FYS3150 - Project 1

Andrew Quan, Oskar Idland, Hishem Kløvnes, Håvard Skåli
(Dated: September 9, 2023)

GitHub Repository

PROBLEM 1

We have the one-dimensional Poisson equation

$$-\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = 100e^{-10x}, \qquad x \in [0,1] \tag{1}$$

where the right hand side of the equation is the source term $f(x)$. We have the boundary conditions $u(0) = 0$ and $u(1) = 0$. Integrating both sides of eq. (1) with respect to x twice we get

$$\begin{aligned}
\frac{\mathrm{d}^2u}{\mathrm{d}x^2} &= -100e^{-10x} \\
\iint \left(\frac{\mathrm{d}^2u}{\mathrm{d}x^2}\right)\mathrm{d}x^2 &= -100 \iint e^{-10x}\mathrm{d}x^2 \\
\int \left(\frac{\mathrm{d}u}{\mathrm{d}x}\right)\mathrm{d}x &= -100 \int \left(-\frac{1}{10}e^{-10x} + C\right)\mathrm{d}x \\
u(x) &= -100\left(\frac{1}{100}e^{-10x} + Cx + D\right) \\
u(x) &= Cx + D - e^{-10x}
\end{aligned} \tag{2}$$

Where C and D are some arbitrary constants. Invoking the boundary conditions we get

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

Thus, plugging these constants into the general solution and rearranging the terms, the final solution becomes

$$u(x) = 1 - \left(1 - e^{-10}\right)x - e^{-10x} \tag{3}$$

just like we wanted to show.

PROBLEM 2

We created a function `u_func` defined in `u_func.cpp` that uses eq. (3) to return the exact solution for a given vector $\vec{x}$. In `main.cpp` we defined a vector of $n = 100$ x-values between 0 and 1 and called on `u_func` with this vector to get a vector $\vec{u}$ containing the exact solutions for each of the x-values. We then wrote these values with ten decimals into a binary file by passing the two vectors to our function defined in `write_to_file.cpp`, and used our script `plot_data.py` to plot them against each other, which is shown in fig. 1.

PROBLEM 3

Our aim is to derive a discretized version of the Poisson equation which lets us calculate approximates values v of the exact values u. To create this program we need to discretize the double derivative, which we can do by recalling its definition:

$$\frac{\mathrm{d}^2u}{\mathrm{d}u^2} = \lim_{\mathrm{d}x\to 0} \frac{-u(x-\mathrm{d}x) + 2u(x) - u(x+\mathrm{d}x)}{\mathrm{d}x^2} \tag{4}$$

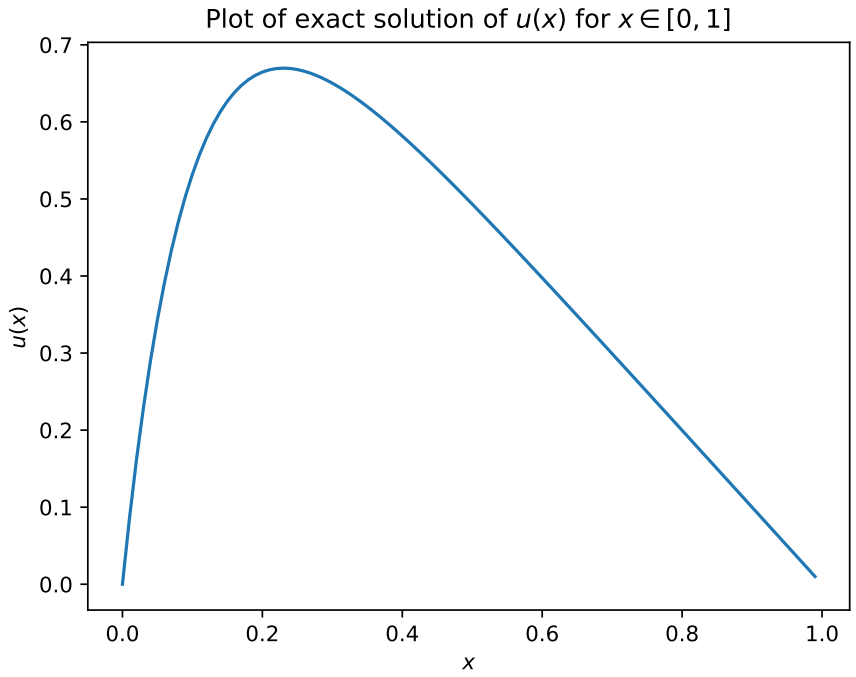

FIG. 1. Plot of the exact solution of $u(x)$ for $x \in [0, 1]$. We used $n = 100$ data points for the plot.

If we let the infinitesimal value dx rather become a finite, small value Δx, we'll still be able to calculate the double derivative with good precision, as long as we don't let this value grow too large. Since this is an approximation, a discretized version of eq. (1), we will use $v(x)$ instead of $u(x)$ in the following procedures to make it clear that this is not an exact solution. The resulting equation becomes

$$\frac{-v(x - \Delta x) + 2v(x) - v(x + \Delta x)}{\Delta x^2} = -100e^{-10x} \tag{5}$$

Let's denote the i^{th} x-value in out set of data points with x_i, so that the corresponding v-value will be denoted with v_i. This implies that at the boundaries we have $x_0 = 0$, $v_0 = 0$, and $x_{N-1} = 1$, $v_{N-1} = 0$, where N are the number of data points. Thus, the final algorithm is

$$\frac{-v_{i-1} + 2v_i - v_{i+1}}{h^2} = f_i \tag{6}$$

where $h = \Delta x$ is defined as

$$h = \frac{x_N - x_0}{N - 1} \tag{7}$$

and we've rewritten the forcing term to include the minus sign that was originally on the left hand side of the equation, meaning that

$$f_i = f(x_i) = -100e^{-10x_i} \tag{8}$$

PROBLEM 4

Let us now multiply h^2 with both sides of eq. (6), consequently rewriting the equation:

$$-v_{i-1} + 2v_i - v_{i+1} = g_i, \qquad g_i = h^2 f_i \tag{9}$$

Furthermore, we may define the vectors $\vec{v}$ and $\vec{g}$ such that

$$\vec{v} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{N-1} \end{bmatrix}, \quad \vec{g} = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{N-1} \end{bmatrix} \tag{10}$$

