FYS4150: Computational Physics

Project 1: Linear 2nd order differential equations

Dina Stabell
Kristine Garvin
Silje Christine Iversen

September 10, 2018

Abstract

A one-dimensional Poisson-equation with Dirichlet boundary conditions, approximated through Taylor expansion and solved numerically as a set of linear equations. The two methods used to solve the set of equations are Gaussian elimination and LU-decomposition. An algorithm for the general and a special case of the Gaussian elimination is developed and implemented. We find that the relative error of the special case of the Gaussian elimination is the least when the step-size used in the algorithm is around $n = 10^5$. When comparing the CPU time for the different algorithms used, we find that the algorithm for the LU-decomposition is the fastest.

Contents

1 Introduction

The aim of this project is to get familiar with various vector and matrix operations, from dynamic memory allocation to the usage of programs in the library package of the course. This will be done through examining Poisson's equation, which is a classical equation from electromagnetism. The focus in this project will be on the general one-dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations. These will then be solved by two different numerical methods, which then are compared with respect to CPU time.

To derive a finite difference approximation, we used Taylor expansion to approximate the derivative, and then defining a new variable of the discrete problem only defined on the grid points. We then used Gaussian elimination and LU-decomposition to solve the linear equations. When dealing with the Gaussian elimination we had to minimize the amount of floating point operations in the algorithms to minimize the CPU time. We also compared the number of floating point operations in the general algorithm to the algorithm in the specific case given.

In the first part of this report we derive a finite difference scheem to approximate the second derivative. We have derived the algorithms for the different methods used, and show how they can be implemented. Then we compare the different methods to estimate which should(theoretically) be the most accurate and efficient method. In the second part we present and discuss the findings. Finally, in the last part, we include some concluding remarks.

2 Method

The general one-dimensional Poisson equation with Dirichlet type boundary conditions is

$$-u_{xx} = f(x) \quad \text{for} \quad x \in (0,1) \tag{1}$$

$$u(0) = u(1) = 0$$

2.1 Defining a finite difference approximation

When defining a finite difference approximation to this equation, we use Taylor expansion to approximate the derivative. Assuming u to be four-times continuously differentiable, the result is

$$u(x+h) = u(x) + hu'(x) + \frac{h^2}{2}u''(x) + \frac{h^3}{6}u^{(3)}(x) + \frac{h^4}{24}u^{(4)}(x+h_1)$$

$$u(x-h) = u(x) - hu'(x) + \frac{h^2}{2}u''(x) - \frac{h^3}{6}u^{(3)}(x) + \frac{h^4}{24}u^{(4)}(x-h_2).$$

(Tveito & Winther, 2009, p.46).

Adding these two equations and reorganizing, gives

$$u(x+h) + u(x-h) = 2u(x) + h^2u''(x) + \mathcal{O}(h^2)$$

$$u'' = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + \mathcal{O}(h^2).$$

The next step is to partition the unit interval into a finite numbers of sub-intervals with the given grid points $x_i = ih$ where $h = 1/(n+1)$. Now defining v to be the solution of the discrete problem, defined only at the grid points, the finite difference approximation is given by

$$-\frac{v_{i-1} - 2v_i + v_{i+1}}{h^2} = f(x_i) \quad \text{for} \quad i = 1, 2, ..., n, \tag{2}$$

with the boundary conditions

$$v_0 = v_{n+1} = 0.$$

(Tveito & Winter, 2009, p.47).

When solving the first terms of the finite difference scheme, using $d_i = h^2 f_i$, the result becomes (remembering that $v_0 = v_{n+1} = 0$):

$$\begin{aligned}
&i = 1: \quad -v_0 + 2v_1 - v_2 = d_1\\
&i = 2: \quad -v_1 + 2v_2 - v_3 = d_2\\
&i = 3: \quad -v_2 + 2v_3 - v_4 = d_3\\
&\vdots\\
&i = n: \quad -v_{n-1} + 2v_n - v_{n+1} = d_n
\end{aligned}$$

By grouping the unknowns in a vector $\mathbf{v}$, the equations can be written as a linear set of equations of the form $\mathbf{Av} = \mathbf{d}$, namely:

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_n \end{bmatrix}$$

The matrix A is a tridiagonal matrix, a special form of matrix where all elements are zero except those on and immediately above and below the leading diagonal.

