Draft version September 8, 2019
Typeset using LaTeX **twocolumn** style in AASTeX62

Numerical Solutions to ODEs by Means of Matrix Operations

Håkon Tansem, Nils-Ole Stutzer, and Bernhard Nornes Lotsberg

Abstract

We compare numerical solution methods to the static Poisson equation. We implement the Thomas algorithm to solve the equation as a tridiagonal linear algebra equation. We see that increasing the matrix size (decreasing step size) linearly decreases the error when comparing with an analytical solution. This decrease dominates the error until we reach approximately n 10^5 steps, in which numerical round-off increases the error again. We specialized our tridiagonal algorithm using the fact that our matrix has constant diagonals, reducing the algorithm cost from $O(9n)$ to approximately $O(4n)$. This does not dramatically improve run-times, but could in theory improve by parallelizing the initial setup.

Comparing our algorithm with the standard LU-decomposition method of solving linear algebra problems, proves that tridiagonal systems should always be solved using the Thomas algorithm. The LU-method uses too much memory to be viable when approaching matrices of the size n 10^5, and also has significantly higher run times.

1. INTRODUCTION

A fundamental characteristic of processes in nature is that things vary with time and space. Thus in order to understand a physical process it is essential to be able to formulate such a change in terms of differential equations, as this how a change is formulated in mathematical terms. Furthermore since most processes in nature are highly complex, making analytical solutions of corresponding differential equations unobtainable, one must often rely on numerical techniques. However, if numerics is used and a high level of accuracy is required, one must use a computer to solve the equations. An understanding of how computers handle numbers and can cause errors is thus important.

In this paper we will explore problems often encountered when solving differential equation numerically on a computer, by looking at a one-dimensional Poisson equation often encountered in physics as a concrete example. We will explore how efficient matrix algorithms like the Thomas algorithm ((**?**)) and LU-decomposition as shown by (Lay et al. 2015, p.142) and (Boyd & Vandenberghe 2004, p.668), as opposed to standard Gaussian elimination, can be used to solve ordinary differential equations (ODEs). Since these algorithms are only approximations, understanding the numerical errors is important. Thus we will also look at how the errors behave when choosing a specific grid size and when truncation errors become evident.

2. METHOD

In order to solve an ODE one can use a variety of different methods and algorithms, some more efficient than others. In this paper we will discuss three different algorithms for solving ODEs, the ultimate aim being to find the most efficient and accurate of them. To be concise we will consider a differential equation often encountered in physics, e.g. electromagnetism, namely the Poisson equation. In electromagnetism the Poisson equation relates the electric potential Φ to a charge density $\rho(\vec{r})$ as

$$\nabla^2\Phi = -4\pi\rho(\vec{r}), \tag{1}$$

for a positional vector $\vec{r}$. Assuming Φ only depends on the radial distance from the charge r, we can simply omit the angular terms in the differential equation, so that it becomes

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\Phi}{dr}\right) = -4\pi\rho(r). \tag{2}$$

To make this more convenient to handle, we use the substitution $\tilde{\Phi} = \Phi/r$, so that equation (2) can be rewritten as

$$\frac{d^2\tilde{\Phi}}{dr^2} = -4\pi\rho(r). \tag{3}$$

The ODE is now on the form

$$-u''(x) = f(x), \tag{4}$$

where the inhomogenous term $-4\pi\rho \to f$, the variable $r \to x$ and the function $\Phi \to u$. From now on we only consider equation (4) in its dimensionless form. For conciseness we impose the (Dirichlett) boundary conditions $u(0) = u(1) = 0$ and let the analytical solution $u(x)$ and the r.h.s. of equation (4) $f(x)$ be given by

$$u(x) = 1 - (1 - e^{10})x - e^{10x} \tag{5}$$
$$f(x) = 100e^{-10x}. \tag{6}$$

Since we know the analytical solution $u(x)$, we may then compare it our numerical solution.

