# An Analysis of Numerical Methods for Solving Second Order Linear Differential Equations

Anders Bråte, Maria Linea Horgen & Kaspara Skovli Gåsvær

(Dated: September 9, 2019)

Solving linear differential equations numerically is vital in many aspects of modern physics and modern computing in general. Hence, being able to effectively and with certainty make good approximations is key. In this paper we study three algorithms for solving a general linear second order linear differential equation: A general and special case of the Thomas algorithm [1] and a LU decomposition algorithm. We also look at the loss of numerical precision for one of the methods, as well as the CPU time for all three. Our findings generally shows that the special case of the Thomas algorithm is the most preferable, with an 35% less average CPU time than the general Thomas algorithm. The LU decomposition algorithm provides a wider area of use, but is limited by the size of the matrix, due to inefficient computer storage.

## I. INTRODUCTION

In this project our initial goal was simple; to solve a one-dimensional second order differential equation that reads

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

This form of equation appears in many situations, where a numerical application often comes natural. One example of this is the famous Poisson's equation, which describes the electric potential given a distribution of charges in space

$$\frac{d^2\phi}{dr^2} = -4\pi r\rho(r). \tag{2}$$

This equation very much looks like our general differential equation, and is easily applicable in simple cases. However when more complex systems arise the numerical option is very much preferable. These numerical calculations however bring power, as well as responsibility in the form of numerical error. Truncation and rounding error haunts every calculation made, and being aware of, and being able to quantify the error and loss of numerical precision is paramount when using numerical calculations in differential equations. We will look at three methods of solving the equation, and the relative errors as compared to the analytical solution. We will also observe the impact of varying amounts of floating point operations in our algorithms, and how it affects calculation time.

Initially we will derive the first two algorithms for solving the equations, one general Thomas algorithm, as well as a special case algorithm which optimises the number of calculations needed. We will in addition compare this with an LU decomposition algorithm using the c++ library armadillo, in order to compare computation times and errors.

### A. Reproducibility

All code is found on Github [2], they follow the same pattern. One c++ file and one python file to initiate for each task. Every python file requires one command line argument, the multiplicity of $N$. That means, if you want to run the code for all $N$ up to $10^5$, you have to type the below command into the terminal window:

```
user$ python3 mainX.py 5
```

## II. METHOD

### A. From differential equations to a linear system

We wanted to solve equation (1) numerically by writing it as a set of linear equations using the Dirichlet boundary conditions. The differential equation has a closed-form solution given by $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$, and the source term is $g(x) = 100e^{-10x}$. The first step is to take a look at the mathematical definition of the second order derivative of a function $u(x)$:

$$u''(x) = \lim_{h \to 0} \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}, \tag{3}$$

where $h$ is an infinitesimal change. When implementing to code, one has to remove the limit and thereby use small but not infinitesimal values of $h$, which leads to approximation errors. This is further explained under *Error analysis*.

To solve equation (3) numerically we needed to discretize it by rewriting $x$ and $u(x)$. In our problem $x \in [0, 1]$, so we wrote $x$ and $u(x)$ as vectors.

$$x = x_i = x_0 + ih, \qquad u(x) = u(x_i) = u_i.$$

$$\hat{x}^T = [x_1, ..., x_n] \qquad \hat{u}^T = [u_1, ..., u_n],$$

The indices of the vectors run from 1 to $n$, where $n$ is the number of time steps. From the Dirichlet boundary conditions we have $u(0) = u(1) = 0$, which gives us

$$u_0 = u(x_0) = u(0) = 0, \qquad u_{n+1} = u(x_{n+1}) = u(1) = 0.$$

The change $h$ is defined as the step length and is given by

$$h = \frac{x_n - x_0}{n} = \frac{1}{n}.$$

This means that the approximation to the second derivative of $u(x)$ can be written as

$$\frac{d^2u}{dx^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = g(x_i) = g_i, \tag{4}$$

where $g_i$ is the solution in each time step, and can be rewritten as

$$u_{i+1} - 2u_i + u_{i-1} = G_i \qquad \text{where} \quad G_i = h^2 g_i.$$

The solution for each time step can then be written as a vector, in the same way as $x$ and $u(x)$.