Using our knowledge of linear algebra, we may now define the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & \vdots \\ 0 & -1 & 2 & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & \dots & 2 \end{bmatrix} \tag{11}$$

which multiplied with $\vec{v}$ gives us

$$\mathbf{A}\vec{v} = \begin{bmatrix} 2 & -1 & \dots & 0 \\ -1 & 2 & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & 2 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{N-1} \end{bmatrix} = \begin{bmatrix} 2v_0 - v_1 \\ -v_0 + 2v_1 - v_2 \\ \vdots \\ -v_{N-2} + 2v_{N-1} \end{bmatrix} \tag{12}$$

We immediately see from eq. (9) that this gives us the vector $\vec{g}$ containing the g_i-values with the same indexes as the corresponding v_i-values from $\vec{v}$. Thus, we can indeed rewrite the original discretized equation as the matrix equation

$$\mathbf{A}\vec{v} = \vec{g} \tag{13}$$

where $\mathbf{A}$ is the tridiagonal matrix with subdiagonal, main diagonal and superdiagonal specified by the signature $(-1, 2, -1)$, just like we wanted to show.

PROBLEM 5

Problem 5a

We now have the vectors $\vec{v}^*$ and $\vec{x}$, both of length m, and the matrix bf of size $n \times n$. This means that we have m number of data points, and that if we wish to solve eq. (13) with the matrix $\mathbf{A}$ we can only solve for n of the data points. This is because we never can multiply a matrix with a vector if the matrix has more or less columns than the amount of elements the vector has. Thus, the number n indicates how many of the values in the complete solution we will be able to calculate by solving the equation.

Problem 5b

If $n < m$ we will only be able to compute the first n elements in the complete solution $\vec{v}^*$, or alternatively the last n elements if we use the last n elements in $\vec{x}$ to compute the elements in $\vec{g}$.

PROBLEM 6

Problem 6a

Now we consider the general tridiagonal matrix with subdiagonal, main diagonal and superdiagonal represented by the vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$, respectively. Such a matrix would have the from

$$\mathbf{A} = \begin{bmatrix} b_0 & c_0 & 0 & \dots & 0 \\ a_0 & b_1 & c_1 & \dots & \vdots \\ 0 & a_1 & b_2 & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & a_{n-2} & b_{n-1} \end{bmatrix} \tag{14}$$

should its dimensions be $n \times n$. To study the general algorithm of computing eq. (13), let us look at the case of $n = 3$. If we augment $\vec{g}$ to $\mathbf{A}$ so that we get

$$[\mathbf{A}\ \vec{g}] = \begin{bmatrix} b_0 & c_0 & 0 & g_0 \\ a_0 & b_1 & c_1 & g_1 \\ 0 & a_1 & b_2 & g_2 \end{bmatrix} \tag{15}$$

we can use the Gaussian elimination method to turn the part of the matrix that is originally just $\mathbf{A}$ into the identity matrix. We could either use forward substitution or backwards substitution to go further, so let's try using the former. First, we can multiply row I of the augmented matrix with a_0/b_0 and then subtract that row from row II:

$$\begin{bmatrix} b_0 & c_0 & 0 & g_0 \\ 0 & b_1 - a_0 c_0/b_0 & c_1 & g_1 - a_0 g_0/b_0 \\ 0 & a_1 & b_2 & g_2 \end{bmatrix} \tag{16}$$

To make the calculations more readable, let's introduce the new variables $\bar{b}_1 = b_1 - a_0 c_0/b_0$ and $\bar{g}_1 = b_1 - a_0 g_0/b_0$. Furthermore, let's now multiply row II with $a_1/\bar{b}_1$ and subtract it from row III so that we're left with

$$\begin{bmatrix} b_0 & c_0 & 0 & g_0 \\ 0 & \bar{b}_1 & c_1 & \bar{g}_1 \\ 0 & 0 & b_2 - a_1 c_1/\bar{b}_1 & g_2 - a_1 \bar{g}_1/\bar{b}_1 \end{bmatrix} \tag{17}$$

Once again we introduce some new variables, $\bar{b}_2 = b_2 - a_1 c_1/\bar{b}_1$ and $\bar{g}_2 = g_2 - a_1 \bar{g}_1/\bar{b}_1$. Dividing row III with $\bar{b}_2$ we're then left with

$$\begin{bmatrix} b_0 & c_0 & 0 & g_0 \\ 0 & \bar{b}_1 & c_1 & \bar{g}_1 \\ 0 & 0 & 1 & v_2 \end{bmatrix} \tag{18}$$

What we're left with on the bottom left is now actually the third value in the solution to the original equation. To find v_2 and v_1 we work our way upward with similar calculations. First we multiply row III with c_1 and subract it from row II, then we divide row II with b_1 so that we have 1 on the diagonal element on row II. Finally we multiply row II with c_0 and subtract it from row I, then dividing row I with b_0 so that we're left with

$$\begin{bmatrix} 1 & 0 & 0 & v_0 \\ 0 & 1 & 0 & v_1 \\ 0 & 0 & 1 & v_2 \end{bmatrix} \tag{19}$$

As we can see we're now left with the vector $\vec{v}$ that would satisfy eq. (13) in column IV. This process is analogous for any equation where $\mathbf{A}$ is an $m \times n$ matrix, $\vec{v}$ is a vector of size n and $\vec{g}$ is a vector of size m. A general pseudo code of this process when $\mathbf{A}$ is a tridiagonal matrix could look like in algorithm 1. Here it is assumed that one would first create a new matrix $\mathbf{A}g$ that is a copy of $\mathbf{A}$ with $\vec{g}$ concatenated to it.

