Project 1

For the course FYS3150

Erik Grammeltvedt, Erlend Tiberg North and Alexandra Jahr Kolstad

September 10, 2019
Week 35 - 37

Contents

1 Abstract

In this project we have studied the importance of second-order differential equations exemplified in this project through Poisson's equation. We have used the set of equations an ordinary differential equation gives, and tried solving them using three different numeric methods. We have computed results from matrix size $n = 10^1$ to $n = 10^7$. We found that the decomposition we made using the special case of our matrix gave the closest results to the given solution and had the shortest run time. The more general method, Thomas-algorithm and the LU-decomposition were therefore inferior in this specific case. The Thomas-algorithm had a higher number of floating point operations, FLOPS, per iteration and the LU-decomposition demanded so much RAM that it crashed for matrices of dimensions $n = 10^5$ and higher. This task does not only show efficiency of different methods, it shows the power of an algorithm that is specifically made for the problem you are solving.

2 Introduction

In this project we are going to solve Poisson's equation as a set of linear equations. Poisson's equation is a partial differential equation written in the file **Project1.pdf**. This equation can be rewritten as an ordinary differential equation

$$-u''(x) = f(x) \tag{1}$$

The functions we are operating with are

$$f(x) = 100e^{-10x} \tag{2}$$

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

These are also given in **Project1.pdf**. We are going to solve the ordinary differential equation by computing the decomposition and forward substitution and the backward substitution of the given matrices with the programming language `C++`. Our group is using the Armadillo package to more easily define and compute with matrices. We are also going to work with and try to understand dynamic memory allocation. The main exercise is grouped into smaller sub exercises ranging from **a)** to **e)**.

For the exercises **b)**, **c)**, **d)** and **e)** the main program is `thomas-algorithm.cpp`. For **b)**, **c)** and **e)** there is an additional program called `plot_data.py` which plots the different algorithm's solutions and compares them to equation (3). For exercise **c)** there is also the program `timing.py` for comparing the time used by the general and the special algorithm. For **d)** we also have `error.py` which computes the error for the algorithm in **b)**. All programs are found at our GitHub-repository.

3 Method

In this section the method for writing the programs and the programs themselves are described in separate subsubsections, with subsections given by the exercises.

3.1 Exercise a)

In the exercise we are given the equation

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \qquad \text{for } i = 1, 2, 3, \dots, n$$

We rewrite the equation to

$$\begin{gathered}-(v_{i+1} + v_{i-1} - 2v_i) = h^2 f_i = \tilde{b}_i \\ -v_{i+1} - v_{i-1} + 2v_i = \tilde{b}_i\end{gathered}$$

In the exercise we are also given the correlation $\tilde{b}_i = h^2 f_i$, which is implemented here.

We define the equation for different values of the integer i to get a set of equations. The exercise also gives the boundry conditions $v_0 = v_{n+1} = 0$.

$$
\begin{aligned}
i=1: &\quad -v_{1+1}-v_{1-1}+2v_1 = -v_2-v_0+2v_1 = -0+2v_1-v_2 = \tilde{b}_1\\
i=2: &\quad -v_{2+1}-v_{2-1}+2v_2 = -v_3-v_1+2v_2 = -v_1+2v_2-v_3 = \tilde{b}_2\\
i=3: &\quad -v_{3+1}-v_{3-1}+2v_3 = -v_4-v_2+2v_3 = -v_2+2v_3-v_4 = \tilde{b}_3\\
\vdots &\\
i=n: &\quad -v_{n+1}-v_{n-1}+2v_n = -v_{n-1}+2v_n-0 = \tilde{b}_n
\end{aligned}
$$