$$\hat{G_i}^T = h^2 \cdot [g_1, ..., g_n].$$

By implementing $i = 1, 2, ..., n$ in equation 4, a pattern emerges regarding the indices:

$$\begin{aligned}
i = 1: &\qquad u_2 + u_0 - 2u_1 = G_1, \\
i = 2: &\qquad u_3 + u_1 - 2u_2 = G_2, \\
\vdots &\qquad \vdots \\
i = n: &\qquad u_{n+1} + u_{n-1} - 2u_n = G_n,
\end{aligned}$$

and the system of $n$ equations can be written as the linear system

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \ddots & \ddots & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ \vdots \\ G_n \end{bmatrix}.$$

The matrix $\mathbf{A}$ is a tridiagonal Töplitz matrix, $\hat{T}$, and in simple terms the system above can be written as

$$\hat{T} \cdot \hat{u} = \hat{G}, \tag{5}$$

which we can solve by Gaussian elimination.

### B. The Thomas algorithm

The linear system in equation 5 can be solved numerically by the Thomas algorithm. This algorithm is a special case of Gauss elimination, and consists of three steps:

1. Decomposing the matrix into vectors.
2. A forward substitution.
3. A backward substitution.

*Step 1: Decomposition*

The Thomas algorithm is a general algorithm for solving linear equations, and the matrix does not need to consist of identical non-diagonal elements and identical diagonal elements. Hence, we can decompose a general tridiagonal matrix to three one-dimensional vectors of length $n$.

$$\hat{T} = \begin{bmatrix} d_1 & a_1 & 0 & \dots & \dots & 0 \\ b_1 & d_2 & a_2 & 0 & \dots & \vdots \\ 0 & b_2 & d_3 & a_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 & \vdots \\ \vdots & \dots & \ddots & b_{n-2} & d_{n-1} & a_{n-1} \\ 0 & \dots & \dots & 0 & b_{n-1} & d_n \end{bmatrix} \rightarrow d = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_n \end{bmatrix}, a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ \vdots \\ a_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}$$

*Step 2: Forward substitution*

By using Gauss elimination on system 5 the first two equations we acquire are

$$d_1 u_1 + a_1 u_2 + 0 + \cdots + 0 = G_1, \tag{6}$$

$$b_1 u_1 + d_2 u_2 + a_2 u_3 + 0 + \cdots + 0 = G_2. \tag{7}$$

Multiplying equation 6 with $b_1/d_1$, and then subtracting equation 6 from 7 we find a new expression for the next diagonal- and source term element:

$$u_2 \underbrace{(d_2 - \frac{a_1 b_1}{d_1})}_{\tilde{d}_2} + a_2 u_3 = \underbrace{G_2 - G_1 \frac{b_1}{d_1}}_{\tilde{G}_2}. \tag{8}$$

Rewriting equation 8 with $\tilde{d}_2$ and $\tilde{G}_2$, and repeating the process with the third acquired equation from system 5 a pattern emerges. The variables $\tilde{d}$ and $\tilde{G}$ can be expressed as

$$\tilde{d}_i = d_i - \frac{a_{i-1}b_{i-1}}{\tilde{d}_{i-1}}, \qquad \tilde{G}_i = G_i - \tilde{G}_{i-1}\frac{b_{i-1}}{\tilde{d}_{i-1}}, \tag{9}$$

with $\tilde{d}_1 = d_1$ and $\tilde{G}_1 = G_1$.