Algorithm 1 Pseudo code for solving $\mathbf{A}\vec{v} = \vec{g}$ for $\vec{v}$

for $i = 0, 1, ..., n-2$ **do** ▷ in this loop we get rid of the subdiagonal in the augmented matrix
  $newrow = \mathbf{A}g[i]\mathbf{A}g[i+1,i]/\mathbf{A}g[i,i]$;
  $\mathbf{A}g[i+1] = \mathbf{A}g[i+1] - newrow$; ▷ $a_i = 0$, $b_{i+1} = b_{i+1} - c_i a_i/b_i$ and $g_{i+1} = g_{i+1} - g_i a_i/b_i$

for $i = n-1, n-2, ..., 1$ **do** ▷ in this loop we get rid of the superdiagonal and normalize the diagonal elements
  $\mathbf{A}g[i] = \mathbf{A}g[i]/\mathbf{A}g[i,i]$; ▷ $v_i = \overline{g}_i/\overline{b}_i$
  $\mathbf{A}g[i-1] = \mathbf{A}g[i-1] - \mathbf{A}g[i-1,i]\mathbf{A}g[i]$; ▷ $v_{i-1} = v_{i-1} - c_{i-1}v_i$

▷ lastly we impose the boundary conditions
$\mathbf{A}g[0,n] = 0.0$; ▷ $v_0 = 0.0$
$\mathbf{A}g[n-1,n] = 0.0$; ▷ $v_{n-1} = 0.0$

Problem 6b

In algorithm 1 we do two FLOPs to three nonzero matrix elements in the first line of the first loop, and then do one FLOP to three matrix elements in the next line. Since we do this $n-1$ times, this loop contains a total amount of $(2*3)(n-1)+3(n-1) = 9(n-1)$ FLOPs. In the next loop we do one FLOP to two nonzero matrix elements in the first line, while in the second line we do one FLOP to two nonzero matrix elements and one FLOP to two other nonzero matrix elements. Since we do this $n-1$ times as well, this loop contains a total of $2(n-1)+2(n-1)+2(n-1) = 6(n-1)$ FLOPs. Thus, the algorithm contains $15(n-1)$ FLOPs. However, in order to apply this algorithm to our program, we had to use vectors instead of matrices because the matrices take up too much storage for sufficiently large values of n. To further optimalize our code we only operated with the vector elements different from zero, instead of operating with the entire row vectors. Thus, we ended up using algorithm 2. In this algorithm we do six FLOPs in the first loop, which we do $n-1$ times, and in the second loop we do three FLOPs, which we also do $n-1$ times. Thus, the optimalized algorithm contains $9(n-1)$ FLOPs.

PROBLEM 7

Problem 7a

When implementing algorithm 1 in C++, we quickly stumbled into a memory issue. To run this algorithm for $n = 10^7$ we would need to create a matrix of size $10^7 \times 10^7$, which would require $10^7 \times 10^7 \times 8 = 8 \times 10^{14}$ bytes of memory when using doubles. This is equal to 800 TB, which is more than the amount of memory on the computers we're using. Thus, we needed to find a way to solve this problem without creating such a large matrix. Looking at the algorithim, we notice we only needed to calculate two rows at a time. Therefore, we could use the armadillo library to store the two rows as vectors. This allowed us to solve the problem, but this approach was still quite slow. We calculated a runtime of approximately 8-16 hours with this approach. Finally we noticed that we don't need to store the entire rows, only the nonzero elements in two rows at a time. This pattern is easy to see in the case of $n = 5$ as seen in fig. 2. In the first loop, we only use index 1, 2 and n in row 1, and index 2, 3, n in row 2. In the second loop, row 1 replaces row 2 and the indexes of interest in row 2 increase by one. To copy over n elements is computationally heavy and a foor-loop in of itself. By noticing the fact that there are only three indices of interest, we can store just 3 values instead of n. This reduces the time complexity from $O(n^2)$ to $O(n)$ and increased computation speed by a factor of 125'000-250'000.

The final function we wrote which implemented the general algorithm is found in the file `find_v_general.cpp`. In `main.cpp` we call on this function for $n_{\text{steps}} = 10, 100, 1000, 10^4, 10^5, 10^6, 10^7$. We also call on the function `write_to_file` for each of these values of n_{steps} to store the solutions $\vec{v}$ and the corresponding vectors $\vec{x}$ to binary files. We chose to write the data to binary files to save time.

Problem 7b

After running our program for all the seven values of n_{steps} we used our script `plot_data.py` to plot the numerical solutions along with the exact solution $u(x)$. The resulting plot is shown in fig. 3. In fig. 4 we plotted the numerical solutions without the one corresponding to $n_{\text{steps}} = 10$, as this one deviated so much that it was difficult to decipher the precision of the other solutions.

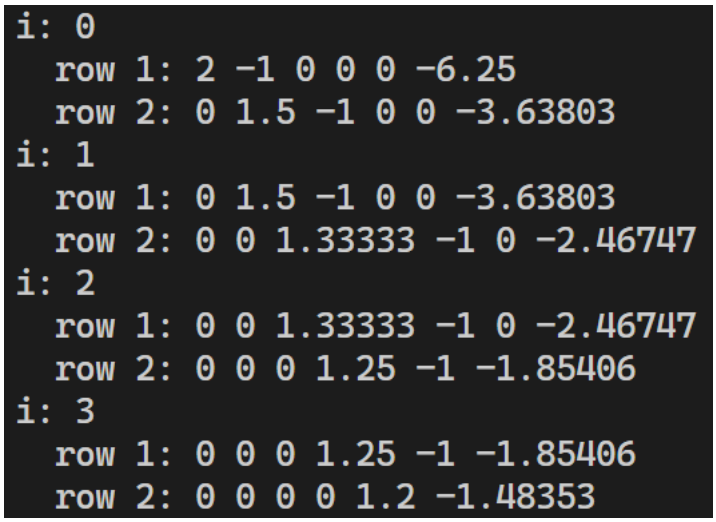

FIG. 2. Screenshot of terminal output showing calculations for $n = 5$

Algorithm 2 Optimalized pseudo code for solving $\mathbf{A}\vec{v} = \vec{g}$ for $\vec{v}$

