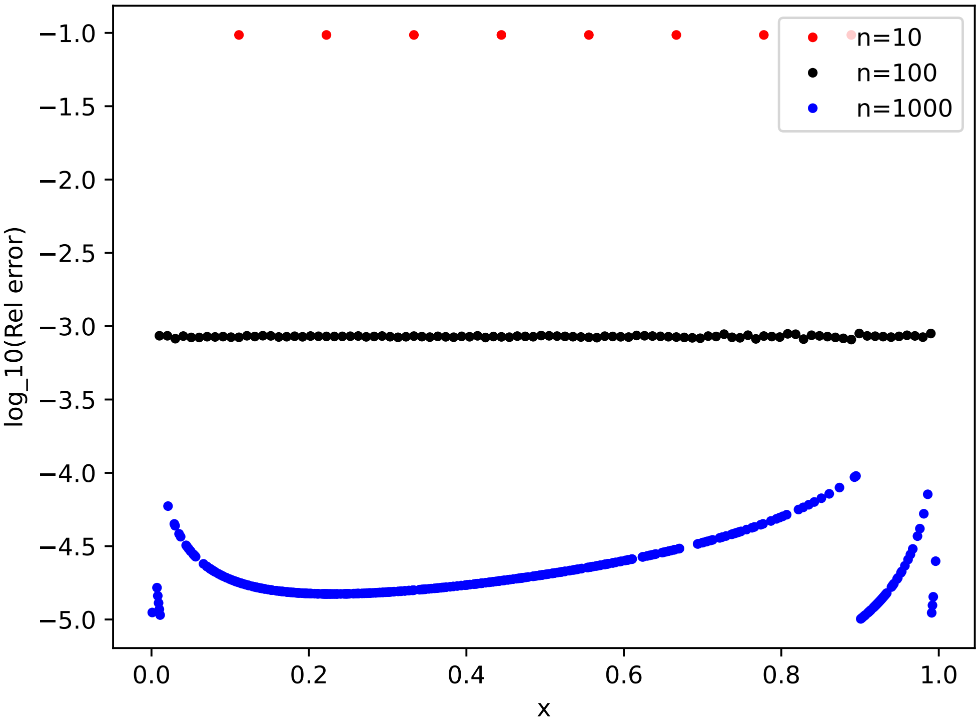

Figure 4: Plot of $\log_{10} \varepsilon_i$

Plot **??** shows the relative error for various number of points. The errors are reduced by two orders of magnitude when we increase the number of points from 10 to 100. In this range our error is of $\mathcal{O}(h^2)$, we see that the truncation error dominates. Near the boundaries we see that the relative error increases for the smallest stepsize. This may be due to possible catastrophic cancellations in **??**, **??**, or **??**. n

Problem 8.3 c)

Maximum relative error for various number of steps.	
n_{steps}	Max relative error %
10	9.7
100	0.089
10^3	0.10
10^4	0.009
10^5	0.004
10^6	0.010
10^7	0.010

The table above shows the maximum relative error for different step size. For the smallest number of points, i.e 10, we have the worst error close to 10% and as we increase the number of points we reduces the error as expected but we remark that around 10^5 points, increasing the number of points actually augment the maximum relative error. We can explain this on the first thought, a contradicting result but as we have seen in class we have two types of errors, the mathematical error due to our approximation and the round-off error due to the computer limited memory. We have seen that reducing the 'distance' between our points would give us a better answer but it will start to increase the round-off error as our step size will get smaller and smaller we have higher chance of catastrophic cancellation. Hence we have to reduce the gap between our points but we have to take in account that doing so will increase the error associated with the computer's memory showing us that an optimal step size is not the smallest possible but an intermediate one minimizing both types of error.

Problem 9

Problem 9.1 a)

In this case we have: $\vec{a} = \{-1, -1, \ldots, -1\}$, $\vec{b} = \{2, 2, \ldots, 2\}$ and $\vec{c} = \{-1, -1, \ldots, -1\}$.

So by using equation (8) and inserting the constant values for a_i we get

$$g_i = g_i + \frac{\tilde{g}_{i-1}}{\tilde{b}_{i-1}}$$

And using the same trick but with the equation for v_i and inserting for c_i we get

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

For $\tilde{b}_i$ we use that since b_i is constant we calculate the first few terms

$$\begin{aligned} \tilde{b}_1 &= 2 \\ \tilde{b}_2 &= \frac{3}{2}\tilde{b}_3 && = \frac{4}{3} \end{aligned}$$

we can see that it makes this pattern for ‘$Tildeb_i = \frac{i+1}{i}$.

Problem 9.2 b)

All of these three operations takes two FLOPs each time and is done $(n-1)$ times. However we still need to calculate $v_n = \frac{\tilde{g_n}}{\tilde{b_n}}$, so the total FLOPs of the special algorithm is $3(n-1)+1 = 3n-2$

Problem 9.3 c)

In the same way as problem 7.1 a), we use main.cpp. Howevere here it calls the function `gen_thoman_alg` from `src/utils.cpp` and makes this file `special_thomas_<N>.txt`, where N is the number of points.

This code has the same structure as task 7 but with different algorithms for $\tilde{b}_i$, $\tilde{g}_i$ and v_i, also no b_i, a_i and c_i were defined.

Problem 10

In this question, we try to compare the execution time for both our algorithms with different step size. In order to compute the execution time we used the chrono function from the C++ library given in the project description. We run the program up to 10^6 steps 5 times every time. It allowed us to compute the mean execution time and we use the standard deviation formula to calculate the error. The results are presented in the table underneath.

Execution time between Thomas algorithm and The special algorithm		
Step size N	Thomas algorithm [s]	Special algorithm [s]
10	6.3960e-06±1.1729e-06	4.7944e-06±4.1929e-6
100	3.14726e-05±1.4968e-05	1.34792e-05±3.6568e-05
10^3	0.0002013±9.371e-05	0.00010249±2.7006e-05
10^4	0.00163008±0.00037397	0.0013955±0.0003512
10^5	0.0135912±0.0021826	0.01068024±0.00043691
10^6	0.127284±0.00466895	0.109234±0.00438743

We can observe that the special algorithm is always a bit faster than the general one which is what we would expect with the FLOPs calculated in the previous exercises. However we can note that the time difference is minimal and we can explain it by the fact that both algorithm are of order $O(N)$ so both execution time scale linearly with N and the optimizations done on both code reduces the execution time but the differences between the two algorithm should not be due to the step size as it increases the execution time linearly for both code.