*Step 3: Backwards substitution*

The last step in the algorithm is the backwards substitution. After step 2 our problem now looks like

$$\hat{T} = \begin{bmatrix} \tilde{d}_1 & a_1 & 0 & \dots & \dots & \dots \\ 0 & \tilde{d}_2 & a_2 & \dots & \dots & \dots \\ & 0 & \tilde{d}_3 & a_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & 0 & \tilde{d}_{n-1} & a_{n-1} \\ & & & & 0 & \tilde{d}_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} \tilde{G}_1 \\ \tilde{G}_2 \\ \vdots \\ \vdots \\ \tilde{G}_n \end{bmatrix}.$$

We want to solve the system for $u$. By working our way backwards we find the general expression for $u_i$ and $u_n$.

$$\tilde{d}_i u_i + a_i u_{i+1} = \tilde{G}_i \quad \rightarrow \quad u_i = \frac{\tilde{G}_i - a_i u_{i+1}}{\tilde{d}_i}, \tag{10}$$

$$\tilde{d}_n u_n = \tilde{G}_n \quad \rightarrow \quad u_n = \frac{\tilde{G}_n}{\tilde{d}_n}. \tag{11}$$

The implementation of the Thomas algorithm is discussed later in the report.

### C. Special case of the Thomas algorithm

We can alter the Thomas algorithm for our special matrix $\mathbf{A}$, where $d_i = 2$ and $a_i = b_i = -1$. By inserting these values in the previous equations (eq. 9, 10 and 11) we obtain two simplified equations. If we look at a few time steps for $\tilde{d}$, while using that $\tilde{d}_1 = d_1 = 2$, it becomes clear that an analytical expression for $\tilde{d}_i$ can be written as

$$\begin{aligned}\tilde{d}_2 &= 2 - \frac{1}{2} = \frac{3}{2} \\ \tilde{d}_3 &= 2 - \frac{1}{3/2} = \frac{4}{3} \\ &\vdots \\ \tilde{d}_i &= \frac{i+1}{i}.\end{aligned} \tag{12}$$

By using the last expression in the equations for the general Thomas algorithm we obtain an algorithm for our special case. The specialized equations are listed below:

$$\tilde{G}_i = G_i - \tilde{G}_{i-1}\frac{b_{i-1}}{\tilde{d}_{i-1}} \quad \rightarrow \quad \tilde{G}_i = G_i + \frac{(i-1)}{i}\tilde{G}_{i-1}, \tag{13}$$

$$u_i = \frac{(\tilde{G}_i - a_i \cdot u_{i+1})}{\tilde{d}_i} \quad \rightarrow \quad u_i = u_{i+1} + \frac{i}{i+1}\tilde{G}_i, \tag{14}$$

$$u_n = \frac{\tilde{G}_n}{\tilde{d}_n} \quad \rightarrow \quad u_n = \frac{n}{n+1}\tilde{G}_n. \tag{15}$$

### D. Error analysis

In order to get a handle on the error in our methods we wish to have a closer look at the numerical inaccuracies, and how we can predict these. We can derive the total error of our approximation in equation 3 by looking at the Taylor expansion for $f(x+h)$ and $f(x-h)$ to acquire the approximation to the second derivative:

$$\begin{aligned}f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(\xi), \qquad \xi \in [x, x+h] \\ f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(\zeta). \qquad \zeta \in [x-h, x]\end{aligned}$$

By adding these together we get

$$f''(x) - \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = -\frac{h}{6}(f'''(\xi) - f'''(\zeta)). \tag{16}$$