▷ before the first loop we specify the nonzero elements of the first row in the augmented matrix, denoted $\mathbf{A}g_0$
$\mathbf{A}g_{0,0} = b[0]$; ▷ first element in first row is b_0
$\mathbf{A}g_{0,1} = c[0]$; ▷ second element in first row is c_0
$\mathbf{A}g_{0,n} = g[0]$; ▷ last element in first row is g_0

for $i = 0, 1, ..., n-2$ **do** ▷ in this loop we get rid of the subdiagonal in the augmented matrix
▷ here we specify the nonzero elements of row no. $i+1$, denoted $\mathbf{A}g_{i+1}$
$\mathbf{A}g_{i+1,i} = a[i]$; ▷ i'th element in row no. $i+1$ is a_i
$\mathbf{A}g_{i+1,i+1} = b[i+1]$; ▷ $(i+1)$'th element in row no. $i+1$ is b_{i+1}
if $i < n-2$ **then** ▷ on the last row we don't have any c-value
$\mathbf{A}g_{i+1,i+2} = c[i+1]$; ▷ $(i+2)$'th element in row no. $i+1$ is c_{i+1}
$\mathbf{A}g_{i+1,n} = g[i+1]$; ▷ n'th element in row no. $i+1$ is g_{i+1}

▷ here we perform the matrix operations between row $\mathbf{A}g_i$ and row $\mathbf{A}g_{i+1}$ only on the nonzero row elements
$\mathbf{A}g_{i+1,i+1} = \mathbf{A}g_{i+1,i+1} - \mathbf{A}g_{i,i+1}\mathbf{A}g_{i+1,i}/\mathbf{A}g_{i,i}$; ▷ $b_{i+1} = b_{i+1} - c_i a_i/b_i$
$\mathbf{A}g_{i+1,n} = \mathbf{A}g_{i+1,n} - \mathbf{A}g_{i,n}\mathbf{A}g_{i+1,i}/\mathbf{A}g_{i,i}$; ▷ $g_{i+1} = g_{i+1} - g_i a_i/b_i$
$\mathbf{A}g_{i+1,i} = 0.0$; ▷ $a_i = a_i - b_i a_i/b_i = 0$

▷ here we store the modified diagonal element and the modified g_i for the next loop
$diag[i] = \mathbf{A}g_{i,i}$; ▷ stores $\overline{b}_i$ for later use
$v[i] = \mathbf{A}g_{i,n}$; ▷ stores $\overline{g}_i$ for later use

▷ here we essentially want to replace row $\mathbf{A}g_i$ with row $\mathbf{A}g_{i+1}$ for the next operations
$\mathbf{A}g_{i,i} = \mathbf{A}g_{i+1,i+1}$; ▷ replaces $\overline{b}_i$ with b_{i+1}
$\mathbf{A}g_{i,i+1} = \mathbf{A}g_{i+1,i+2}$; ▷ replaces c_i with c_{i+1}
$\mathbf{A}g_{i,n} = \mathbf{A}g_{i+1,n}$; ▷ replaces $\overline{g}_i$ with g_{i+1}

▷ the $(i+1)$'th row is now the last row, so we store the nonzero values from this row in $diag$ and v
$diag[n-1] = \mathbf{A}g_{i+1,i}$; ▷ stores $\overline{b}_{n-1}$ for later use
$v[n-1] = \mathbf{A}g_{i+1,n}$; ▷ stores $\overline{g}_{n-1}$ for later use

for $i = n-1, n-2, ..., 1$ **do** ▷ in this loop we get rid of the superdiagonal and normalize the diagonal elements
$v[i] = v[i]/diag[i]$; ▷ $v_i = \overline{g}_i/\overline{b}_i$
$v[i-1] = v[i-1] - c[i-1]v[i]$; ▷ $v_{i-1} = v_{i-1} - c_{i-1}v_i$

▷ lastly we impose the boundary conditions
$v[0] = 0.0$; ▷ $v_0 = 0.0$
$v[n-1] = 0.0$; ▷ $v_{n-1} = 0.0$

Problem 7b

After running our program for all the seven values of n_{steps} we used our script `plot_data.py` to plot the numerical solutions along with the exact solution $u(x)$. The resulting plot is shown in fig. 3. In fig. 4 we plotted the numerical solutions without the one corresponding to $n_{\text{steps}} = 10$, as this one deviated so much that it was difficult to decipher the precision of the other solutions.

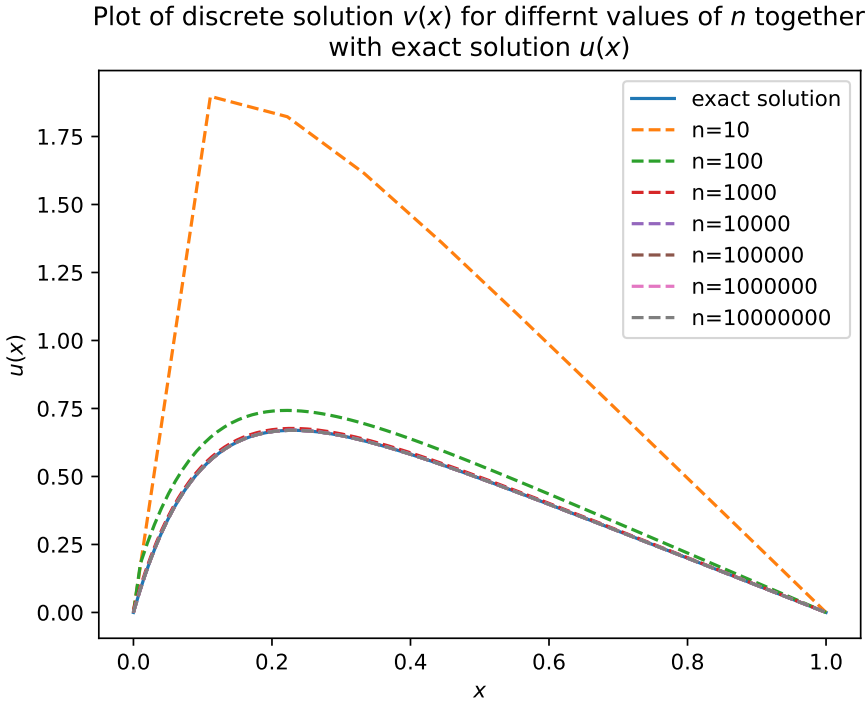

FIG. 3. Plot of the numerical solutions $v(x)$ for seven different values of n_{step} (dashed lines) along with the exact solution $u(x)$ (solid line).