Equations can be rewritten as a matrix equation, which gives a matrix **A** with integers as elements, a vector $\vec{v} = [v_1, v_2, v_3, \ldots, v_n]$ and another vector $\vec{\tilde{b}} = [\tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \ldots, \tilde{b}_n]$. This gives the matrix equation

$$\mathbf{A}\vec{v} = \vec{\tilde{b}} \tag{4}$$

where $\vec{v}$ is the solution. The matrix and the vectors are given as

$$
\begin{bmatrix}
2 & -1 & 0 & 0 & \ldots & 0\\
-1 & 2 & -1 & 0 & \ldots & 0\\
0 & -1 & 2 & -1 & \ldots & 0\\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots\\
0 & 0 & 0 & -1 & 2 & -1\\
0 & 0 & 0 & 0 & -1 & 2
\end{bmatrix}
\begin{bmatrix} v_1\\ v_2\\ v_3\\ \vdots\\ v_{n-1}\\ v_n \end{bmatrix}
=
\begin{bmatrix} \tilde{b}_1\\ \tilde{b}_2\\ \tilde{b}_3\\ \vdots\\ \tilde{b}_{n-1}\\ \tilde{b}_n \end{bmatrix}
\tag{5}
$$

Therefore the matrix equation has been proved.

3.2 Exercise b)

3.2.1 Calculations

For the forward substitution algorithm there are $5(n-1)$ FLOPS. For the backward substitution algorithm there are $3(n-2)$ FLOPS. We precalculate v's last element and that is an additional FLOP. The total number of floating point operations is therefore $8n-10$.

In this exercise we compute with the matrix equation

$$\begin{bmatrix} d_1 & c_1 & 0 & 0 & \dots & 0 \\ a_1 & d_2 & c_2 & 0 & \dots & 0 \\ 0 & a_2 & d_3 & c_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_{n-2} & d_{n-1} & c_{n-1} \\ 0 & 0 & 0 & 0 & a_{n-1} & d_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

where $b_i = h^2 f_i$ in the rest of the calculations for the exercises. Rewriting this matrix equation into a set of linear equations, we can try to solve for $\vec{v}$. However, this prove to not be easily done, as we get a lot of unknown variables. For instance after a couple of matrix operations we get the equation

$$\left(d_2 - \frac{a_1 c_1}{d_1}\right) v_2 + c_2 v_3 = b_2 - b_1 \frac{a_1}{d_1}$$

We can shorten this equation by setting $\tilde{d}_2 = d_2 - \frac{a_1 c_1}{d_1}$ and $\tilde{b}_2 = b_2 - b_1 \frac{a_1}{d_1}$. Furthermore we also get the following equation after some matrix operations

$$\left(d_3 - \frac{a_2 c_2}{\tilde{d}_2}\right) v_3 + c_3 v_4 = b_3 - \tilde{b}_2 \frac{a_2}{\tilde{d}_2}$$

where we define $\tilde{d}_3 = d_3 - \frac{a_2 c_2}{\tilde{d}_2}$ and $\tilde{b}_3 = b_3 - \tilde{b}_2 \frac{a_2}{\tilde{d}_2}$. This leads to a general formula for $\tilde{d}_i$ and $\tilde{b}_i$. For the forward substitution the equations used in the algorithm are

$$\tilde{d}_i = d_i - \frac{a_{i-1} c_{i-1}}{\tilde{d}_{i-1}} \tag{6}$$

$$\tilde{b}_i = b_i - \frac{\tilde{b}_{i-1} a_{i-1}}{\tilde{d}_{i-1}} \tag{7}$$

with the condition that $\tilde{d}_1 = d_1$. d is a vector given by $d = [d_1, d_2, \dots, d_n]$, a is a vector given by $a = [a_1, a_2, \dots, a_{n-1}]$, and c is a vector given by $c = [c_1, c_2, \dots, c_{n-1}]$.

Now we can more easily find $\vec{v}$. The new equations from the forward substitution gives a new matrix equation after some matrix operations

$$\begin{bmatrix} d_1 & c_1 & 0 & 0 & \dots & 0 \\ 0 & \tilde{d}_2 & c_2 & 0 & \dots & 0 \\ 0 & 0 & \tilde{d}_3 & c_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \tilde{d}_{n-1} & c_{n-1} \\ 0 & 0 & 0 & 0 & 0 & \tilde{d}_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \vdots \\ \tilde{b}_{n-1} \\ \tilde{b}_n \end{bmatrix}$$

This can also be rewritten as a set of linear equations. After a few matrix operations we can for instance find that

$$v_4 = \frac{-c_4 v_5 + \tilde{b}_4}{\tilde{d}_4}$$

Finally we can find a simple equation for $\vec{v}$, given by that we can find a general formula for the vector based on matrix operations such as above. For the backward substitution the equation used in the algorithm is

$$v_i = \frac{\tilde{b}_i - c_i v_{i+1}}{\tilde{d}_i} \tag{8}$$

with the conditions that $v_0 = 0$ and $v_n = \frac{\tilde{b}_n}{\tilde{d}_n}$.