The function value $f(x)$ can normally not be exact represented in the computer, so the value we are actually working with can be written as $f(x)(1+\epsilon_a)$, where $\epsilon_a$ is the difference between the function value and the nearest exact floating point. We can therefore rewrite the left side of equation 16 to

$$
\begin{aligned}
f''(x) - \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} &= f''(x) - \frac{f(x+h)(1+\epsilon_1) - 2f(x)(1+\epsilon_2) + f(x-h)(1+\epsilon_3)}{h^2} \\
&= \underbrace{f''(x) - \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}}_{-\frac{h}{6}(f'''(\xi) - f'''(\zeta))} - \frac{f(x+h)\epsilon_1 - 2f(x)\epsilon_2 + f(x-h)\epsilon_3}{h^2} \\
&= -\frac{h}{6}(f'''(\xi) - f'''(\zeta)) - \frac{f(x+h)\epsilon_1 - 2f(x)\epsilon_2 + f(x-h)\epsilon_3}{h^2} \\
&= \left|\frac{h}{6}(f'''(\xi) - f'''(\zeta))\right| + \left|\frac{f(x+h)\epsilon_1 - 2f(x)\epsilon_2 + f(x-h)\epsilon_3}{h^2}\right|.
\end{aligned}
$$

We then use the triangular inequality on the last expression, which gives us

$$
\leq \left|\frac{h}{6}f'''(\xi)\right| + \left|\frac{h}{6}f'''(\zeta)\right| + \frac{1}{h^2}\Big(\Big|f(x+h)\epsilon_1\Big| + 2\Big|f(x)\Big| + \Big|f(x-h)\Big|\Big). \tag{17}
$$

We define a new variable $\sigma$ on the whole interval $(0,1)$, which is $max\{|f'''(\xi)|, |f'''(\zeta)|\}$ and notice that when $h$ is small $f(x \pm h) \approx f(x)$. If we then evaluate $f(x)$ in its maximum value on the interval $(0,1)$ we get an expression that is bigger or equal to eq. 17:

$$
\begin{aligned}
&\leq \frac{h}{6}\Bigg(\underbrace{\Big|f'''(\xi)\Big| + \Big|f'''(\zeta)\Big|}_{2|f'''(\sigma)|}\Bigg) + \frac{1}{h^2}\Big(\Big|f(x+h)\epsilon_1\Big| + 2\Big|f(x)\Big| + \Big|f(x-h)\Big|\Big) \\
&\leq \frac{h}{3}\max_{\sigma\in(0,1)}|f'''(\sigma)| + \frac{1}{h^2}\max_{\chi\in(0,1)}|f(\chi)(|\epsilon_1| + 2|\epsilon_2| + |\epsilon_3|).
\end{aligned}
$$

We assume that $\epsilon_1, \epsilon_2$ and $\epsilon_3$ are all less or equal to machine precision, so the final inequality reads

$$
\leq \frac{h}{3}\max_{\sigma\in(0,1)}|f'''(\sigma)| + \frac{4\epsilon_M}{h^2}\max_{\chi\in(0,1)}|f(\chi)| = \epsilon_{tot}. \tag{18}
$$

## III. IMPLEMENTATION

The implementation of the general case of the Thomas algorithm is based on equations 10, 9 and 11. The algorithm consists of step 2 and 3 described above. Handling matrices numerically is in general a bad idea. This has to do with computer memory and the number of floating point operations (flops). A flop corresponds to one mathematical operation. By using a dense matrix the number of flops required to solve system 5 goes as $2/3 \cdot N^3$. The Thomas algorithm requires $9N$

flops, and the special case requires only 4N flops [3]. That is why we are starting out with vectors, and thus step 1 is not included in the implementation.

Below follows the pseudo-code for implementing the general Thomas algorithm.

**Algorithm 1:** Forward substitution

**for** $1 \leq i \leq \mathrm{n}$ **do**

$\tilde{d}_i = d_i - (a_i \cdot b_i)/\tilde{d}_{i-1}$

$\tilde{G}_i = G_i - \tilde{G}_i \cdot (b_i/\tilde{d}_i)$

**end for**

The backward substitution is done in a similar manner, except the iterations are now backwards (given by the name). It is important to note that the indices of our mathematical equations, and hence our algorithms, not necessarily transfers to your preferred programming language.