PROBLEM 8

Problem 8a

In fig. 5 we see a plot of the logarithm of the absolute errors

$$\log_{10}(\Delta_i) = \log_{10}(|u_i - v_i|) \tag{20}$$

of the numerical solutions as functions of x_i for the seven different values of n_{step}.

Problem 8b

In fig. 6 we instead see a plot of the logarithm of the relative errors

$$\log_{10}(\epsilon_i) = \log_{10}\left(\left|\frac{u_i - v_i}{u_i}\right|\right) \tag{21}$$

of the numerical solutions as functions of x_i for the seven different values of n_{step}.

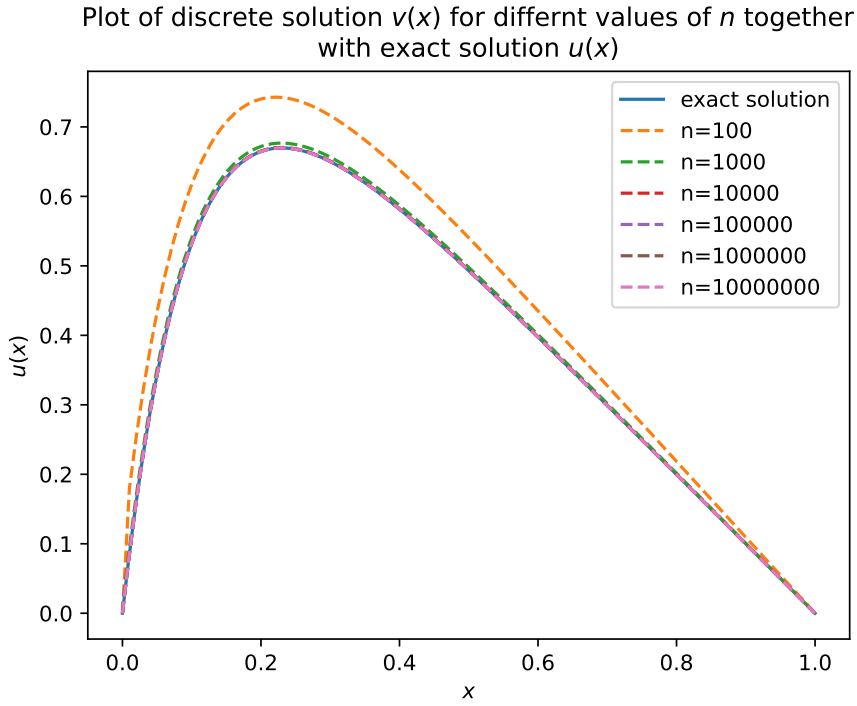

FIG. 4. Plot of the numerical solutions $v(x)$ (dashed lines) without $n_{\text{step}} = 10$ along with the exact solution $u(x)$ (solid line).

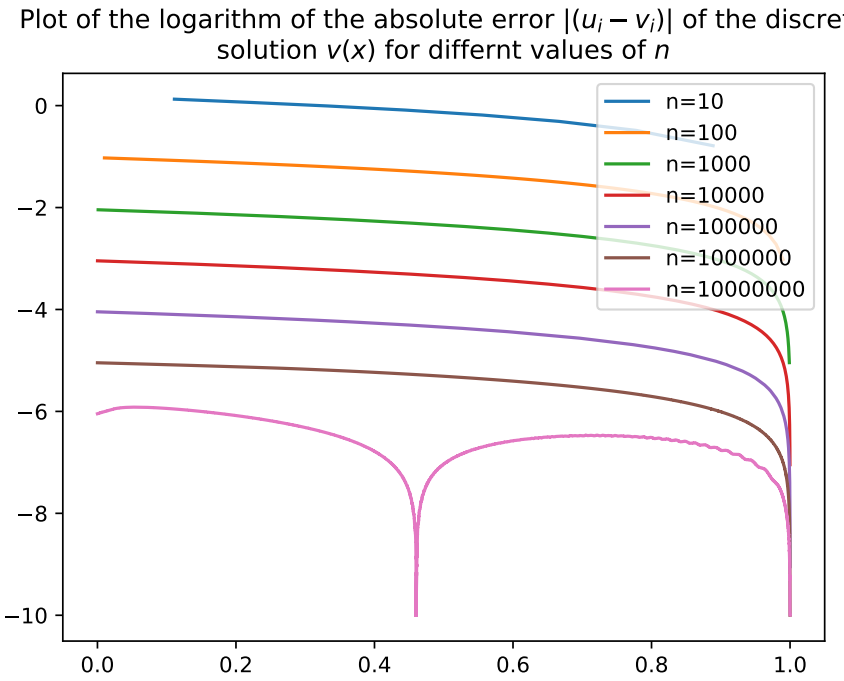

FIG. 5. Plot of the logarithm of the numerical solutions' absolute errors $\log_{10}(\Delta_i)$ $v(x)$ for the seven different values of n_{step}.