2.2 Floating point operations

When solving problems numerically on computers one often deals with large numbers. One goal of this project is to find the most efficient solver of the differential equation with respect to both CPU time and memory allocation. Therefore the floating point operations (FLOPS) needed to calculate the algorithms of each solver are being evaluated. The number of FLOPS in an algorithm is important when it is to be implemented on a computer. Floating point operations is addition, subtraction, multiplication and division. The problems with these operations are that they will lead to round-off errors and they take longer time to calculate, so they will increase the CPU time.

2.3 Gaussian elimination - General solver

The Gaussian elimination will provide an algorithm to solve the equations. For this algorithm and the linear system, $\mathbf{Av} = \mathbf{d}$, to be equivalent it is important that the elements in the algorithm for the diagonal are nonzero. A way of ensuring this is to check whether the matrix A is diagonal dominant:

$$\mid b_1 \mid > \mid c_1 \mid \;\text{ and }\; \mid b_i \mid \geq \mid a_i \mid + \mid c_i \mid$$
$$for\ i = 2, 3, ..., n$$

Where b_i are diagonal elements and a_i, c_i are non-diagonal elements (Tveito & Winther, 2009, p.53).

For this project we look at a general tridiagonal matrix, where the elements along the diagonal and non-diagonal are not necessarily the same. We have $\mathbf{Av} = \mathbf{d}$, which we can write as

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_1 & b_2 & c_2 & \ddots & \vdots \\ 0 & a_2 & b_3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \cdots & 0 & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_n \end{bmatrix}$$

to make it easier to understand and follow the process of the elimination. The first step of this Gaussian elimination is to get rid of the elements $a_1, ..., a_{n-1}$ right below the diagonal in the matrix $\mathbf{A}$, which is a process called forward substitution. The result of this forward substitution is that the first unknown v_1 is eliminated from the rest of the $n-1$ equations, and then the second unknown v_2 is eliminated from the rest of the $n-2$ equations and so on until one reaches the last row where the only unknown left is v_n. The next step is to use backward substitution, which means that we find the values of the unknowns $\mathbf{v}$ by first calculating v_n, and then v_{n-1} by using the value we found for v_n, and so on (Tveito & Winther, 2009, p.51).

Let us study a 4×4 matrix to understand this elimination process better. We have the linear set of equations (where $\mathbf{v}$ is the only unknown):

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix}$$

The forward substitution is done by subtracting a multiple of one row from the one below to get rid of the elements below the leading diagonal. In this specific case the multiple is a_1/b_1 to get rid of the element a_1. The linear set becomes

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & b_2 - \frac{a_1}{b_1}c_1 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 - \frac{a_1}{b_1}d_1 \\ d_3 \\ d_4 \end{bmatrix}$$

and the new elements are called $\tilde{b_i}$ and $\tilde{d_i}$ respectively, such that $\tilde{b_2} = b_2 - (a_1c_1)/b_1$ and $\tilde{d_2} = d_2 - (a_1d_1)/b_1$. To eliminate the next element a_2, a multiple of $a_2/\tilde{b_2}$ and the second row elements is subtracted from the third to get

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & b_3 - \frac{a_2}{\tilde{b_2}}c_2 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} d_1 \\ \tilde{d_2} \\ d_3 - \frac{a_2}{\tilde{b_2}}\tilde{d_2} \\ d_4 \end{bmatrix}$$

Finally the fourth row elements are subtracted by a multiple $a_3/\tilde{b_3}$ of the third, and

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & 0 & b_4 - \frac{a_3}{\tilde{b_3}}c_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} d_1 \\ \tilde{d_2} \\ \tilde{d_3} \\ d_4 - \frac{a_3}{\tilde{b_3}}\tilde{d_3} \end{bmatrix}$$