So far everything is still continous, but to enable solving it by computer we must discretize the equation. The first step in discretizing the equations is to divide the interval $x \in (0, 1)$ on which to solve the equation, into $n + 2$ points. Then the ith grid point will be given by $x_i = ih$, for a step size $h = \frac{1}{n+1}$. The boundary points are then $x_0 = 0$ and $x_{n+1} = 1$. Next we define the ith discrete values of the solution by $u(x_i) \equiv v_i$ and let the r.h.s of the ODE be discretized by $f(x_i) \equiv f_i$. We then have $v_0 = v_{n+1}$ as boundary conditions.

Now that we have found a suitable discretized representation of the grid, we must find an expression for a discrete approximation of a second order derivative, in order to express the ODE on discrete form. We do this by first considering the Taylor expansion for the function $u(x)$ around a value x_{i+1} and x_{i-1} given as

$$\begin{aligned} u(x_i + h) &= v_{i+1} \\ &= u(x_i) + hu'(x_i) + \frac{h^2}{2!}u''(x_i) + \cdots \\ &= v_i + hv_i' + \frac{h^2}{2!}v_i'' + \cdots \\ u(x_i - h) &= v_{i-1} \\ &= u(x_i) - hu'(x_i) + \frac{h^2}{2!}u''(x_i) - \cdots \\ &= v_i - hv_i' + \frac{h^2}{2!}v_i'' - \cdots . \end{aligned}$$

Now adding these two together and solving of the double derivative we get an approximation for the second order derivative of u as

$$u_i'' = \frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} + 2\sum_{j=1}^{\infty} \frac{v_i^{2j+2}}{(2j+2)!}h^{2j}. \tag{7}$$

If we assume that the second term (the sum) is small compared to the first term, we can write

$$v_i'' \approx \frac{v_{i+1} + v_{i-1} - 2v_i}{h^2}, \tag{8}$$

and let the second term be the error of the approximation

$$\epsilon_{analy} \equiv 2\sum_{j=1}^{\infty} \frac{v_i^{(2j+2)}}{(2j+2)!}h^{2j}. \tag{9}$$

Assuming that the first term in ϵ_{analy} is dominant , i.e. the rest of the terms fall of rapidly, we can see that the numerical error should behave as $\epsilon \sim h^2$. However, since we want to solve the equation on a computer, the error will not necessarily behave like this. Nevertheless it is a good starting point in understanding the numerical error.

To complete the error analysis we first must obtain the numerical solution v_i. Inserting the numerical second order derivative into equation (4) we get

$$v_i'' \approx -\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i. \tag{10}$$

Since we already know the solutions of v_i at the boundary $i = 0$ and $i = n + 1$, we only need to consider the ODE for $i = 1, 2, 3, \ldots, n$. Next, multiplying both sides by h^2 we get the equation

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i \equiv \tilde{f}_i. \tag{11}$$

We kan now write out the ODE for different $i = 1, 2, \ldots, n$:

$$\begin{aligned} -0 + 2v_1 - v_1 &= \tilde{f}_1 \\ -v_1 + 2v_2 - v_1 &= \tilde{f}_2 \\ &\vdots \\ -v_{n-1} + 2v_n - 0 &= \tilde{f}_n, \end{aligned}$$

since $v_0 = v_{n+1} = 0$. We see that if we define vectors $\vec{v}^T \equiv [v_1, v_2, \ldots, v_n]$ and $\vec{\tilde{f}}^T \equiv [\tilde{f}_1, \tilde{f}_2, \ldots, \tilde{f}_n]$, we can write the above set of equation as a matrix equation

$$A\vec{v} = \vec{\tilde{f}}, \tag{12}$$

with a coefficient matrix

$$A = \begin{bmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{bmatrix}.$$

Because this coefficient matrix is a positive definite tridiagonal Töplitz matrix, meaning that it is not singular, we can always find a solution to equation (12). However, one can generalize this technique to a matrix equation with a tridiagonal matrix corresponding to coefficients of a different derivative approximation formula. We will thus consider a matrix