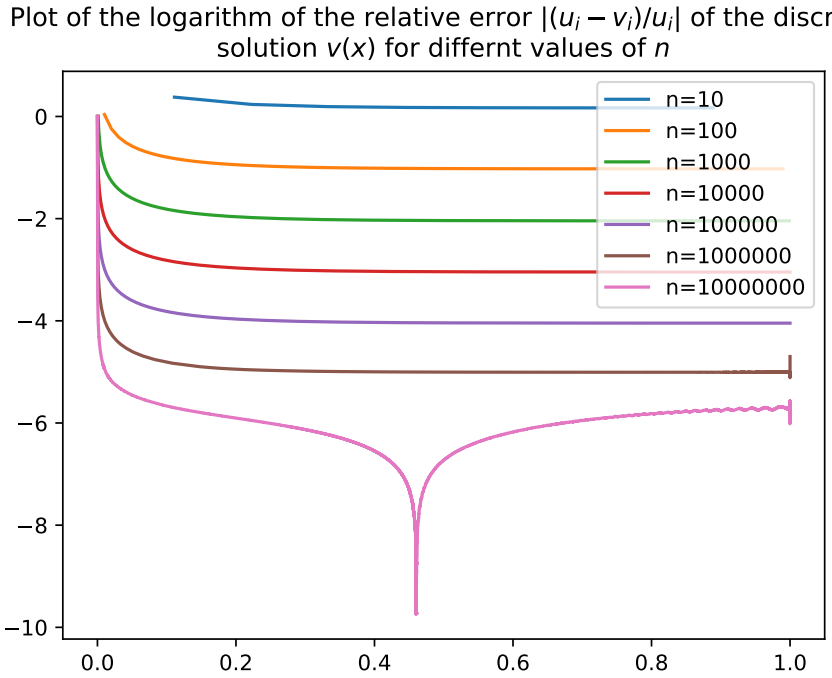

FIG. 6. Plot of the logarithm of the numerical solutions' relative errors $\log_{10}(\epsilon_i)$ $v(x)$ for the seven different values of n_{step}.

TABLE I. Table showing the maximum relative error $\max(\epsilon_i)$ of the numerical solutions along with the corresponding x-values x_i, for seven values of n_{step}.

n_{steps}	$\max(\epsilon_i)$	x_i
10	2.390320018853711	0.1111111111
10^2	1.094617632100603	0.0101010101
10^3	1.0091451046464817	0.001001001
10^4	1.0009115217009974	0.00010001
10^5	1.0000900036001439	1.00001×10^{-05}
10^6	1.0000111111111112	10^{-6}
10^7	1.0	10^{-7}

Problem 8c

Table I shows the maximum relative errors of the numerical solutions for the seven values of n_{steps}, along with the corresponding x-values where the errors were largest. We see that the errors are largest near $x = 0$, but naturally not at $x = 0$ since we set the first and last values of $\vec{v}$ to zero. This is not surprising considering the plot of the logarithm of the relative errors, as these all peak near $x = 0$ This is due to the fact that we divide by u_i when calculating ϵ_i, and the first u_i are very close to zero, as well as $u_i - v_i$ being close to zero.

PROBLEM 9

Problem 9a

To specialize our algorithm from Problem 6 where $\mathbf{A}$ is specified by the signature $(-1, 2, -1)$ we essentially used the general algorithm. The only difference is that the function `find_v_special` we created that is defined in `find_v_special.cpp` doesn't take in vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$. Instead the constant doubles $a = -1.0$, $b = 2.0$ and $c = -1.0$ are defined locally within the function.

Problem 9b

The special algorithm contains the exact same number of FLOPs as the general algorithm, which is $9(n-1)$.

Problem 9c

See `find_v_special.cpp`.

PROBLEM 10

Figures 7, 8, 9, 10, 11 and 12 show comparisons of fice runtimes with the general algorithm and the special algorithm for $n_{\text{steps}} = 10, 100, 1000, 10^4, 10^5, 10^6$ respectively. The diagrams also compare the mean of the five runtimes in each case. As expected, considering there is an equal amount of FLOPs in them both, there are minor differences between the two algorithms, and these may very well be random. It does however seem like the general algorithm tends to be somewhat slower than the special algorithm, which may have something to do with the fact that the general algorithm picks out elements from the vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$ before performing computations.

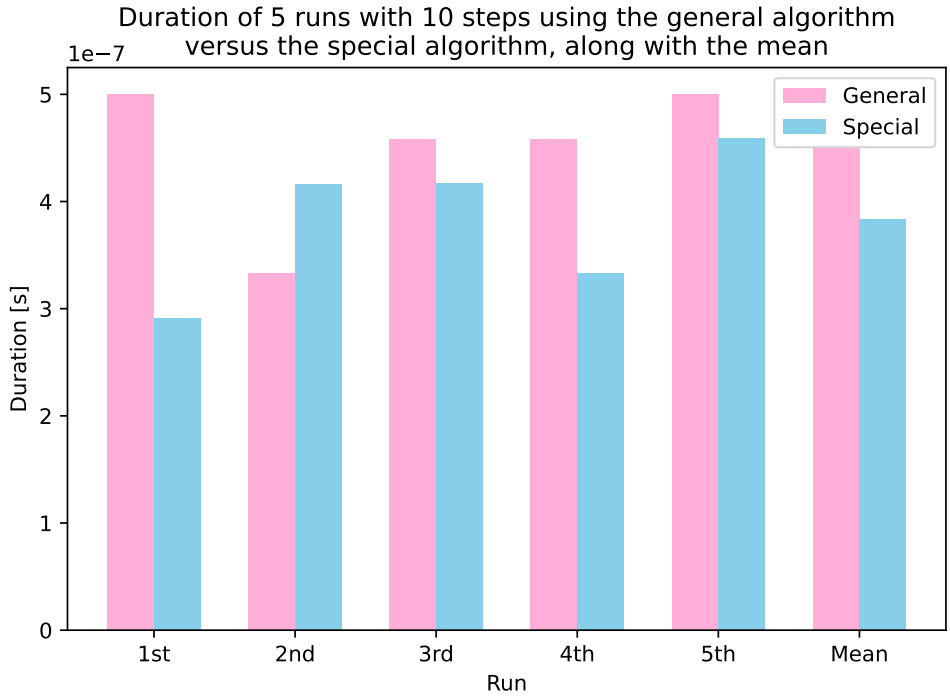

FIG. 7. Comparison of runtimes for $n_{\text{step}} = 10$ from five seperate runs as well as the mean of these using the general algorithm (pink) and special algorithm (blue).