**Algorithm 2:** Backward substitution

**for** $\mathrm{n} \geq i \geq 1$ **do**

**if** $i = n$ **then**

$u_n = \tilde{G}_n/\tilde{d}_n$

**else**

$u_i = (\tilde{G}_i - a_i \cdot u_{i+1})/\tilde{d}_i$

**end if**

**end for**

The implementation of the special Thomas algorithm is fairly similar, and hence is not included in the report.

## IV. RESULTS

Figure 1 shows the general Thomas algorithm applied on system 5 for time steps $N = 10, 100$ and 1000, compared with the closed-form solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$.

We compared the CPU time for the three different algorithms, with time steps up to $10^6$. The results are listed in table II.

A quantitative measurement for how prominent the total error is in our calculation is the relative error. For every time step, and hence step-length, we extracted the maximum error, and the result is presented logarithmically in figure 2 and in table II. From the expression in equation 18 we found

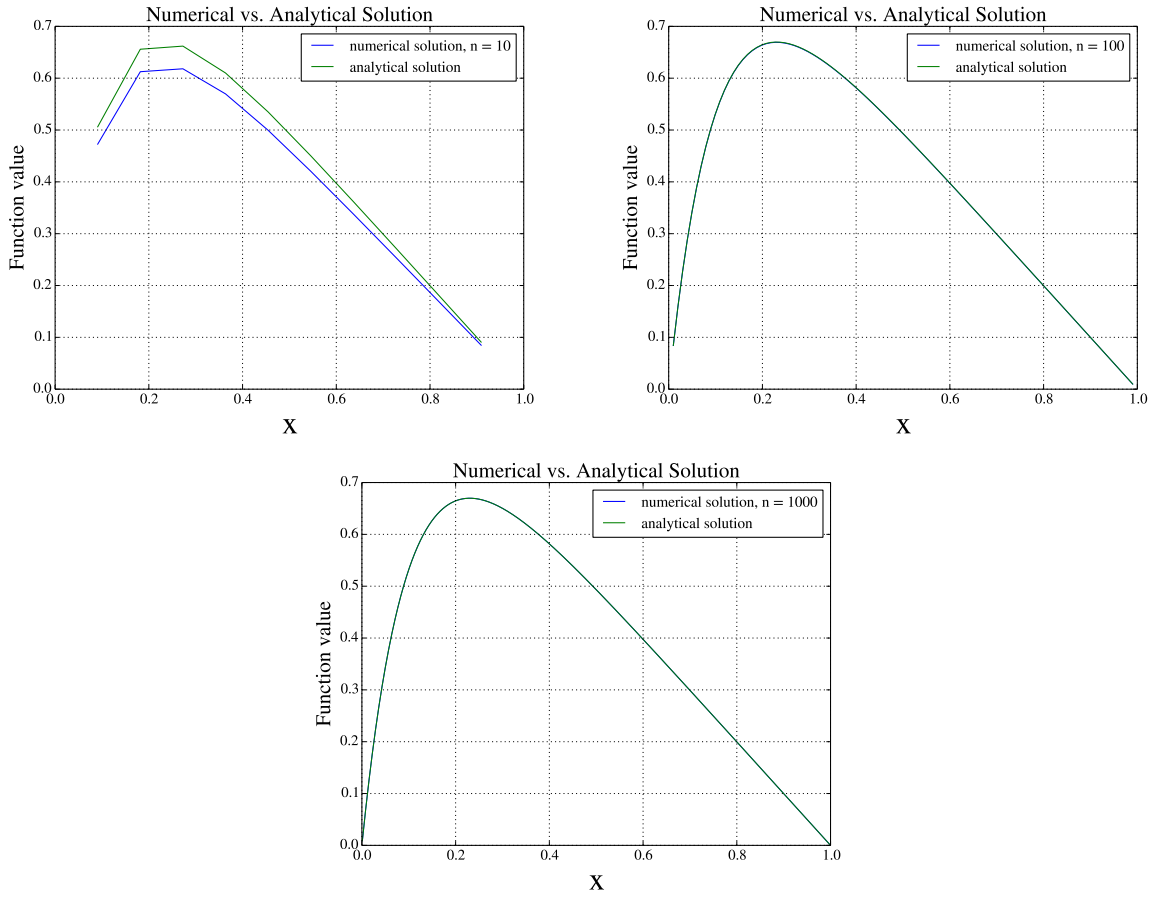


FIG. 1. The Thomas algorithm plotted againts the closed-form solution for three different time steps.