$$A = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix},$$

which can be described in its entirety by three vectors $\vec{a} = [a_1, \dots, a_{n-1}]$, $\vec{b} = [b_1, \dots, b_n]$ and $\vec{c} = [c_1, \dots, c_{n-1}]$ representing the diagonal as well as the lower and upper diagonals. The remaining matrix elements are made up of zeros, and therefore do not contain any relevant information. However, if this were to be a dense matrix, it would have to be saved in its entirety. This is an important advantage, since saving such a tridiagonal matrix requires much less memory than saving a dense matrix. Suppose we use double precision on each matrix element, meaning each matrix element takes 8 bytes of memory. Then for an $n \times n$ matrix we would need $n \cdot n \cdot 8$ bytes to save the entire matrix. If we use the fact that the matrix A is tridiagonal and only its three diagonals are saved, we only need $(n + (n-1) + (n-1)) \cdot \text{bytes} = 8(3n-2)\text{bytes}$. Thus if we require a very fine grid, meaning large n, we quickly run out of space in the computers memory.

Now that we have discussed the way to save the matrix A as three vectors, we need to find an algorithm to solve the matrix equation. The first step in that algorithm is simply a Gaussian elimination of the lower diagonal elements $a_1, \dots, a_{n-1}$. If we write out the matrix equation we get

$$b_1 v_1 + c_1 v_2 + 0 \cdots = \tilde{f}_1 \tag{13}$$

$$a_1 v_1 + b_2 v_2 + c_2 v_2 + 0 \cdots = \tilde{f}_2 \tag{14}$$

$$\vdots \tag{15}$$

$$\cdots + 0 + a_{n-1} v_{n-2} + b_{n-1} + c_{n-1} = \tilde{f}_{n-1} \tag{16}$$

$$\cdots + 0 + a_{n-1} v_{n-1} + b_n v_n = \tilde{f}_n. \tag{17}$$

Now we can take the second row and subtract the first row multiplyed by a factor a_1/b_1 in order to eliminate the first elements of the lower diagonal a_1. At the same time it changes the second diagonal element as $\tilde{b}_2 = (b_2 - \frac{a_1}{b_1} c_1)$, and the first r.h.s. element to $F_2 = \tilde{f}_2 - \frac{a_1}{b_1} \tilde{f}_1$. If this is done for every row downwards we eliminate all the lower diagonal elements. This then results in a generalized algorithm for this operation

$$\tilde{b}_i = b_i - \frac{a_{i-1}}{\tilde{b}_{i-1}} c_{i-1} \tag{18}$$

$$F_i = \tilde{f}_i - \frac{a_{i-1}}{\tilde{b}_{i-1}} \tilde{f}_{i-1} \tag{19}$$

which is calculated in a loop using $i = 1, 2, 3, \dots, n$. This algorithm is simply a forward substitution. We now have the following set of equations