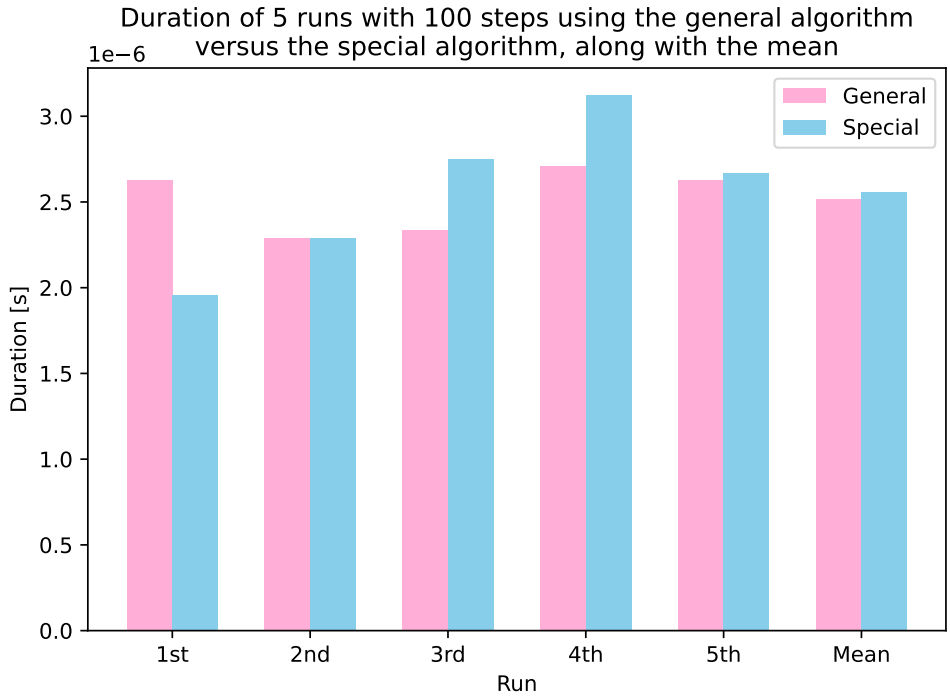

FIG. 8. Comparison of runtimes for $n_{\text{step}} = 100$ from five seperate runs as well as the mean of these using the general algorithm (pink) and special algorithm (blue).

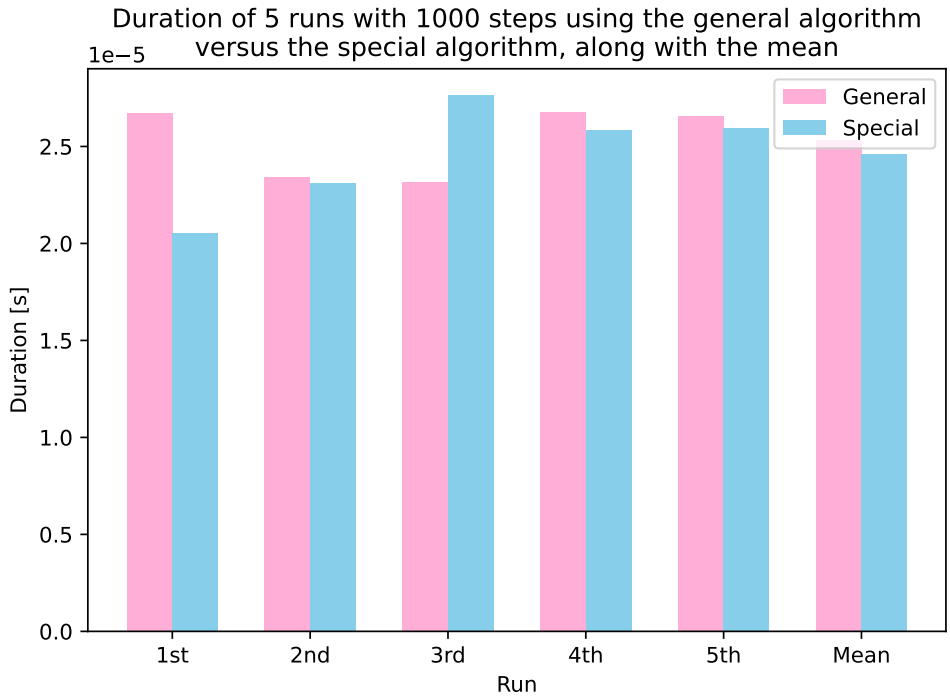

FIG. 9. Comparison of runtimes for $n_{\text{step}} = 1000$ from five seperate runs as well as the mean of these using the general algorithm (pink) and special algorithm (blue).