From this substitution we can write general algorithms for $\tilde{b}_i$ and $\tilde{d}_i$, and the algorithms become

$$\tilde{b}_i = b_i - \frac{a_{i-1}c_{i-1}}{\tilde{b}_{i-1}} \tag{3}$$

$$\tilde{d}_i = d_i - \frac{a_{i-1}\tilde{d}_{i-1}}{\tilde{b}_{i-1}} \tag{4}$$

To find a general expression for the unknown vector elements $\mathbf{v}$, the backward substitution comes into play. Starting with the fourth row, we get

$$\tilde{b}_4 v_4 = \tilde{d}_4 \qquad \Rightarrow v_4 = \tilde{d}_4/\tilde{b}_4 \tag{5}$$

$$\tilde{b}_3 v_3 + c_3 v_4 = \tilde{d}_3 \qquad \Rightarrow v_3 = \left(\tilde{d}_3 - c_3 v_4\right)/\tilde{b}_3 \tag{6}$$

$$\tilde{b}_2 v_2 + c_2 v_3 = \tilde{d}_2 \qquad \Rightarrow v_2 = \left(\tilde{d}_2 - c_2 v_3\right)/\tilde{b}_2 \tag{7}$$

$$b_1 v_1 + c_1 v_2 = d_1 \qquad \Rightarrow v_1 = \left(d_1 - c_1 v_2\right)/b_1, \tag{8}$$

and this indicates that v_i can be written as

$$v_i = \left(\tilde{d}_i - c_i v_{i+1}\right)/\tilde{b}_i. \tag{9}$$

Thus, the total algorithm becomes:

- Create all of the vectors needed to solve this numerical problem: x, v, a, b, c, d, $\tilde{b}$, $\tilde{d}$
- Insert the values for vectors we already know
- Insert the boundary conditions for the problem, that is $v_0 = v_{n+1} = 0$
- Insert $\tilde{d}_1 = d_1$ and $\tilde{b}_1 = b_1$
- Find $\tilde{d}_i$ and $\tilde{b}_i$ for $i = 2, ..., n$ by doing the forward substitution (using equation 3 and 4)
- Find v_i for $i = 1, ..., n$ by doing the backward substitution (using equation 9). Calculate v_n first, then v_{n-1} and so on

Floating point operations

In this case of the Gaussian elimination we have three algorithms, for $\tilde{b}_i$, $\tilde{d}_i$ and v_i. When counting the number of floating point operations (FLOPS) in these equations, one gets 3 FLOPS from 3, 3 FLOPS from 4 and 3 FLOPS from 9. However, one of the operations is computed twice, $a_i/\tilde{b}_i$, so by doing this in advance, one FLOP can be eliminated. This leads to a total of 8 FLOPS in the general Gaussian elimination. This is for every time the algorithms are run, so after n steps the number of FLOPS will have the size $8(n-1) = \mathcal{O}(8n)$.

Implementation

To be able to find the numerical solution to the problem, we implemented a Python script named *General.py*. This script includes the forward- and backward substitution, and the implementation of these substitutions is shown below. The vectors *d_tilde* and *b_tilde* represent the vectors $\tilde{d}$ and $\tilde{b}$, while the vectors *a_vec*, *b_vec*, *c_vec* and *d_vec* represents the vectors a, b, c and d. Obviously, v is the unknown vector v. In this script the user has to choose which n that should be used. This is done in the command line when calling the program. For $n = 10$, one should write **"$ python General.py 10"**.

```python
# FORWARD SUBSTITUTION
d_tilde[0] = d_vec[0]
b_tilde[0] = b_vec[0]
for i in range(1, n):
    b_tilde[i] = b_vec[i] - c_vec[i-1]*a_vec[i-1]/b_tilde[i-1]
    d_tilde[i] = d_vec[i] - d_tilde[i-1]*a_vec[i-1]/b_tilde[i-1]

# BACKWARD SUBSTITUTION
v[n] = d_tilde[i-1]/b_tilde[i-1]
for i in range(n-1, 0, -1):
    v[i] = (d_tilde[i-1] - c_vec[i-1]*v[i+1])/b_tilde[i-1]
```

2.4 Gaussian elimination - Specialized solver

For the special case with identical matrix elements along the diagonal, $b = 2$, and identical values for the elements right above/below the diagonal, $a = c = -1$, the algorithms already developed for the Gaussian elimination can be simplified as follows (by using the given numbers):

$$\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}} \tag{10}$$

$$\tilde{d}_i = d_i + \frac{\tilde{d}_{i-1}}{\tilde{b}_{i-1}} \tag{11}$$

$$v_i = \tilde{d}_{i-1} + \frac{v_{i+1}}{\tilde{b}_{i+1}} \tag{12}$$

Floating point operations

The special case of the Gaussian elimination simplifies the algorithms, and thereby the number of FLOPS is reduced. Now it is easy to see that the number of FLOPS is 2 for each of the three equations, adding up to a total of 6 FLOPS. Again this is done $n-1$ times, so that the total number of FLOPS adds up to $6(n-1) = \mathcal{O}(6n)$ FLOPS.