$$\tilde{b}_1 v_1 + c_1 v_2 = F_1 \tag{20}$$

$$\tilde{b}_2 v_2 + c_2 v_3 = F_2 \tag{21}$$

$$\vdots \tag{22}$$

$$\tilde{b}_n v_n = F_n \tag{23}$$

The next step in solving the matrix equation is to use a backwards substitution. Since we know the nth solution $v_n = F_n/\tilde{b}_n$, we can simply insert this into the $(n-1)$-solution to solve for v_{n-1} as $v_n = (F_n - c_n v_{n+1})/\tilde{b}_n$. By doing this backwards substitution for all remaining rows we obtain the numerical solution of the ODE. This algorithm for solving ODEs is called the Thomas algorithm (**?**). In order to get a rough idea for how efficient the algorithm is, we count the floating point operations (FLOPs) for the algorithm, so as to compare its efficiency to for instance a standard row reduction of a dence matrix. A standard Gauss elimination according to ((Boyd & Vandenberghe 2004)), takes $\frac{2}{3}n^3$ FLOPs (if only the leading term is dominant). We see that there are 3 FLOPs in each of the two lines of the forward substitution in equation (18) and equation (19) and the backwards substitution, in addition to one FLOPs for the nth solution used in the backwards substitution. Since both substitutions are run n times, the algorithm in total involves $9n + 1$ flops. To further explore the algorithms efficiency we measure its run time for different matrix sizes n. Also we plot the numerical and analytical solutions $v(x)$ and $u(x)$ in the same plot, so as to see how they match.

Now that we have disscussed how we can use the Thomas algorithm to solve an ODE, we can specialize the algorithm to our original matrix used in equation (12). Since we in this special case know that all elements along each diagonal have the same value, the operations performed in the forward substitution actually become deterministic, meaning that we can precalculate the updated diagonal elements $\tilde{b}_i$ in a seperate loop outside the algorithm itself. This of course drastically reduces the amount of FLOPs, but more on that later. Since all the $a_i = c_i = -1$ and all the $b_i = 2$ we see that we get the updated diagonal elements in the following function of i

due to the forward substituion:

$$\tilde{b}_1 = b_1 \tag{24}$$

$$\tilde{b}_2 = b_2 - \frac{a_1}{b_1}c_1 = b_2 - \frac{1}{b_1} = 2 - \frac{1}{2} = \frac{3}{2} \tag{25}$$

$$\tilde{b}_3 = b_3 - \frac{a_2}{\tilde{b}_2}c_2 = b_3 - \frac{1}{\tilde{b}_2} = 2 - \frac{2}{3} = \frac{4}{3} \tag{26}$$

$$\vdots \tag{27}$$

$$\tilde{b}_i = \tilde{b}(i) = \frac{i+1}{i}, \tag{28}$$

which we save in an array filled in a loop for $i = 1, 2, 3, \dots, n$. For consistency we let $\tilde{b}_0 = \tilde{b}_{n+1} = 2$ although they are not needed for solving. Now the only thing needed in the main algorithm is the update of the f_i's as

$$F_i = \tilde{f}_i + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}}, \tag{29}$$

where we inserted that $a_{i-1} = -1$. The backwards substitution on the other hand is the exact same as in the general Thomas algorithm, only inserting $a_i = c_i = -1$. Counting the FLOPs of this specialized algorithm we see that both the forward and backward substitutions take 2 FLOPs per loop iteration due to pre-calculations, in addition to one FLOPs for the n-th solution used in the backwards substitution, giving $4n+1$ flops in total. Thus this algorithm should theoretically take half the time to run.

Now that we have solved the ODE using two different algorithms we can calculate the relative error between the numerical solution v and the analytical solution u. We calculate the relative log-error as

$$\epsilon_i = \log_{10}\left|\frac{u_i - v_i}{u_i}\right| \tag{30}$$

Furthermore we can for different grid sizes n calculate the maximum of the relative error $\max(\epsilon_i)$ over all x_i. These maximum errors are then plotted as a function of the grid size n. Doing this we obtain a graphical illustration of the accuracy of the algorithm used as a function of the grid size n. Also we may see at which grid size the loss of numerical precision due to round-off errors in the algorithms becomes evident. This loss of numerical precision should theoretically occur whenever two almost equal numbers are subtracted from eachother. Since numbers on a computer only are represented with a finite amount of digits, we can lose information by truncation of digits outside the range of representation ((Knut 2017, ch 4.2)).

Since we have a mathematical expression given by equation (9) for the error of the second order derivative approximation, we can compare the relative error computed to the theoretical behavior of the mathematical error. As discussed earlier, the analytical error $\epsilon_{\text{analy}} \sim h^2$ assuming the first error term is dominant. Therefore when plotting the maximum of the relative error as a function of the grid size n with logarithmic axes we should get a linear graph with slope -2 for increasing n, if the relative error follows the analytical error. However, the relative error and the analytical error should at some point start to deviate from eachother due to truncation error.

