Solving A Second Order Differential Equation Using Tridiagonal Matrix Algorithms

Erlend Andreas Longva, Astrid Bragstad Gjelsvik
Department of Physics, University of Oslo, Norway
(Dated: December 13, 2020)

In this report we explore some different methods of solving the tridiagonal matrix equation for a second order differential equation. We found that the FLOPS of the general algorithm could be more than halved by developing a special algorithm. For $n = 10^3$ steps the CPU time was reduced from 0.3 millisecond to ≈ 0.15 millisecond. We also found that the relative error was minimized for $n \approx 10^6$. The LU decomposition method was found to use ≈ 0.5 second for $n = 10^3$.

I. INTRODUCTION

In a lot physical problems, we are interested in solving differential equations numerically. Many important equations can be written as linear second-order differential equations. One example is Poissons's equation from electromagnetism, which describes how an electrostatic potential Φ is generated by a localized charge distribution $\rho(\mathbf{r})$. In three dimensions, this equation reads

$$\Delta^2\Phi = -4\pi\rho(\mathbf{r})$$

For one dimension, we can rewrite it to

$$-u''(x) = f(x)$$

where f is the source term. This form of linear second-order differential equation can represent many different physical problems as well, and it is of great interest to find good ways of solving it numerically. We are also interested in solving them with as little computational expense as possible. In this project we explore different methods for solving the equation numerically. We investigate how we can write our problem on matrix form, and use both algorithms for tridiagonal matrices and LU-decomposition to solve the problem. We compare computation time for different methods, and discuss how the resolution we solve the equation with affects the numerical error.

In the following theory section we will discuss the mathematics behind our tridiagonal matrix algorithms in the theory section, and how we set up our problem. In the methods section we explain we how we implemented this theory in our programs, and how we performed the LU-decomposition. After that we present our results. Then follows a discussion of the results, and finally a conclusion.

II. THEORY

In this project we solve the one-dimensional Poisson equation $-u''(x) = f(x)$ by rewriting it as a set of linear equations. We solve it numerically on the interval $x \in (0, 1)$ and with Dirichlet boundary conditions $u(0) = u(1) = 0$. To do this we find a discretized approximation v_i to u with grid points $x_i = ih$ where h is our step size $1/(n+1)$ and we have $n+2$ grid points in total. The first and last points of v are given by

$$v_0 = 0, \quad v_{n+1} = 0.$$

To find the other n points, we approximate the second derivative of u so that for point each $i = 1, \ldots, n$ our Poisson equation becomes

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

where $f_i = f(x_i)$. By multiplying with h^2 on both sides of the equation we can write this as

$$-v_{i-1} + 2v_i - v_{i+1} = d_i,$$

where $d_i = h^2 f_i$. This means that we have n linear equations that look like

$$\begin{aligned}
-v_0 + 2v_1 - v_2 &= d_1 \\
-v_1 + 2v_2 - v_3 &= d_2 \\
&\vdots \\
-v_{n-2} + 2v_{n-1} - v_n &= d_{n-1} \\
-v_{n-1} + 2v_n - v_{n+1} &= d_n
\end{aligned}$$

where $v_0 = v_{n+1} = 0$. This means that we have a linear set of equations that we can write on the form

$$\mathbf{A}\mathbf{v} = \mathbf{d},$$

where $\mathbf{A}$ is a $n \times n$ tridiagonal matrix

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

so that

$$\begin{bmatrix}
2 & -1 & 0 & \ldots & \ldots & 0 \\
-1 & 2 & -1 & 0 & \ldots & \ldots \\
0 & -1 & 2 & -1 & 0 & \ldots \\
 & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & \ldots & & -1 & 2 & -1 \\
0 & \ldots & & 0 & -1 & 2
\end{bmatrix}
\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \ldots \\ \ldots \\ v_n \end{bmatrix}
=
\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \ldots \\ \ldots \\ d_n \end{bmatrix}$$

A. General algorithm for tridiagonal matrix

Since **A** is tridiagonal matrix we can solve these equations for **v** using a particular algorithm, often known as the Thomas algorithm. The general algorithm, for a case in which the elements on the diagonal and non-diagonals of the matrix are not respectively identical, is based on sorting the elements of the tridiagonal matrix into three vectors of length n. We can call these vectors $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$, and they are distributed in **A** so that