| Time Used [s] | | | |
|---|---|---|---|
| N | General Thomas | Special Thomas | LU |
| $10^1$ | 3.0e-06 | 3.0e-06 | 8.8e-05 |
| $10^2$ | 5.0e-06 | 4.0e-06 | 1.1e-03 |
| $10^3$ | 6.9e-05 | 1.9e-05 | 6.0e-02 |
| $10^4$ | 3.9e-04 | 2.0e-04 | 3.4e+01 |
| $10^5$ | 3.3e-03 | 1.9e-03 | No data |
| $10^6$ | 2.9e-02 | 2.0e-02 | No data |

TABLE I. The CPU time for the three algorithms solving system (5).

the analytical expression for the step-length which produces the smallest relative error:

$$\begin{aligned} \frac{d\epsilon_{tot}}{dh} &= 0 \\ \frac{1}{3}\max_{\sigma\in(0,1)}|f'''(\sigma)| - \frac{8\epsilon_M}{h^3}\max_{\chi\in(0,1)}|f(\chi)| &= 0 \\ \log(h) &= \underline{-5.59} \end{aligned} \tag{19}$$

We also plotted equation 18 in a logarithmic manner, and expected a global minimum for $\log(h) = -5.59$. This is the right side of figure 2.

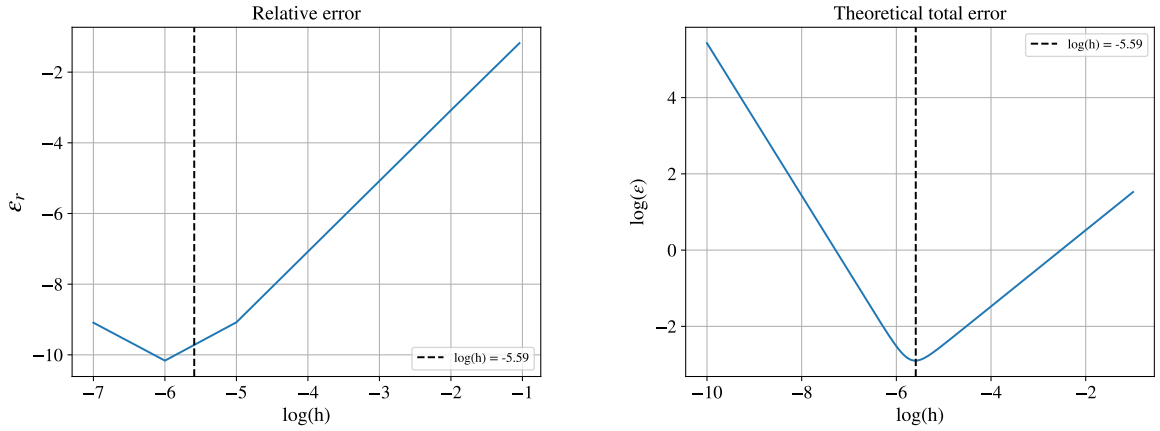


FIG. 2. Left: Maximum relative error, for the special case of the Thomas algorithm, for each step-length presented in a logarithmic plot. Vertical line being the analytical least log(h). Right: Theoretical total error, with the vertical line being the expected step-length which generate the least error.

| Relative Error | |
|---|---|
| N | Maximum ($\epsilon_r$) |
| $10^1$ | -1.18 |
| $10^2$ | -3.09 |
| $10^3$ | -5.08 |
| $10^4$ | -7.08 |
| $10^5$ | -9.08 |
| $10^6$ | -10.16 |
| $10^7$ | -9.09 |

TABLE II. The maximum relative error $\epsilon_r$ of each time step $N$, for the special case of the Thomas algorithm

Finally to confirm that the LU decomposition method generated the same results as the Thomas algorithm we also compared the LU algorithm outputs with the closed-form solution. This result can be seen in figure 3.

