Solving the 1D Poisson equation

Ivar Svalheim Haugerud* and Cecilie Glittum†
Department of Physics, University of Oslo

(Dated: September 10, 2018)

We solve numerically the one dimensional Poisson equation with Dirichlet boundary conditions. The equation is discretized which makes us able to write the problem as a set of linear equations, and then solve the equation by Gaussian elimination. As the matrix we're studying is tridiagonal, we develop an efficient algorithm for solving the system. We end up with two algorithms, one for a general tridiagonal matrix, which uses $8n$ FLOPS, and one for our special case, which uses $4n$ FLOPS. We find that if we were to use the standard LU decomposition algorithm, it would use $3n^3/2$ FLOPS, and store too much data for our memory when n gets large. When solving problems numerically it is important to spesialize the algorithm for the problem you want to solve. This can reduce computation time while giving the exact same result as a more general algorithm. It is important to use the optimal steplength, we found that we minimize the error by chosing a steplength equal to $10^{-5.5}$.

I. INTRODUCTION

Linear second order differential equations (DEs) on the form

$$\frac{d^2u}{dx^2} + q(x)u(x) = f(x) \tag{1}$$

are important in several contexts in physics. In electromagnetism we have Poisson's equation, which is an DE for the electrostatic potential Φ generated by a localized charge distribution $\rho(\vec{r})$. The equation reads

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

By considering a spherical symmetric charge distribution and potential, Poisson's equation simplifies to

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\Phi}{dr}\right) = -4\pi\phi(r), \tag{3}$$

which can be rewritten to

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

by the substitution $\Phi(r) = \phi(r)/r$. This is a linear second order DE of the form given in Eq. (1) with $\phi \to u$, $r \to x$ and $f(x) = 4\pi x\rho(x)$. The equation then reads

$$-\frac{d^2u}{dx^2} = f(x). \tag{5}$$

We want to solve this equation with Dirichlet boundary conditions, in this case $u(0) = u(1) = 0$, for $x \in [0, 1]$.

Here, we use three different algorithms to solve the one dimensional Poisson equation. The first algorithm is based on Gaussian elimination, but considers a more general case. The second algorithm is a specialization of the general algorithm, while the third algorithm uses LU decomposition.

* http://www.github.uio.no/ivarsh/FYS4150
† http://www.github.uio.no/cecilgl/FYS4150

II. ALGORITHMS

The problem we want to solve, has an analytical solution, but we want to solve it numerically. To solve the problem numerically we need to discretize the equation. Instead of having continuous functions we get

$$\begin{aligned} x &\to x_i \in [x_0, x_1, x_2, \ldots, x_i, \ldots, x_{n+1}] \\ u(x) &\to u(x_i) = u_i \in [u_0, u_1, u_2, \ldots, u_i, \ldots, u_{n+1}] \\ f(x) &\to f(x_i) = f_i \in [f_0, f_1, f_2, \ldots, f_i, \ldots, f_{n+1}]. \end{aligned} \tag{6}$$

The value for x is given by $x_i = x_0 + ih$, where h is defined as $h = (x_{n+1} - x_0)/(n+1)$. For our system we let x go from 0 to 1, which means that h is given by $h = 1/(n+1)$.
To solve Poisson's equation numerically, we need an discrete approximation to the double derivative. An expression for this can be found by Taylor expansion of $u(x \pm h)$ around x.

$$u(x \pm h) = u(x) \pm hu'(x) + \frac{h^2u''(x)}{2!} \pm \frac{h^3u'''(x)}{3!} + \mathcal{O}(h^4). \tag{7}$$

Poisson's equation contains only the double derivative of $u(x)$. We see that we cancel the first and third derivatives by calculating

$$u(x+h) + u(x-h) = 2u(x) + \frac{2h^2u''}{2!} + \mathcal{O}(h^4). \tag{8}$$

This equation can be solved for u'', which gives us

$$u''(x) = \frac{u(h+h) + u(x-h) - 2u(x)}{h^2} + \mathcal{O}(h^2). \tag{9}$$

In our Taylor expansion, the truncation error was proportional to h^4, but since we divide by h^2 in the approximation of the double derivative, the truncation error in this approximation is proportional to h^2. Since we are not working with continuous functions we discretize the function by writing f_i instead of $f(x)$ and $u_{i\pm1}$ instead

of $u(x \pm h)$. With this notation Eq. (9) becomes

$$f_i = -\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} + \mathcal{O}(h^2). \tag{10}$$