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

To find **v** we want to a decomposition so that our matrix is on a form where the elements below the diagonal are zeroes and the elements on the diagonal are ones. In doing so we get new values in the **c** and **d** vector, and we can call these new vectors $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{d}}$. In the first equation we do this by dividing with b_1 on both sides and get

$$v_1 + \frac{c_1}{b_1} v_2 = \frac{d_1}{b_1} \Rightarrow v_1 + \tilde{c_1} v_2 = \tilde{d_1}$$

where $\tilde{c_1} = \frac{c_1}{b_1}$ and $\tilde{d_1} = \frac{d_1}{b_1}$ are the first elements in the new $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{d}}$ vectors. We can now do a forward substitution and use these values to find the next. In the second equation we want to eliminate the **a** vector element. We can do this by multiplying the (now manipulated) first equation with the first **a** element and subtracting it from the second equation:

$$\begin{aligned} & a_1 v_1 + b_2 v_2 + c_2 v_3 = d_2 \\ - \quad & a_1 \cdot (v_1 + \tilde{c_1} v_2 = \tilde{d_1}) \\ = \; & (b_2 - a_1 \tilde{c_1}) v_2 + c_2 v_3 = d_2 - a_1 \tilde{d_1} \end{aligned}$$

If we divide by the factor $(b_2 - a_1 \tilde{c_1})$ on both sides we now get

$$v_2 + \frac{c_2}{b_2 - a_1 \tilde{c_1}} v_3 = \frac{d_2 - a_1 \tilde{d_1}}{b_2 - a_1 \tilde{c_1}} \Rightarrow v_2 + \tilde{c_2} v_3 = \tilde{d_2}$$

where $\tilde{c_2} = \frac{c_2}{b_2 - a_1 \tilde{c_1}}$ and $\tilde{d_2} = \frac{d_2 - a_1 \tilde{d_1}}{b_2 - a_1 \tilde{c_1}}$. These new elements in the **c** and **d** vectors can again be used to find the next elements, and the general algorithm for finding element $i = 2, \dots, n$ in the vectors becomes

$$\tilde{c_i} = \frac{c_i}{b_i - a_{i-1} \tilde{c}_{i-1}}$$
$$\tilde{d_i} = \frac{d_i - a_{i-1} \tilde{d}_{i-1}}{b_i - a_{i-1} \tilde{c}_{i-1}}$$

This forward substitution gives us $\sim 6(n-1)$ floating point operations (FLOPs). Since the denominator is the same for both c_i and d_i, we only have to calculate this once for each i while computing, which gives us 2 FLOPs for each i. There are 2 FLOPs in the calculation of the numerator of d_i as well for each i, and 1 FLOP from each of the divisions for c_i and d_i. This amounts to 6 FLOPs per i in total, and we have $n-1$ iterations.

After having done the forward substitution we will have equations that look like this:

$$\begin{bmatrix} 1 & \tilde{c_1} & 0 & \dots & \dots & 0 \\ 0 & 1 & \tilde{c_2} & 0 & \dots & \dots \\ 0 & 0 & 1 & \tilde{c_3} & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & 0 & 1 & \tilde{c}_{n-1} \\ 0 & \dots & & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{d_1} \\ \tilde{d_2} \\ \tilde{d_3} \\ \dots \\ \dots \\ \tilde{d_n} \end{bmatrix}$$

We can then do a backward substitution to find **v**. If we start with our nth equation, we can see that we have $v_n = \tilde{d_n}$. We can then substitute this value for v_n into our $(n-1)$th equation and get

$$\begin{aligned} v_{n-1} + \tilde{c}_{n-1} v_n &= \tilde{d}_{n-1} \\ v_{n-1} &= \tilde{d}_{n-1} - \tilde{c}_{n-1} v_n \end{aligned}$$

The general algorithm for finding v for $i = n-1, \dots, 1$ thus becomes

$$v_i = \tilde{d_i} - \tilde{c_i} v_{i+1}$$

This backwards substitution gives us 2 FLOPs for each i. Since we have $n-1$ iterations, this gives us $\sim 2(n-1)$ FLOPs in total. The total amount of FLOPs in this general algorithm for a tridiagonal matrix should in other words be $\sim 8(n-1)$.