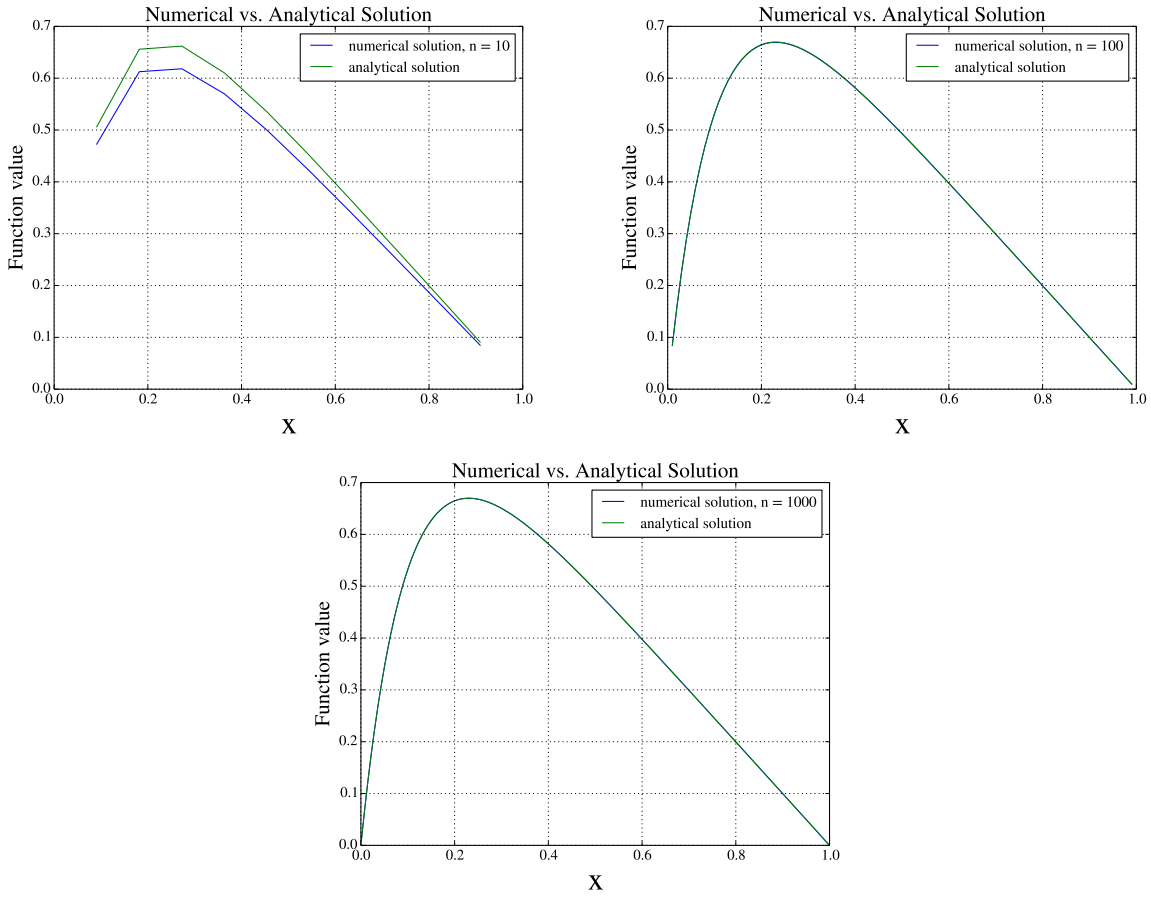


FIG. 3. The different plots shows the LU algorithm's output to the closed-form solution for $N = 10, 100$ and 1000.