Implementation

The implementation of the Python script to find the numerical solution for the special case is found in the file named *Special.py*. The only change in this program compared to the program for the general case, is that we do not have to use the vectors for a, b and c. The part of the code which show the forward- and the backward substitution is shown below.

```python
# FORWARD SUBSTITUTION
d_tilde[0] = d_vec[0]
b_tilde[0] = 2.0
for i in range(1, n):
    b_tilde[i] = 2.0 - 1.0/b_tilde[i-1]
    d_tilde[i] = d_vec[i] + d_tilde[i-1]/b_tilde[i-1]

# BACKWARD SUBSTITUTION
for i in range(n, 0, -1):
    v[i] = (d_tilde[i-1] + v[i+1])/b_tilde[i-1]
```

2.5 LU - decomposition

LU-decomposition is a form of Gaussian elimination where you factorize a matrix into two matrices, **L** and **U**. The **L** matrix is a lower triangular matrix, while **U** is an upper triangular matrix (Hjorth-Jensen, 2015, p.173). This gives the matrix

$$\mathbf{A} = \mathbf{L}\mathbf{U} \tag{13}$$

where we have

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \ddots & \vdots \\ l_{31} & l_{32} & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & l_{nn-1} & 1 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \ddots & u_{3n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & u_{nn} \end{bmatrix}$$

For this project the LU factorization is done by a function, from the scipy.linalg library, called lu factor. The function returns a matrix containing U in its upper triangle, and L in its lower triangle (the unit diagonal elements of L are not stored).

For a matrix to have a LU-factorization, the determinant has to be nonzero. The algorithm for the determinant can be written as:

$$det\ [\mathbf{A}] = det\ [\mathbf{LU}] = det\ \mathbf{L} \cdot det\ \mathbf{U} = u_{11}u_{22}...u_{nn} \tag{14}$$

The LU-factorization is unique if this criteria is met and if **A** is non-singular, which we assume.

(Hjorth-Jensen, 2015, p.174)

The system of equations $\mathbf{Av} = (\mathbf{LU})\mathbf{v} = \mathbf{d}$ can be solved in two steps, where **L**, **U** and **d** are known:

$$\mathbf{Uv} = \mathbf{y} \tag{15}$$
$$\mathbf{Ly} = \mathbf{d} \tag{16}$$

The first step is to solve the system of linear equations from equation 16:

$$\begin{aligned} y_1 &= d_1 && (17)\\ l_{21}y_1 + y_2 &= d_2 && (18)\\ l_{31}y_1 + l_{32}y_2 + y_3 &= d_3 && (19)\\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + \cdots + y_n &= d_n && (20) \end{aligned}$$

In the first equation there is only one unknown, y_1, so the solution is trivial. This solution is then used to calculate the unknown in the second equation, y_2, and so on. The next step is to solve the system from 15:

$$\begin{aligned} u_{11}v_1 + u_{12}v_2 + \cdots + v_4 &= y_1 && (21)\\ u_{22}v_2 + u_{23}v_3 + \cdots + u_{2n}v_n &= y_2 && (22)\\ u_{33}v_3 + u_{34}v_4 + \cdots + u_{3n}v_n &= y_2 && (23)\\ &\vdots \\ u_{nn}v_n &= y_n && (24) \end{aligned}$$

The method to solve this set of equations is similar to the method to solve the equations from 16. In the last equation there is only one unknown, v_n, so the solution of this is trivial. This solution is used to solve for the unknown in the second to last equation, v_{n-1}, and so on.

Finally, this systems of equations gives a set for equations for every column of A.

$$\begin{aligned} j=1: \quad & a_{11} = l_{11}u_{11}\\ & a_{21} = l_{21}u_{11}\\ & a_{31} = l_{31}u_{11}\\ & \vdots \\ & a_{n1} = l_{n1}u_{11}\\ j=2: \quad & a_{12} = u_{12}\\ & a_{22} = l_{21}u_{12} + u_{22}\\ & a_{32} = l_{31}u_{12} + l_{32}u_{22}\\ & \vdots \\ & a_{n2} = l_{n1}u_{12} + l_{n2}u_{22}\\ & \vdots \\ j=n: \quad & a_{nn} = l_{n1}u_{1n} + l_{n2}u_{2n} + \cdots + l_{nn}u_{nn} \end{aligned}$$

When going from the first to the second column we do not need any further information from the matrix elements a_{i1}. This is a general property throughout the whole algorithm. Thus the memory locations for the matrix $\mathbf{A}$ can be used to store the calculated matrix elements of $\mathbf{L}$ and $\mathbf{U}$. (Hjorth-Jensen, 2015, p.174). To save memory and gain performance in our program, this has been implemented through the boolean variable *overwrite* which is an option in the LU factor function that has been used to solve the equation.