B. Specialized algorithm for tridiagonal matrix

In our case it is also possible to develop a more specialized algorithm, using the fact that the matrix has identical matrix elements along the diagonal and the non-diagonal elements, respectively. We can use this to develop a simpler algorithm for the forward substitution. If we write out first substitutions we see that we have

$$\begin{aligned} & \text{I.} \quad 2v_1 - v_2 = d_1 \\ & \Rightarrow v_1 - \frac{1}{2} v_2 = \frac{1}{2} d_1 \end{aligned}$$

This first equation gives us $\tilde{c_1} = -\frac{1}{2}$ and $\tilde{d_1} = \frac{1}{2}$. If we use now look at our second equation and subtract the previous equation multiplied with the factor -1 from it,

we get

$$\text{II.} \quad \frac{3}{2}v_2 - v_3 = d_2 + \frac{1}{2}d_1$$
$$\Rightarrow \frac{3}{2}v_2 - v_3 = d_2 + \tilde{d}_1$$
$$\Rightarrow v_2 - \frac{2}{3}v_3 = \frac{2}{3}(d_2 + \tilde{d}_1)$$

If we repeat this for the third equation again, we get

$$\text{III.} \quad v_3 - \frac{3}{4}v_3 = \frac{3}{4}(d_3 + \tilde{d}_2)$$

and we start to see a pattern. We see at the $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{d}}$ elements are given by the algorithm

$$\tilde{c}_i = -\frac{i}{i+1}$$
$$\tilde{d}_i = \frac{i}{i+1}(d_i + \tilde{d}_{i-1})$$

Since $\tilde{\mathbf{c}}$ no longer depends on any previous elements, only on i, we can calculate it outside of our forward substitution loop. This means that we should reduce our number of FLOPs quite significantly. Our forward substitution now has only 2 FLOPs. These are the addition of d_i and $\tilde{d}_{i-1}$ and the multiplication with $\frac{i}{i+1}$, which we can extract from $\tilde{\mathbf{c}}$. In other, this specialized algorithm should only have $\sim 4(n-1)$ FLOPs in total, if we include the backward substitution.

III. METHOD

For all program code, output files and plots obtained see our: GitHub-Repository

The goal of this project is to compare different methods for solving systems of linear equations. We start out by implementing a general algorithm that can solve for a diagonal matrix with diagonal b and two off diagonals a and c. We then use our explicit matrix and specialize the algorithm to reduce floating point operations as described in the theory section. Lastly we look at the LU decomposition method, which we implement using the `LU` and `solve` functions from the armadillo library.

A. The general algorithm

The first algorithm we implement is known as the tridiagonal matrix algorithm or Thomas algorithm, after Llewellyn Thomas. It is a form of Gaussian elimination. The full derivation of the method is given in the theory section. To generate vectors we use the armadillo library. We first implement a c++ function `tridiag_general` that takes four vectors, the diagonal and off diagonals of the matrix a, b and c, together with the known vector d and the number of time steps n.

We structure our main program so that it can loop over increasing values of n by iterating over powers i using $n = 10^i$ up to a maximum given as a command line argument. This gives us some flexibility when running our program code. From the command line we also take the filename of the output data file. For each iteration of power we generate a new data file. To solve the first task we iterate over $n = 10, 100$ and 1000 and compare these with the closed-form solution to the differential equation. We generate our plots by importing the data into `numPy` arrays using a Python program and plot them using `matplotlib.pyplot`. The resulting plots are shown in figure 1.

B. Comparing the specialized and general algorithms

We then want to test how much the specialized algorithm, as described in the theory section, increases the speed of our program. To do this we let both algorithms work their magic up to a maximum $n = 10^6$ and time both algorithms independently. To compare the times we write them out to a file. This was then repeated five times and taken the average of, thus increasing the precision of the data. The resulting values are presented in table I.