The last algorithm we will discuss in the scope of this paper is how to use LU-decomposition to solve the ODE on the form seen in equation (12). As the name of this algorithm suggests, the first step in finding the solution is to find a LU-decomposition of the coefficient matrix A. We know that A is a positive definite matrix in our case. Then we can always write $A = LU$, where L and U are lower and upper triangular matrices respectively. A choice for such a decomposition can be given by the lower triangular matrix

$$L = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ l_{n1} & \dots & l_{n(n-2)} & l_{n(n-1)} & 1 \end{bmatrix}.$$

and the upper triangular matrix

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & 0 & 0 & u_{nn} \end{bmatrix}.$$

In order to see how we use this to solve the matrix equation we take a look at equation (12) again

$$A\vec{v} = LU\vec{v} = L(U\vec{v}) = \tilde{\vec{f}}. \tag{31}$$

We now let

$$U\vec{v} \equiv \vec{y}, \tag{32}$$

reducing equation (12) to

$$L\vec{y} = \tilde{\vec{f}}. \tag{33}$$

Next we solve equation (33) to obtain $\vec{y}$. Since the matrix L is triangular, this is easy to solve. The computed $\vec{y}$ is then inserted into equation (32), which is very easy to solve. Because the matrix U is upper triangular, equation (32) is simply solved by a backwards substitution. Lastly, to compare the performance of the LU-decomposition method, we again run the algorithm for different grid sizes n to compute the relative error and run time each time the program is run. According to ((Boyd & Vandenberghe 2004)) the LU-decomposition in itself takes $\frac{2}{3}n^3$ FLOPs, in addition to n^2 FLOPs for both the solving of $U\vec{v} = \vec{y}$ and $L\vec{y} = \vec{\tilde{f}}$. Compared to the other two discussed algorithms the LU-method should thus be quite a lot slower. The fact that dense matrices are used poses another problem, namely that higher matrix dimensions n quickly will result in the computers RAM is exceeded when all the matrix elements are saved, as discussed earlier.

3. RESULTS

The following results were produced running on an Intel Core i7-6700HQ CPU with a clock speed of 2.60 GHz and 8Gb RAM. The computations were made using C++ using the GNU C++ compiler in Ubnutu 16.04. Solving the ODE given by equation (4) using both the Thomas algorithm and the specialized Thomas algorithm produced the plot shown in Figure (1). In this figure the numerical solutions to both the Thomas and the specialized Thomas algorithms are compared to the analytical solution given by equation (5). For each algorithm two grid sizes were used, where $n = 10$ and $n = 1000$.

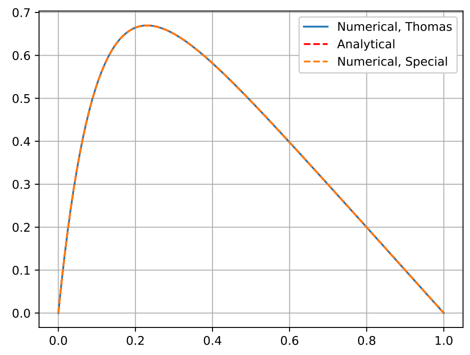

Figure 1. A figure showing the solution of equation (4) using both the Thomas algorithm and the special algorithm using $n = 1000$ compared with the analytical solution given by equation (5).

We studied the evolution of the relative log-error, given by equation (30), as the number of grid points n increases for the specialized Thomas algorithm. This produced the plot shown in Figure (2). Table (1) shows the same error for a few chosen values of n in tabular form for the three methods tested. When n reached the magnitude of approximately 10^5 the array became too large for the computers memory to handle thus making the LU decomposition unable to handle matrices of this size.

The CPU time was also calculated for all three methods. A figure of the CPU time versus the number of linearly spaced grid points n is shown in Figure (3).