3.2.2 The programming

First we are going to look at the program `thomas-algorithm.cpp`. This is the main program for this exercise, which computes the decomposition and forward substitution and the backward substitution of the given matrices. More dispriptive comments are included in the respective program on GitHub. The file consists of programming for other exercises as well, so exercise **b)** is referred to as general algorithm. All operations for this exercise will be commented with this name. The code asks the user for an input for the variable n, which is the dimension for the matrix $\mathbf{A}$ and the length of the vectors $\vec{v}$ and $\vec{b}$. The clock starts at this point in the program. Then the computer runs through a for-loop to compute the new arrays `ad`, `d_new` and `b_tld_new` for the forward substitution. They are respectively the variable $\frac{a_{i-1}}{\tilde{d}_{i-1}}$, the new diagonal elements to $\mathbf{A}$ and the new array $\vec{\tilde{b}}$. The new variable and arrays are described in the equations (6) and (7). Furthermore the backward substitution is written in a for-loop which generates the vector $\vec{v}$ based on equation (8). After this for-loop the clock is stopped and the time is printed to the terminal. The program then makes `.txt`-files which includes the data for the array x, vector $\vec{v}$ for the general and special algorithm and for the LU-decomposition, as well as the solution $u(x)$. These files are made in a different directory than the main program `thomas-algorithm.cpp`, so be aware that the files are made in two directories up.

Secondly we are going to look at the program `plot_data.py`. The code requests an input from the user in the terminal window requesting a file. This file will be the "filename" and the file that will be read in the program. The program now opens the file and reads it. It then divides the program into different lines and divides each line into five different elements that the for-loop fills into the five empty lists in the top of the program. The lists are named after the variables stored in them. The values of x from the main program is stored in the `x` list. The list `v_gen` is filled with the values from the general function in the main program. The list `v_spl` contains the values given by the special function in the main program. The list `v_LU` has the values of the LU-decomposition from the main program. Finally the list `u` contains the values of the function $u(x)$ in the main program. The program then converts the lists to arrays using the numpy library. Then the `try` function discards the four first and four last characters for naming purposes in the `plot` function. If the file passes the `try` function the arrays go to plotting and the graph is generated using the pylab library in python.

3.3 Exercise c)

3.3.1 Calculations

For the forward substitution algorithm there are $2(n-1)$ FLOPS. For the backward substitution algorithm there are $2(n-2)$ FLOPS. We precalculate v's last element and that is an additional FLOP. In total we then have $4n-5$ number of FLOPS.

The equation for the elements on the diagonal is given in (6). This equation is derived from forward substitution from calculations for **b)**, 3.2.

With the knowledge that the diagonal elements are $a_1 = a_2 = \ldots = a_{n-1} = -1$ and $c_1 = c_2 = \ldots = c_{n-1} = -1$ we can shorten the equation to the form

$$\tilde{d_i} = d_i - \frac{1}{\tilde{d}_{i-1}}$$

When asserting different integer values for i we can compute the elements.

$$
\begin{aligned}
\tilde{d}_1 &= d_1 = 2 \\
\tilde{d}_2 &= 2 - \frac{1}{2} = \frac{3}{2} \\
\tilde{d}_3 &= 2 - \frac{1}{\frac{3}{2}} = \frac{4}{3} \\
\tilde{d}_4 &= 2 - \frac{1}{\frac{4}{3}} = \frac{5}{4} \\
&\vdots \\
\tilde{d}_n &= 2 - \frac{1}{1} = 1
\end{aligned}
$$

From this we can derive a general formula for the diagonal elements

$$\tilde{d}_i = \frac{i+1}{i}$$

Forward substitution is also given by equation (7). We can shorten this equation with the same knowledge from earlier that $a_1 = a_2 = \ldots = a_{n-1} = -1$.

$$\tilde{b}_i = b_i + \frac{\tilde{b}_{i-1}}{\tilde{d}_{i-1}}$$

This is the equation used for the forward substitution in our special algorithm.