The result leads to three algorithms:

$$u_{ij} = a_{ij} - \Sigma l_{ik} \cdot u_{kj} \tag{25}$$

$$u_{jj} = a_{jj} - \Sigma l_{jk} \cdot u_{kj} \tag{26}$$

$$l_{ij} = \frac{1}{u_{jj}}(a_{ij} - \Sigma l_{ik} u_{kj}) \tag{27}$$

Floating point operations

The number of FLOPS for solving by LU-decomposition is $\frac{2}{3}\mathcal{O}\left(n^3\right)$. This quadratic number makes the calculations quite slow when using lagrer values of n.

Implementation

The LU-decomposition way of solving the problem is implemented in the file *LUdecomp.py*. The piece of the code below shows the built-in ways of the calculation. The LU-decomposition is done in two steps with functions from the scipy.linalg library.
First we use **lu factor** which gives a factorization of our matrix A.
lu is a matrix containing U in its upper triangle, and L in its lower triangle. The unit diagonal elements of L are not stored.
piv is an array of pivot indices representing the permutation matrix P: row i of matrix was interchanged with row piv[i].
The **lu solve** function solves the equation system Av = d, given the LU factorization from **lu factor**.
The array **w** is the solution of the system, does not include the boundary conditions.
To increase performance:
overwrite a = True, lets the function reuse the input matrix A and vector d.
check finite = False, the function will not check that the input matrix contains only finite numbers.

```
# LU FACTORIZATION
lu, piv = sl.lu_factor(A, overwrite_a=True, check_finite=False)
# LU SOLVER
w = sl.lu_solve((lu, piv), d, trans=0, overwrite_b=True, check_finite=False)
v[1:-1] = w
```

3 Results

3.1 Analytic solution

The analytic solution of the equation we are solving (equation 1) is

$$\begin{aligned} -u_{xx} &= f(x) \\ -u_{xx} &= 100e^{-10x} \\ -u_x &= 10e^{-10x} + C \\ u &= -e^{-10x} + Cx + D \end{aligned}$$

Now, inserting the boundary conditions will provide the values of C and D, and we get

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

The solution becomes

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x}, \tag{28}$$

which is the solution given in the task. This solution is implemented in the Python scripts which solves the numerical problem. The reason for this that we want to be able to compare the numerical solutions to the analytic one.

3.2 Numerial solution

By running the program *General.py* for a given number of n, the code produces plots of the numerical solution. For $n = 10$, which produces a solution where the vector $\mathbf{v}$ goes from v_0 to $v_{n+1} = v_{11}$, we get the plot showed in Figure 1. We can see that by using such a small number for n, or such a large number of the spacing $h = 1/(n+1) = 1/11$, both the analytic- and the numerical solution in the plot become far from smooth. We can also see that the numerical solution do not match with the analytic solution. The largest positive values of the numerical solution are too small. All in all, this numerical solution is not good enough, and we need a smaller spacing between the calculated numbers.