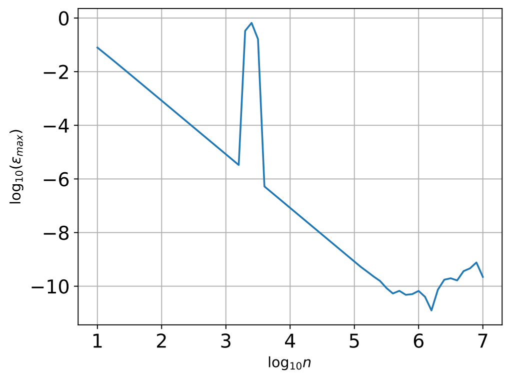

Figure 2. A figure showing the maximum relative error given by the equation (30) versus the number of linearly spaced grid points n with logarithmic axes. Note the linear behaviour in the of the graph before $\log_{10} \approx 5.5$, while there is an unpredictable behaviour above $\log_{10} \approx 5.5$.

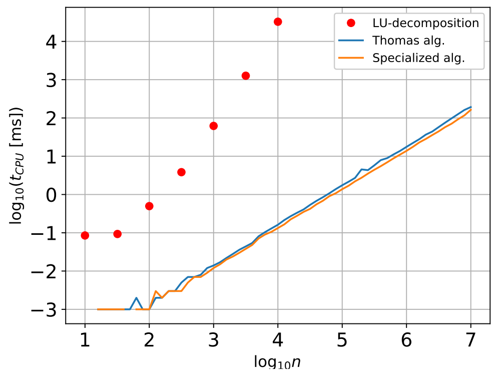

Figure 3. A figure showing the CPU time vs the number of linearly spaced grid points n with logarithmic axes for the Thomas algorithm and the specialized Thomas algorithm. Note that this is only a sample of the CPU time, as no mean over several runs was performed.

4. DISCUSSION

The first result obtained was the plots of the numerical solutions for $n = 10$ and $n = 1000$ for both the Thomas and the spesialised Thomas algorithm. Looking at Figure (1) one can clearly see that both n used

Table 1. The following table shows the relative log-error given by equation (30) for varying degrees of n for the different methods used. For n larger than 10^5 the matrix used in the LU decomposition exceeded the number of bytes available on the computers RAM.

n	Specialised algorithm	Thomas algorithm	LU-decomposition
10^1	-1.100	-1.100	-1.179
10^2	-3.079	-3.079	-3.088
10^3	-5.079	-5.079	-5.080
10^4	-7.079	-7.079	-7.079
10^5	-9.077	-8.842	Na
10^6	-10.050	-6.075	Na
10^7	-9.757	-5.525	Na

result in quite good approximations as they approach the analytical solution closely. It seems as if both algorithms behave very similarly, since the both graphs corresponding to the same n overlap. We can see that all numerical solutions start and end at the same place as the analytical solution due to the boundary conditions being imposed. The fact that the approximations for $n = 10$ already approach the analytical solution quite closely, meaning that the approximations converge to the analytical solution fast. Especially the approximations for $n = 1000$ are essentially overlapping with the analytical solution.

When comparing the computation time for the Thomas algorithm and the specialized algorithm, our results showed that in general the specialized algorithm provided a slightly lower computation time. This is shown in Figure (3). While running the program multiple times, the regular Thomas algorithm can come out faster than the specialized algorithm for smaller n. The amount of FLOPs for the regular Thomas algorithm is $9n$ while the specialized Thomas algorithm has $4n$ FLOPs. Although the specialized algorithm has less FLOPs, our results indicate that we do not gain a significant increase in runtime. For low n, the regular algorithm may also prove to be faster than the specialized algorithm. One can also see that the runtime increases linearly (in a log scale plot) with n which would be expected as the FLOPs of both algorithms scales with n. It is also worth noting that there is a lot of noise, especially for lower n which may sometimes make the regular algorithm faster than the specialized algorithm for lower n. In addition to calculating the number of arithmetic operations, we also have memory reads and memory writes. Another factor to take into consideration would be background applications running on the computer as these are not constant for each time the program is run. A reasonable way to improve the results would be to take the average after computing the run time for the same n multiple times.