C. Calculating the maximum relative error

We also implement a program that use the specialized algorithm to solve the problem up to $n = 10^8$ and for each power iteration computes the relative error. We solve $|(\mathbf{v} - \mathbf{u})/\mathbf{u}|$ and using a handy armadillo function pick out the maximum relative error for each iteration. We then take the logarithm of the maximum value. These values are then stored in an array which is saved in an output file. The results are then plotted, as shown in figure 2.

D. LU decomposition

As a comparison to our two previous algorithm we also implement the more "brute force" method of LU decomposition. To do this we make good use of armadillo and simply create our differential operator matrix A together with the known vector d. These are then sent to a function that makes use of the armadillo LU function and the solver function. From the LU function we get the lower triangular matrix L and the upper triangular matrix U which is used in turn in the solver function with an intermediate unknown array y to produce the array of solutions.

For this method we are mostly interested in the time usage of the algorithm. Hence we do not save any of the

solutions but run the algorithm for $n = 10, 100$ and 1000 and save the cpu time for each of the iterations in an output file. These time values are shown in table II.

IV. RESULTS

A. General algorithm

Using the general algorithm we get the results shown below compared with the exact closed-form solution. As aforementioned we run the algorithm for different values of n. The results are not terrible for $n = 10$, but far from great. For $n = 100$ and 1000 the results are fine. Later we will have a closer look at the relative error for the special algorithm.

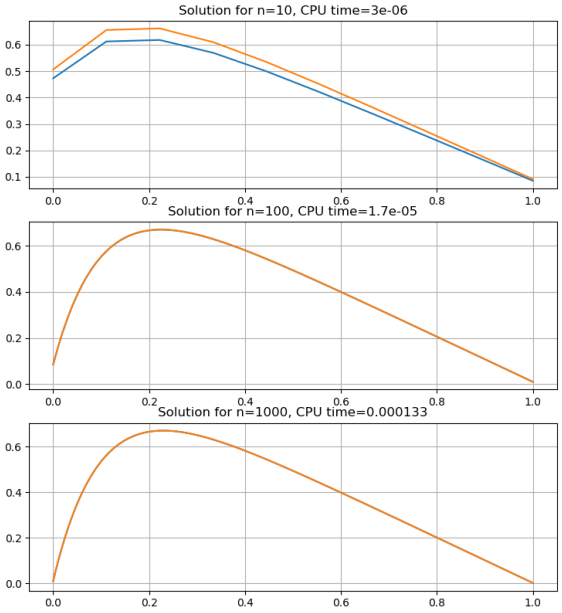

Figure 1. Comparison of the numerical solution using the general algorithm with the exact closed form solution. The numerical solution is shown in blue, while the exact is shown in orange.

B. Specialized algorithm

We record the elapsed CPU time of both the specialized and the general algorithm to see if our theoretical assumptions based off of the number of FLOPS are true. This is done up to $n = 10^6$. Our results are shown below in table I.

Table I. Comparison of CPU time.

steps	time (general algo) [ms]	time (specialized algo) [ms]
10	$9.00\ 10^{-3}$	$5.00\ 10^{-3}$
10^2	$3.90\ 10^{-2}$	$1.90\ 10^{-2}$
10^3	0.300	0.148
10^4	3.05	1.49
10^5	16.0	6.00
10^6	94.5	51.8

C. Relative error

We also want to look at how the relative error changes as n increases. Our intuition would suggest that it simply go down as n increases. Our results show that there is an optimal value around $n = 10^6$ before the relative error starts to increase. The results are presented below in figure 2.

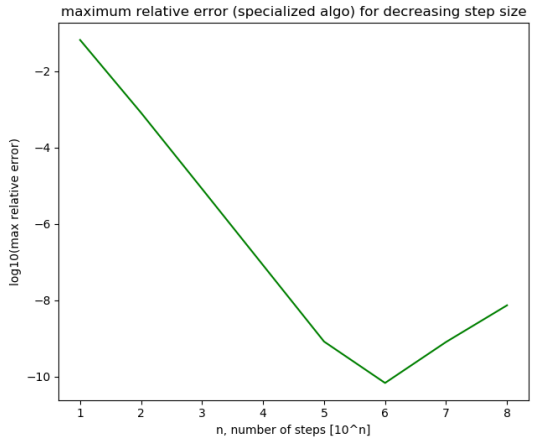

Figure 2. Plot of logarithm of the maximum relative error for a given number of steps 10^n using the specialized algorithm.