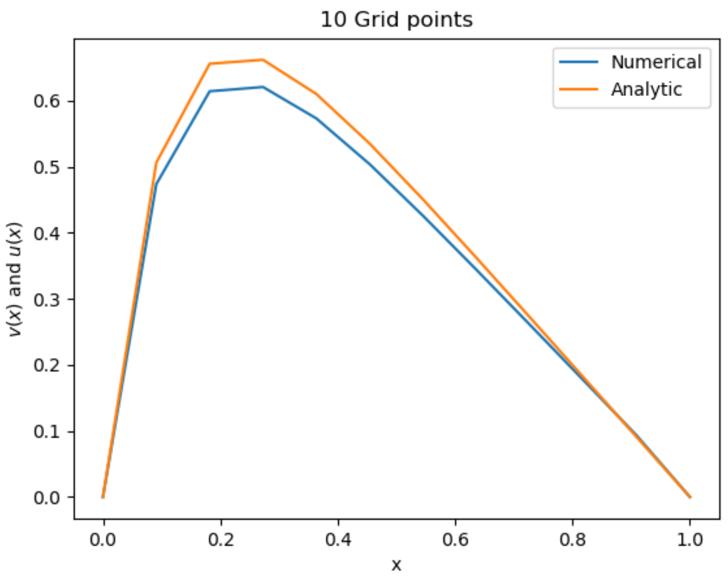

Figure 1: Numerical $v(x_i)$ vs. analytic $u(x_i)$ for $i = 1, ..., n+1$

By running the same program for $n = 100$, the plot in Figure 2 is produced. Now both of the solutions become more smooth, and the numerical solution gets closer to the analytic one. But we do still not have a perfect match between the two solutions. This can be seen in Figure 3 where we have zoomed in at the top point of the plot.

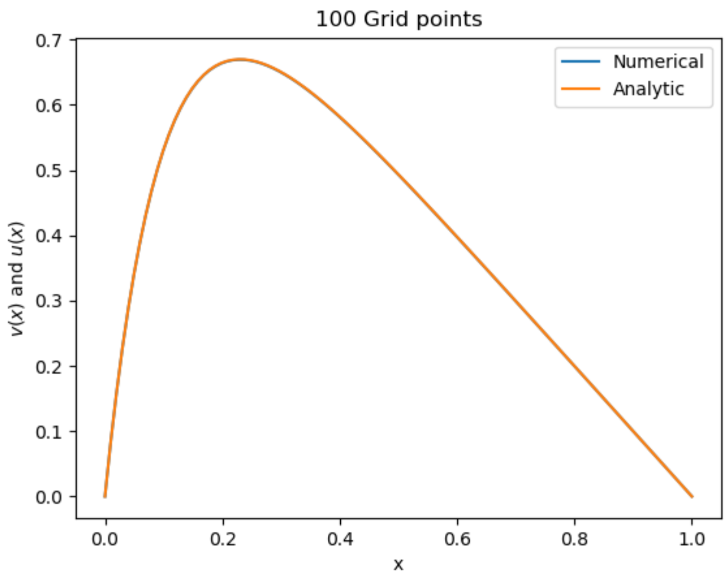

Figure 2: Numerical $v(x_i)$ vs. analytic $u(x_i)$ for $i = 1, ..., n + 1$

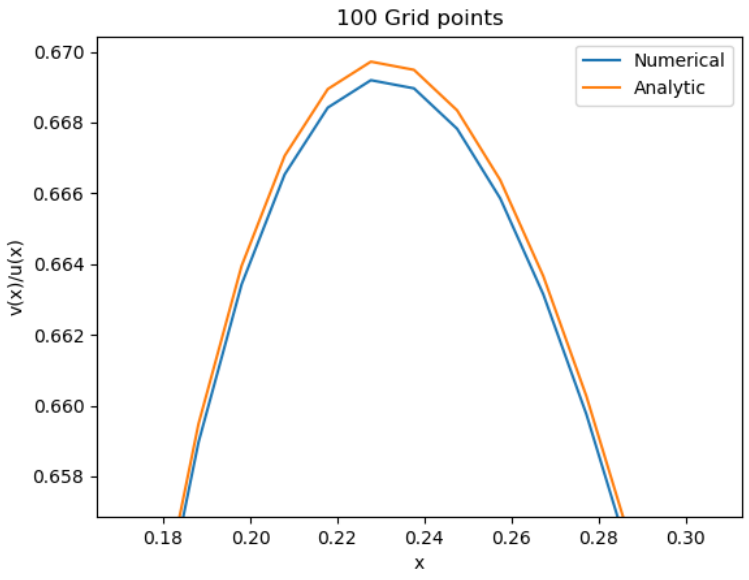

Figure 3: Zoomed in version of Figure 2

Increasing the size of n once more such that $n = 1000$ produces the plot in Figure 4. From this plot we find that the numerical solution is even closer to the analytic one, since zooming in at the top point shows that $u(x_i) - v(x_i)$ is way smaller than for $n = 100$ (see Figure 5).