The equation for backward substitution is equation (8). As with the forward substitution we can shorten this equation with the knowledge from earlier that $c_1 = c_2 = \ldots = c_{n-1} = -1$.

$$\tilde{v}_i = \frac{\tilde{b}_i + v_{i+1}}{\tilde{d}_i}$$

This is the equation used for the backward substitution in our special algorithm.

3.3.2 The programming

The program for this exercise is also given in the file `thomas-algorithm.cpp`. The structure of finding the solution $\vec{v}$ for the matrix given in this exercise is the same as for the matrix in exercise **b)**, 3.2. It is only different because this matrix has predefined matrix elements, so we can precalculate the values for the new diagonal to spare usage of FLOPS.

In this exercise we also compute with the program `plot_data.py`. However, there are no differences between the algorithm in this exercise and in exercise **b)**, 3.2, where the program is commented, so for an explanation of the program please read the associated subsubsection **3.2.2**.

Lastly we have the file `timing.py`. The purpose of this program is to make a table for the run times in order to compare them. The program starts with creating lists for the data. It then reads the given file and fills the lists with data using the for-loop. Then the number of data points and the initialized average for the two lists are given. After that the for-loop generates the averages and the code underneath makes the values in the list into actual averages. Lastly the time average for each program is printed.

3.4 Exercise d)

3.4.1 Calculations

In this task we are asked to make a program that calculates the relative error of the numeric method compared to the given formula.

$$\varepsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{9}$$

The formula is given as a function of $\log_{10}(h)$ for each value of v_i and u_i, given in equations (8) and (3). When this is computed each step length will give several values of x. Both u and v are a function of x and will therefore have several values within one step length of h. Within one step length we extract the biggest value of ε_i and make a plot comparing the difference from $u(x)$ to $v(x)$ over x. We decided to make a plot over a table because we felt like it better visualizes the error for our task.

3.4.2 The programming

The program for this problem is located under the subheading; *computing errors for general-algorithm* in the main program `thomas-algorithm.cpp`. The program starts by setting up an epsilon max array to store the values that will come out of the for-loop. The for-loop runs through the ε_i function described in equation (9) and takes out the biggest error through an if-else statement. Then the array is stored in a `.txt`-file which we use to show the graphs through a self made python program.

The python graph generating program, `error.py`, starts by defining the file name and making lists to store the information from the file. Then opens and reads the information from the file using a for-loop. The for-loop divides the strings in the line with split and then allocates the first value of the line to the array `log_h` and the second element in the line to `log_error`. Then turns the lists into arrays and plots the function using pylab's plotting functions.

3.5 Exercise e)

3.5.1 Calculations

In this task we look at the LU-decomposition as a potential solver instead of the general and special Thomas algorithm.

The general number of FLOPS of the LU-decomposition is given by $\frac{2}{3}n^3$. This will be true for any LU-decomposition, where n is the dimensions of the $(n \times n)$ matrix.

3.5.2 The programming

The program that answers to this task starts with the subheading *A-matrix for LU-decomposition* and ends with the subheading *display duration*, and is found in the file `thomas-algorithm.cpp`. The purpose of this code is to do the LU-decomposition of the matrix $\mathbf{A}$ and calculate its processing time. First the program generates the matrix $\mathbf{A}$ using Armadillo and a for-loop. Armadillo fills the matrix with zeros and the for-loop fills the three diagonals with their respective values -1 , 2 and -1. The next part of the program; *make a vector of the pointer-array b_ tld*, converts the `b_tld` vector through a for-loop by using Armadillo's vector function. Now the clock starts and the LU-decomposition begins. First the LU-decomposition is done by using the `mat` and `lu` functions from Armadillo. The program uses these values to allow the function `solve` from Armadillo to give the answer for the LU-decomposition as vectors. Then the clock stops and the difference between time start and stop is done by a simple subtraction and the program closes.