D. LU decomposition

Lastly we had a look at the elapsed CPU time for the LU decomposition method of solving our problem. Our results clearly indicate that the LU decomposition method is far slower than that of the general and specialized algorithms. It also does not increase linearly like the general and specialized algorithms but exponentially. The resulting CPU times are presented in table II.

V. DISCUSSION

In this project we have studied different methods of solving systems of linear equations, more specifically the

Table II. CPU time for LU decomposition method.

steps	time (LU decomp) [s]
10	0.00119
10^2	0.0204
10^3	0.516

tridiagonal matrix, and attempted some simple experiments to test the methods. We implemented the general algorithm for solving a tridiagonal matrix equation and observed that it did the job quite nicely. For $n = 10$ the accuracy is not great, but this is to be expected since we employ a second order integration method. What is most important to take away from the comparison results is that our method works as intended, hence this result should be regarded as a simple test. It is always good programming practise to compare your numerical solution with some analytically known solutions before proceeding. This is especially important if the actual systems to be solved does not have any exact closed-form solutions.

We specialized our tridiagonal algorithm to reduce the number of flops. To test the decrease in CPU time we tested this and found that it took about half the CPU time. The results varied a bit so we attempted to do repeated runs of the program and find the mean values of the elapsed time. We still did not get very consistent results. This could have something to do with the computer, where the computer should be viewed as a laboratory system of sorts. The CPU could for example be heated resulting in slower clock speed and thus slower computations. For future projects it might be worth while to run the programs on different systems and perhaps do a clean reboot of the system before running the programs to collect more precise data. We can still clearly read from the results that the CPU time increases linearly for every added power in n which fits with our predictions, and that the specialized algorithm almost halves the CPU time needed.

The relative error of our specialized method showed us that the accuracy of our numerical method is capped at around $n = 10^6$. After that the method becomes unstable and the numerical truncation error becomes large enough to influence our results. If we wanted to suppress this we would need to employ a higher order integration method which would result in a more complex matrix equation and we would need to develop a new algorithm.

Finally we evaluated the LU decomposition method and found that it required a lot more CPU time than the other algorithms. This happens because the method brute forces the computer to go through every element of the matrix. This results in a lot of FLOPS used to evaluate zero multiplication something that we bypassed by developing our previous algorithm. This also lead us to anticipate an exponential relation in the time consumption for this method, which we observed in our results. An attempt was also made at running the LU decomposition method for $n = 10^5$ but this was not possible as the system ran out of memory before completing the run. However, this method can be done much faster if we implemented a code that utilized the GPU which can perform many independent FLOPS since it usually has more than a thousand cores. This makes the GPU ideal for parallel processing meaning matrix and vector operations can be done at great speeds.

VI. CONCLUSION

We have compared two different algorithms for solving the tridiagonal matrix equation. The first was on a general form which could take any value for the three diagonals. We compared this method with a closed-form solution as presented in figure 1. We then developed a specialized algorithm for a tridiagonal matrix with diagonal elements $b = 2$, and off-diagonal elements $a = -1$. Comparing the elapsed CPU times for these we found that the specialized algorithms approximately halved the CPU time. For $n = 10^3$ the general method took 0.3 millisecond, while the specialized took ≈ 0.15 millisecond. We then ran the specialized algorithm to evaluate the relative error and found that it was minimized at about $n = 10^6$. We expect this to caused by be the truncation error of our integration method. Lastly we implemented the LU decomposition method and recorded the CPU time. For $n = 10^3$ the elapsed CPU time was ≈ 0.5 second.

This project has served as a good example of how to conduct computational science and introduced some principles to compare different methods for problem solving. These exercises also helped us to understand the basics of programming in c++ and of the technicalities of using the armadillo package to do vector and matrix operations. We also learnt the importance of outlining the data flow of our programs prior to doing the programming and that good old fashioned pen and paper is your best friend in understanding what you are doing.