When we calculate this numerically we will not include the end points of our system, since these are set by our boundary conditions, $u_0 = u_{n+1} = 0$. To get a better understanding of this algorithm we will write out the terms, and define a new array $\vec{f^*} = h^2\vec{f}$ to make it simpler

$$\begin{aligned} f_1^* &= -u_2 - u_0 + 2u_1 \\ f_2^* &= -u_3 - u_1 + 2u_2 \\ f_3^* &= -u_4 - u_2 + 2u_3. \\ &\vdots \\ f_n^* &= -u_{n+1} - u_{n-1} + 2u_n \end{aligned}$$

If we have any history working with linear algebra we can see that this set of equations can be translated to a linear algebra problem. Eq. (10) can be written as a matrix-vector product

$$\hat{A}\vec{u} = \vec{f^*}, \tag{11}$$

where $f \in \mathbb{R}^n$ and $\hat{A}$ is an $n \times n$ matrix defined as

$$\hat{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & \vdots \\ 0 & -1 & 2 & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \vdots & \vdots & \dots & 2 \end{bmatrix} \tag{12}$$

For our problem we know what $\vec{f^*}$ is, and we want to solve for $\vec{u}$. $f(x)$ is defined as

$$f(x) = 100e^{-10x}, \tag{13}$$

which we will need to use in our numerical solution. Using the method described above we will be able to approximate u very accurately. Since the system has an analytical solution,

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x}, \tag{14}$$

we can test our numerical results against the analytical. To do this we will use the following expression for the relative error

$$\epsilon = \log_{10}\left(\frac{|u_{\text{num}} - u_{\text{ana}}|}{|u_{\text{num}}|}\right) \tag{15}$$

We will not include the endpoints ($i = 0$ and $i = n + 1$) in any of our algorithms, since these are fixed to a value of zero. Therefore our indexing will start at 1 and end at n, while the values for index 0 and $n + 1$ are set to zero.

A. Gaussian elimination of tridiagonal matrix

Eq. (11) would solve the 1D Poisson equation, but in the beginning we want to be a bit more general. Our matrix is tridiagonal, so we solve the problem for a general tridiagonal matrix where none of the elements are necessary equal.

For the computation algorithm to be as fast as possible we will not define an $n \times n$ array. This would lead to storing n^2 double elements in our computer, and we would run out of memory fast. Since most of our matrix is zero's, we will instead only use the three vectors along the diagonal, and find the the algorithm by just using these vectors.

We will therefore make our matrix $\hat{A}$ consist of three different vectors, $a, c \in \mathbb{R}^{n-1}$, and $b \in \mathbb{R}^n$

$$\hat{A} = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} \tag{16}$$

The general algorithm for solving $\hat{A}\vec{u} = \vec{f^*}$ needs only two steps, a forward substitution and a backward substitution. We will derive this formula by using Gaussian elimination on the equation.

Let's find the general algorithm by using a 4×4 matrix and look for a pattern. Our system can then be written as

$$\hat{A}\vec{u} = \vec{f^*}$$

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_1^* \\ f_2^* \\ f_3^* \\ f_4^* \end{bmatrix}$$

The system we will use Gaussian elimination on, to transform the first 4×4 elements into the identity operator, is therefore

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & f_1^* \\ a_1 & b_2 & c_2 & 0 & f_2^* \\ 0 & a_2 & b_3 & c_3 & f_3^* \\ 0 & 0 & a_3 & b_4 & f_4^* \end{bmatrix} \tag{17}$$

First we use the forward substitution to remove all of the elements along the bottom diagonal. To remove element a_i we multiply the line above a_i (line i) by a constant a_i/b_i which will make the element which was previously equal to b_i equal to a_i. After this is done we subtract row i from row $i + 1$, and the element which originally was a_i will then be equal to zero. This easier to see by doing the calculations. We multiply row I of (17) by a_1/b_1 to fulfil the requirement just stated, and subtract this from

row II to get

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & f_1^* \\ 0 & b_2 - c_1 a_1/b_1 & c_2 & 0 & f_2^* - f_1^* a_1/b_1 \\ 0 & a_2 & b_3 & c_3 & f_3^* \\ 0 & 0 & a_3 & b_4 & f_4^* \end{bmatrix} \tag{18}$$