In this exercise we also compute with the program `plot_data.py`. However, there are no differences between the algorithm in this exercise and in exercise **b)**, 3.2, where the program is commented, so for an explanation of the program please read the associated subsubsection **3.2.2**.

4 Results and discussion

Our results are as shown in the Appendix. We also have `.txt`-files for all the raw data generated by the projects up on GitHub.

4.1 Exercise a)

The results of this exercise is the matrix equation (5).

4.2 Exercise b)

In figures (1), (2) and (3) we see how the general Thomas-algorithm compares to the other algorithms and the exact solution. We can clearly see that the algorithm benefits from the larger n (as it should). The approximation starts off as a so-so good approximation. However, it gains accuracy fairly quickly for each n we run through. All data was made by running `thomas-algorithm.cpp` in **QtCreator**. See the corresponding `.txt`-files for the raw data. The raw data was put in `plot_data.py` which created the actual plots. To read more about how the programs work, see 3.2.

Ignoring the initial-condition we chose to calculate mid-algorithm and looking at one iteration we have 8 FLOPS.

4.3 Exercise c)

In `thomas-algorithm.cpp` additional code was added. Using the new equations (see 3.3) we created a specialized algorithm for our given matrix. Again, by looking at figures (1), (2) and (3) we see the algorithm's data compared to the others and the exact solution. We can actually see that the special algorithm is more accurate than the general algorithm from the get-go. This could be because of the fewer FLOPS (fewer chances for numerical errors). However, this would be more visible at higher values of n. By debugging the program (printing `d_new` for each of the algorithms) we see that they in fact are not the same. By inserting the following code after filling each array we can see the differences

```
    std::cout << "index,  value\n";
  for(int i = 0; i < n; i++){
      std::cout << "    " << i << ", " << d_new[i] << std::endl;
  }

```

And the differences are as follows

```
index, value
    0, 2
    1, 1.5
    2, 1.33333
    3, 1.25
    4, 1.2
    5, 1.16667
    6, 1.14286
    7, 1.125
    8, 1.11111
    9, 1.1
index, value
    0, 2
```

```
1, 2
2, 1.5
3, 1.33333
4, 1.25
5, 1.2
6, 1.16667
7, 1.14286
8, 1.125
9, 1.11111
```

We can now understand that `d_new` is filled differently because of the analytical expression. This can easily be fixed by doing $\frac{i+2}{i+1}$ instead of $\frac{i+1}{i}$. However, we found this error beneficial and decided to stick with it. After all, it turns out to be an improvement to the accuracy.

When it comes to the differences in calculation-time between the general and special algorithm we use the output from `timing.py` with the input `timings1000000.txt`. The program gives us the following output:

```
The average time for the general Thomas-algorithm after 10 runs is: 0.019301s
The average time for the special Thomas-algorithm after 10 runs is: 0.014612s
The specialized algorithm is 0.004689s faster than the general algorithm
```

This shows us that the specialized algorithm indeed is faster. This mainly comes down to the reduction of FLOPS.

Ignoring the initial-condition we chose to calculate mid-algorithm and looking at one iteration we have 4 FLOPS. This is half the FLOPS of the general algorithm.

4.4 Exercise d)

The error is calculated at line 162 in `thomas-algorithm.cpp`. It calculates the relative error made by the general algorithm. The raw data is put in `error.txt` and thereafter read by `error.py`. The program shows a plot (not a table) and is shown in figure (4). We can clearly see that the error decreases for larger n. From $n = 10^5$ to $n = 10^6$ we actually see that the program rapidly gains accuracy. This seems peculiar, but after mulitple runs for calculating errors and adjusting the codes we still get this plot. This might be since we have defined $h = \frac{x_{n-1} - x_0}{n-1}$ giving us an ugly value ($n = 10 \rightarrow h = 0.11\ldots$). When we reach $n = 10^6$ the machine probably rounds of h and leads to an easier calculation in the rest of the algorithm.

When it comes to the peak of accuracy we can see that $n = 10^6$ is most accurate as the error start increasing at $n = 10^7$. The increase of error comes from the machine's numerical error.

4.5 Exercise e)

Again, in figures (1), (2) and (3) we see the LU-decomposition compared to the other algorithms and to the exact solution. The LU-decomposition is identical to the general Thomas-algorithm except for our predefined endpoints.