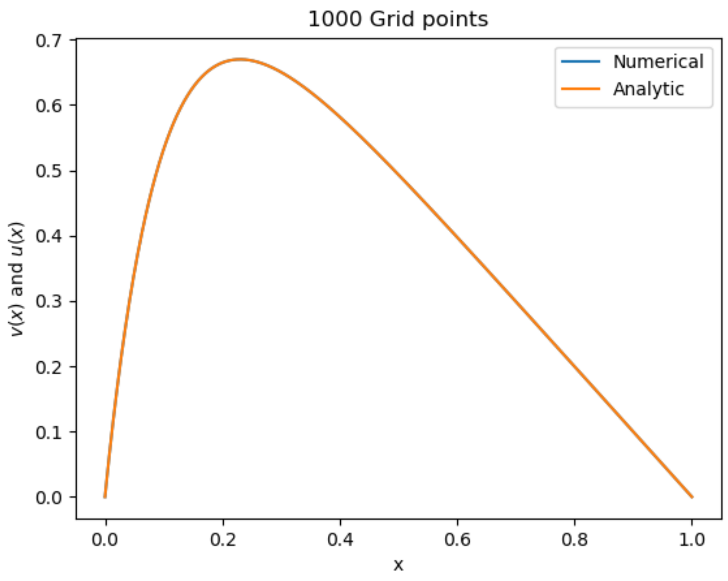

Figure 4: Numerical $v(x_i)$ vs. analytic $u(x_i)$ for $i = 1, ..., n + 1$

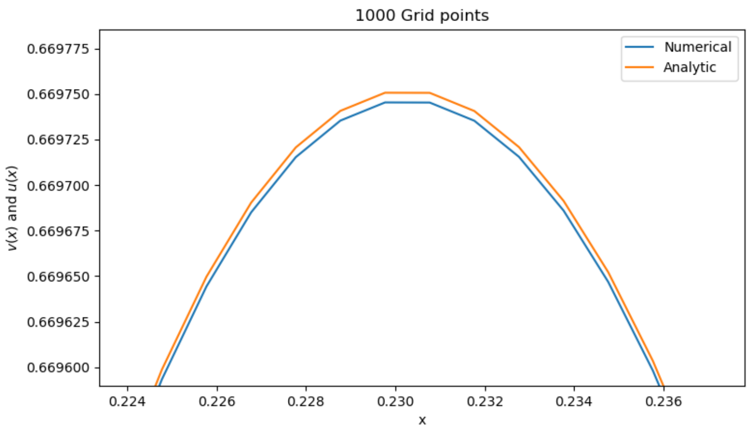

Figure 5: Zoomed in version of Figure 5

3.3 Floating point operations

The different algorithms described, has different number of floating point operations, the general Gaussian elimination algorithm has $8(n-1) = \mathcal{O}(8n)$ FLOPS, the special case of the Gaussian elimination has $6(n-1) = \mathcal{O}(6n)$ FLOPS, and the LU-decomposition has $\mathcal{O}(\frac{2}{3}n^3)$FLOPS.

According to this it is expected that the implementation of the special case of the Gaussian elimination is the fastest algorithm with the least round-off error.

3.4 Relative error

To calculate the relative error in the data set $i = 1, ..., n$, we use the formula

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

This error is supposed to be computed as a function of $log_{10}(h)$, and for each step length, $h = 1/(n+1)$, we extract the max value of the relative error. This is done in the program *error.py*, where we use the values $n = 10$ until $n = 10^7$. A piece of the code is shown below. Here we have to slice the vectors since the vectors include index numer $i = 0$ and $i = n+1$ - which are indexes we should not include.

```
# Compute the relative error:
v_sliced = v[1:-1]
u = u(x)
u_sliced = u[1:-1]

error_max = np.max(np.abs((v_sliced - u_sliced)/u_sliced))
```

Table 1 shows the maximum relative error for different values of n. Remember that the corresponding step size is calculated as $h = 1/(n+1)$, so the step size decreases with increasing n. We have also plotted the relative error as a function of $log_{10}(h)$, and the result is shown in Figure 6.

n	**Relative error**
10	6.61×10^{-2}
10^2	8.17×10^{-4}
10^3	8.31×10^{-6}
10^4	8.33×10^{-8}
10^5	1.43×10^{-9}
10^6	8.40×10^{-7}
10^7	2.98×10^{-6}

Table 1: Maximum relative error ϵ for different values of n.