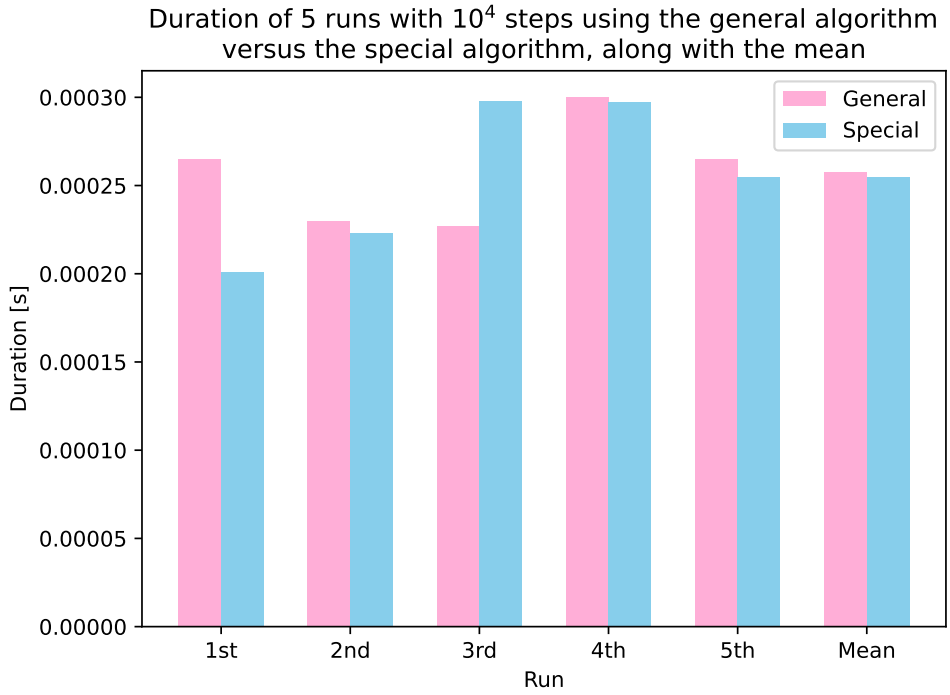

FIG. 10. Comparison of runtimes for $n_{\text{step}} = 10^4$ from five seperate runs as well as the mean of these using the general algorithm (pink) and special algorithm (blue).

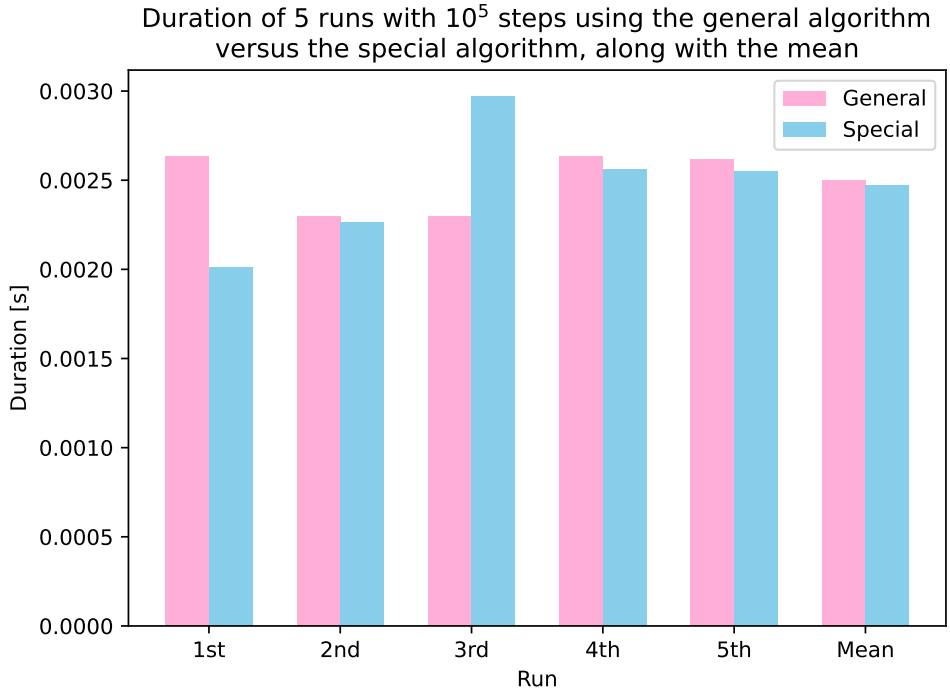

FIG. 11. Comparison of runtimes for $n_{\text{step}} = 10^5$ from five seperate runs as well as the mean of these using the general algorithm (pink) and special algorithm (blue).

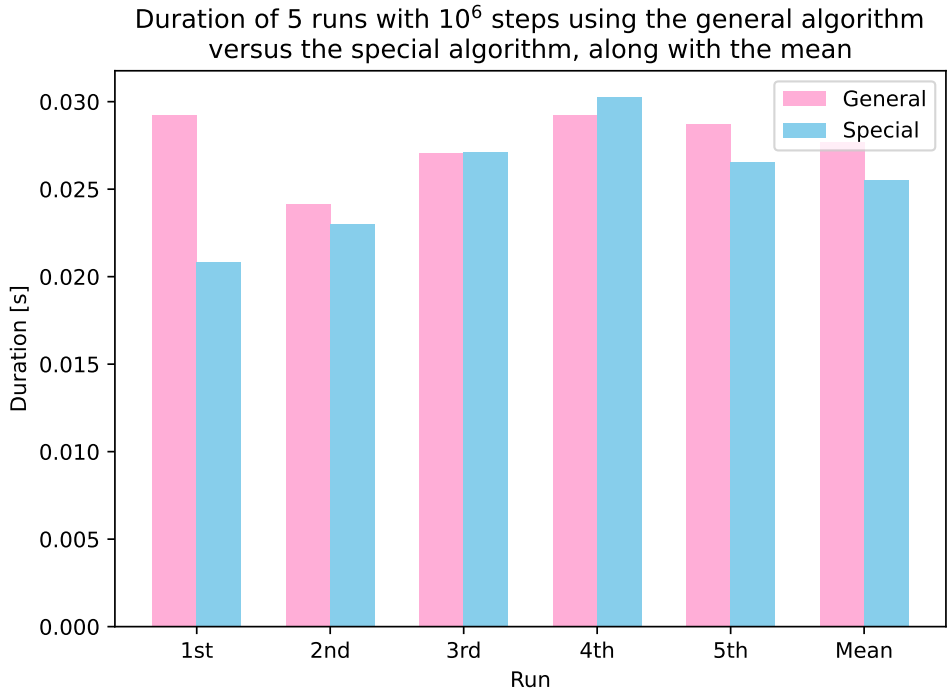

FIG. 12. Comparison of runtimes for $n_{\text{step}} = 10^6$ from five seperate runs as well as the mean of these using the general algorithm (pink) and special algorithm (blue).