While looking at Figure (3) one can see that for the LU decomposition, the runtime is a lot higher than for the other methods. Since this method has $2n^3/3 + 2n^2$ FLOPs this result is expected. Since we have a tridiagonal matrix the LU decomposition will perform excessive computations that are not needed. This is due to the additional memory needed to represent a dense matrix. Therefore this method should not be applied on this kind of problem. However, if one would want to solve some arbitrary matrix equation, with a non-singular positive definite matrix, this would be an efficient method as handling triagonal matrices L and U is quite easy.

While looking at the relative maximum error given by equation (30) for an increasing n, one can see from (2) that the logarithm of the error drops with a slope of approximately -2. This is shown in Figure (2). This is expected as the error should theoretically behave as $\epsilon \sim h^2$, making the error decrease with a slope of -2 in a plot with logarithmic axes. A notable feature is that at approximately $n = 10^6$, the error changes behaviour. It stops falling off linearly and begins increasing in a chaotic manner. A reason for this may be loss of numerical precision. When n becomes significantly large the steps in our computational grid become accordingly small. Doing arithmetic operations on small number can lead to round off and truncation errors ((Knut 2017, ch 4.2)). These errors can propagate and negatively influence the solution.

Looking at the error for the regular Thomas algorithm, as shown in Table (1), one can see that the onset of this behavior starts earlier. Although the increase in speed may not be hugely significant, the specialized Thomas algorithm allows us to perform computations with higher resolution while still maintaining precision. A reason for this may be the amount of FLOPs required.

Since the regular algorithm requires more FLOPs, there will be more arithmetic operations performed on small numbers which can make the error propagate faster. However, an advantage of the regular Thomas algorithm is its versatility, as it is a generalized method also being able to handle derivative approximations with coefficients changing along the diagonals of the coefficient matrix A. When n exceededs a value of around 10^5, the matrix used in the LU decomposition became too large for the computers memory to handle. This resolution is too rough for any significant error to separate the three algorithms.

5. CONCLUSION

We have tested three different algorithms for solving an ODE often encountered in physics. We compared the Thomas algorithm with a specialized Thomas algorithm optimized for our case of a symmetric tridiagonal Töplitz matrix where all elements on both super-diagonals are equal. We also compared the two latter mentioned algorithms to the standard LU-decomposition for solving a linear set of equations. Both the Thomas algorithm and the specialized Thomas algorithm proved far superior to the LU-decomposition for this problem, as the LU-decomposition consumed way more CPU time. While the specialized algorithm did not seem to provide a huge improvement to computation time, it provides the ability to apply a larger resolution before any error propagates making it a preferred algorithm of choice for this problem. However, more reasearch is needed to definitively state whether the Thomas or specialized Thomas algorithm is superior, as we did not measure any mean of the run time for each n, and since we could not rule out hardware related error sourses (such as background processes and memory reads and writes) to our measurments.

REFERENCES

Boyd, S., & Vandenberghe, L. 2004, Convex Optimization (New York, NY, USA: Cambridge University Press)

Jensen, M. H. 2015, Computational Physics, Lecture notes Fall 2015, , , https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf, Retrieved 6.9.2019

Knut, M. 2017, Numerical Algorithms and Digital Representation, , , https://www.uio.no/studier/emner/matnat/math/MAT-INF1100/h17/kompendiet/matinf1100.pdf, Retrieved 6.9.2019

Lay, D., Lay, S., & McDonald, J. 2015, Linear Algebra and Its Applications, Always Learning (Pearson Education Limited).

https://books.google.no/books?id=CeL5rQEACAAJ