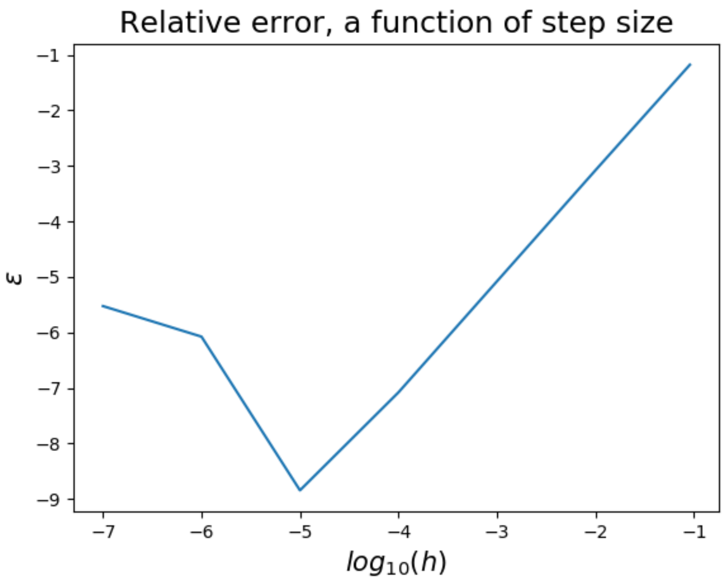

Figure 6: Maximum relative error ϵ for different values of h.

We can see that we have the smallest relative error when $n = 10^5$. If we increase the value of n, or in other words decrease the step size further than $h = 1/(10^5+1)$, the error will start growing. This can be explained by the limitations when representing numbers on computers, which can cause round off errors for sufficiently large and small numbers.

3.5 CPU time

In this project we have found a numerical solution to a problem in three different ways. In each of the different ways, we have checked the CPU time for running through the algorithm. Since the CPU time changes each time we run the program, we ran the programs ten times. We then found the mean of those ten results.

In the first program *General.py* where we used the Gaussian elimination with a general solver and in the second program *Special.py* where we used the Gaussian elimination with a specialized solver, we got the results shown in table 2. We can clearly see that the special solver with reduced number of FLOPS (compared to the general solver) is the most fast algorithm.

n	**General**	**Special**
10^6	3.69 s	2.71 s

Table 2: CPU-time

We then tried to compare the program *LUdecomp.py* where we use LU-decomposition with the program for the Gaussian elimination with a specialized solver - since this specialized solver was the program with the lowest CPU-time in the previous test. Trying first for $n = 10^5$ gave us a result which we did not want - we got memory error. This is probably because the number of floating points operations one needs to be able to use LU-decomposition are way higher than for the other methods. Decreasing n to $n = 10^4$ gave us the results in Table 3. Here we see that the LU-solver is faster than the specialized solver - but on the other hand the LU-decomposition raises a major problem since we can not use it for $n = 10^5$, which is the number where the relative error is smallest.

n	**Special**	**LU-solver**
10^4	2.4×10^{-2} s	1.22×10^{-1} s

Table 3: CPU-time

4 Conclusion

When working with numerical problems, one might think that the larger n, the better. Larger n means smaller spacing between the numbers, and probably a more precise and smooth solution. This is not the case. We have seen that this is the case up til $n = 10^5$, and that if we choose larger values for n, the relative error will grow and results become wrong. If we want to use $n = 10^5$, then we can not use LU-decomposition since this way of solving the equations requires a lot of FLOPS, and we will get memory error. LU-decomposition seemed to be the one with the lowest CPU-time though, so we have to figure out what is most important: a fast CPU-time or a minimal relative error.

All in all, when solving differential equations on a computer, one has to think carefully about which method to use. Each method has its own number of FLOPS, and some are faster than the others.

5 Appendices

Codes are found through this link:
https://github.com/krisgarv/FYS4150/tree/master/Problemsets/1st/CODES

Plots are found through this link:
https://github.com/krisgarv/FYS4150/tree/master/Problemsets/1st/PLOTS

References

[1] Tveito, A. & Winther, R.(2009). *Introduction to Partial Differential Equations - A Computational Approach.* [Berlin]:Springer-Verlag Berlin Heidelberg.

[2] Hjorth-Jensen, M. (2015). *Computational Physics - Lecture notes Fall 2015.* University of Oslo.