## V. DISCUSSION

From figure 1 it is clear that the results from the Thomas algorithm are converging towards the closed-form solution for an increasing number of time steps. The first and last item of the solution vector is not included in the graphics due to them being zero, given by the boundary conditions.

During our numerical experiment we have compared three different algorithms whose outputs, in a perfect world where computers handle floating point numbers with exact precision, are supposed to be identical. An interesting feature to consider upon deciding which algorithm is the most suited for your purpose is the CPU time used. The CPU time is the time used by the computer's central processor for processing instructions of a computer program. From table I it is evident that the LU decomposition algorithm is the least favourable method. This has to do with the number of flops, and the fact that the algorithm operates with a dense matrix. For time steps $N \geq 10^5$ the matrix would require $N \times N$ memory slots to contain every number in the matrix. This exceeds

the memory available in our computers, and therefor we are not able to run the LU algorithm.

The difference in CPU time between the general and special Thomas algorithm is significantly less. On average the special case of the Thomas algorithm is 35% faster than the general case. This result is as expected based on the number of flops. Since the special Thomas algorithm is specialized for our problem, the general Thomas algorithm would be the preferred method on a general basis. When dealing with non-tridiagonal matrices, the Thomas algorithm that we derived is no longer useable. For that reason the LU algorithm should not be totally excluded when dealing with matrices of reasonable small dimensions.

A potential problem with the Thomas algorithm is the division with the variable $\tilde{d}$ in equations 10, 9 and 11. For very small $\tilde{d}$ the expression would approach infinity, which would ultimately lead to overflow. In our case $\tilde{d}$ will never be smaller than one, since the matrix we are dealing with is positive-definite. This becomes evident in the special case of the Thomas algorithm and the analytical expression for $\tilde{d}$ (eq. 12). Even though this is not a problem for our specific problem, it is important to keep in mind for further use with the Thomas algorithm.

In the final expression of the error analysis (eq. 18), the first term corresponds to the mathematical error due the truncation of the Taylor expansion and the second to loss of numerical precision. From result 19 we know the loss of numerical precision becomes the dominant factor at step-length $h \approx 10^{-6}$. This coincides with the left side in figure 2, where the turning point for the relative error is roughly $h \approx 10^{-6}$. This means that for step-lengths $h < 10^{-6}$ the results may no longer be reliable.

In reality the CPU times changed slightly each time we ran our program. We concluded they change according to our computers effort based on other tasks being handled, not from the algorithm itself, and since the differences were relatively small, we used numbers from a random run in our results. What we wanted to use our results from the CPU time counter for was to look at the relationship between CPU time used by the different methods, not specifically what time each method on its own would use. We found that even though the numbers them selves changed marginally for each run, the correlation between them did not.

## VI. CONCLUSION

In this paper we focused on the importance of making good approximations by looking at our limits when it comes to numerical errors. We explored different methods of numerically solving a

second order linear equations, and found that being able to specialize the methods for our specific purpose proved efficiency in both CPU time used and in limiting numerical error, as the results from the specialized Thomas Algorithm clearly shows. We found that as expected, shorter step-length increased the precision of our results, but only to a certain point. Figure 2 clearly shows that decreasing the step-length only improves our results up to the point where numerical errors from the computers limitation in handling of floating point numbers completely takes over the solution. For further work on this topic it would be interesting to investigate the possible existence of a formula that connects optimal step-length with the number of floating point operations of the specified algorithm.

---


[1] Karniadakis, G. E., Kirby II, R. M. (2003). In Parallel Scientific Computing in C++ and MPI. Pages 359-361. https://doi.org/10.1017/cbo9780511812583

[2] GitHub adress for Project 1 in FYS3150, https://github.com/KasparaGaasvaer/FYS3150/tree/master/PROSJEKT1

[3] Jensen, M. H., Lecture in FYS3150 29/8-19.