Comparing the execution time of our tridiagonal solver (special algorithm) and the LU-decomposition we get the following times.

Table 1: Execution time for the two methods.

n	Tridiagonal	LU-Decomposition
10	$2.00 \cdot 10^{-7}$	$3.17 \cdot 10^{-4}$
100	$1,40 \cdot 10^{-6}$	$1.40 \cdot 10^{-3}$
1000	$1.44 \cdot 10^{-5}$	$3.36 \cdot 10^{-2}$

We can see from the table above that our specialized algorithm is vastly superior to the LU-decomposition. This is no surprise considering our algorithm has $\sim 4n$ FLOPS whilst the LU-decomposition has $\sim \frac{2}{3}n^3$. In addition the LU-decomposition has way more memory-reads and memory-writes and this slows the method down a lot.

When we run the LU-decomposition for a matrix with $n = 10^5$ the program crashes and states that it is "out of memory" (using **Qt**). This happens due to the fact that the program makes the computer generate a matrix $\mathbf{A}$ that has $10^5 \cdot 10^5$ number of elements. Each element is 8 bytes so we get a total of $8 \cdot 10^{10}$ bytes. This is the equivalent to 80 gigabytes. This is way to much for any standard computer to handle. And even if you get past the first matrix you will still have to generate 3 more ($\mathbf{L}$,$\mathbf{U}$ and sometimes $\mathbf{P}$) to complete the LU-decomposition.

5 Conclusion and perspective

The solution for equation (4), $\vec{v}$, has a good approximation to the exact solution of equation (3) both for the general algorithm and the special algorithm. This is proven by that the relative error described in equation (9) is small. It is also possible to consider the plot made in `plot_data.py` which shows that for different values of n, more specifically $n = 10, 100, 1000$, the graphs are approximately the same. See figures (1), (2) and (3).

If in future projects there is need for a solver of a second order ordinary differential equation, then the findings in this project can be useful, as the error is relatively small and the program fairy simple. At least for a special case as described in exercise **c)**, 3.3. One con of the methods in this project is that for larger values of n, the time and the memory the programs uses to run is too large. This is for the general and special algorithm with $n = 10^8$ and greater values of n. For the LU-decomposition n is limited at $n = 10^5$ and greater. Matrices with these dimensions is not recommended to run the programs.

6 Appendix

Below are the plots for the different exercises. Also see the GitHub-repository for the `.txt`-files `data10.txt`, `data100.txt`, `data1000.txt` and `error.txt`.

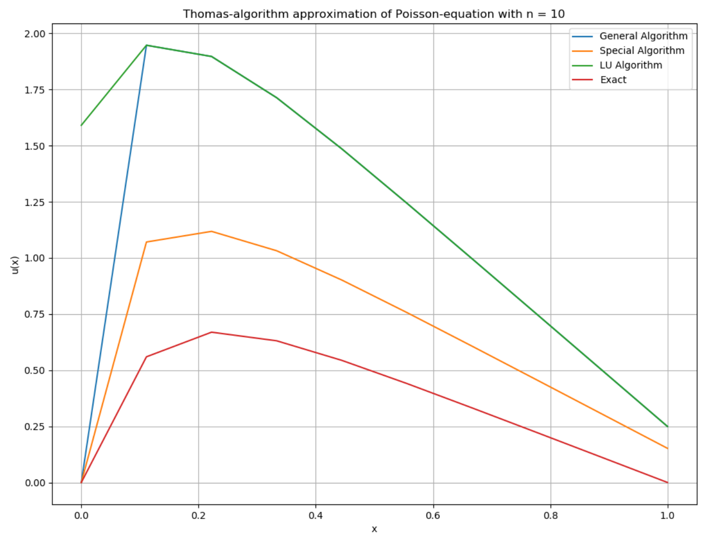

Figure 1: The plot of the different algorithms for $n = 10$.

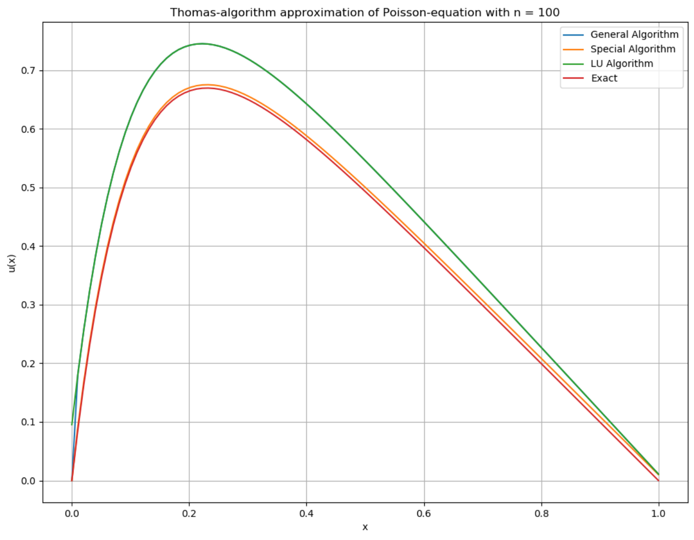

Figure 2: The plot of the different algorithms for $n = 100$.

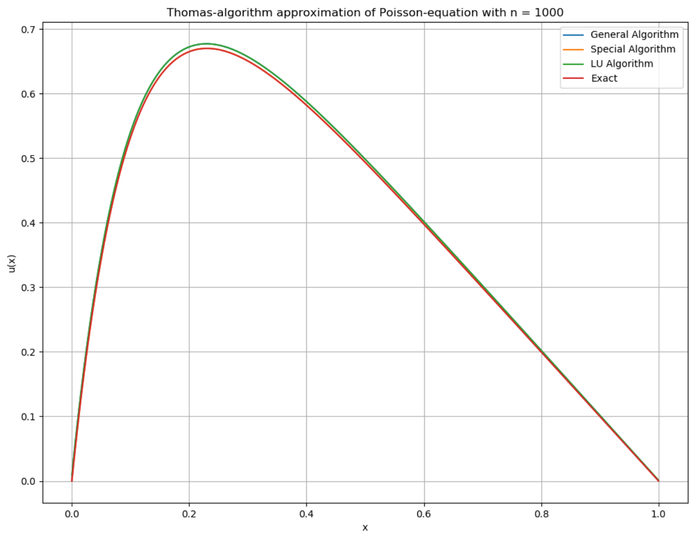

Figure 3: The plot of the different algorithms for $n = 1000$.

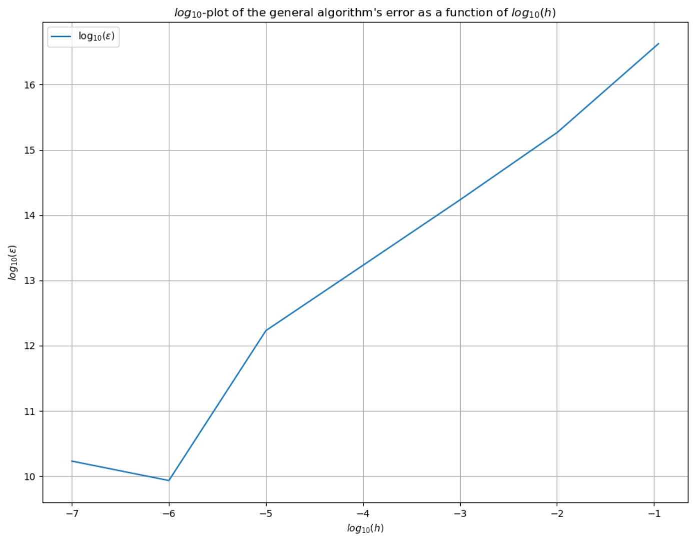

Figure 4: The plot of the error from $n = 10, 100, 1000..., 10^7$.

7 References

Project1.pdf by Morten Hjorth-Jensen. The PDF to the project we were given.

Our GitHub-repository.

Link to an article about the tridiagonal matrix algorithm. This includes general theory about the algorithm and how it works.

Link to lecture slides in FYS3150 - Computational Physics. See page 168 and the rest of chapter **6.4 Linear Systems** for theory behind the tridiagonal matrix algorithm.