Since we have started to have some ugly terms in our system we will introduce some new variables to make the algorithm more clear, we therefore define that $\tilde{b}_2 = b_2 - c_1 a_1/b_1$, and $\tilde{f}_2 = f_2^* - f_1^* a_1/b_1$. This definition makes us able to write Eq. (18) as

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & f_1^* \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & a_2 & b_3 & c_3 & f_3^* \\ 0 & 0 & a_3 & b_4 & f_4^* \end{bmatrix} \tag{19}$$

We then have to continue with Gaussian elimination by multiplying line II by $a_2/\tilde{b}_2$ and subtracting it from line III we get

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & f_1^* \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & b_3 - c_2 a_2/\tilde{b}_3 & c_3 & f_3^* - \tilde{f}_2 a_2/\tilde{b}_2 \\ 0 & 0 & a_3 & b_4 & f_4^* \end{bmatrix} \tag{20}$$

By again using a similar redefinition, $\tilde{b}_3 = b_3 - c_2 a_2/\tilde{b}_3$ and $\tilde{f}_3 = f_3^* - \tilde{f}_2 a_2/\tilde{b}_2$, brings us to (20) to

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & f_1^* \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & \tilde{b}_3 & c_3 & \tilde{f}_3 \\ 0 & 0 & a_3 & b_4 & f_4^* \end{bmatrix} \tag{21}$$

From these calculations we can see the pattern. Doing the forward substitution gives us the expression for a general $\tilde{b}_i$ and $\tilde{f}_i$

$$\tilde{b}_i = b_i - \frac{c_{i-1} a_{i-1}}{\tilde{b}_{i-1}}, \tag{22}$$

$$\tilde{f}_i = f_i^* - \frac{\tilde{f}_{i-1} a_{i-1}}{\tilde{b}_{i-1}}. \tag{23}$$

We have now found an algorithm which removes the bottom diagonal array from our matrix. We will now need to do the backward substitution to remove the upper diagonal array in our matrix. From the forward substitution we see that row I did not change, but to make the backward substitution algorithm more elegant we define $\tilde{f}_1^* = f_1^*$ and $\tilde{b}_1 = b_1$. The system of equations has become

$$\begin{bmatrix} \tilde{b}_1 & c_1 & 0 & 0 & \tilde{f}_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & \tilde{b}_3 & c_3 & \tilde{f}_3 \\ 0 & 0 & 0 & \tilde{b}_4 & \tilde{f}_4 \end{bmatrix} \tag{24}$$

To solve the set of equations, we want the identity matrix for the first $n \times n$ elements. This is easy to achieve for the last row; we simply divide the hole row by $\tilde{b}_4$. The expression we now find in the right most column is the answer we are looking for, we will call this element $u_4 = \tilde{f}_4/\tilde{b}_4$. Our system of equations now looks like

$$\begin{bmatrix} \tilde{b}_1 & c_1 & 0 & 0 & f_1^* \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & \tilde{b}_3 & c_3 & \tilde{f}_3 \\ 0 & 0 & 0 & 1 & u_4 \end{bmatrix} \tag{25}$$

To remove c_3 we multiply line IV by c_3 and subtract it from line III. We then find

$$\begin{bmatrix} \tilde{b}_1 & c_1 & 0 & 0 & f_1^* \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & \tilde{b}_3 & 0 & \tilde{f}_3 - u_4 c_3 \\ 0 & 0 & 0 & 1 & u_4 \end{bmatrix} \tag{26}$$

We normalize line III by dividing by $\tilde{b}_3$, we then define $u_3 = (\tilde{f}_3 - u_4 c_3)/\tilde{b}_3$, which gives us

$$\begin{bmatrix} \tilde{b}_1 & c_1 & 0 & 0 & f_1^* \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & 1 & 0 & u_3 \\ 0 & 0 & 0 & 1 & u_4 \end{bmatrix} \tag{27}$$

We continue to do this for every row. It is important for this part of the algorithm to start at the bottom, and to work upwards from $i = n-1$ to $i = 1$. Next step would now be to multiply row III with c_2 and subtract it from row II, then normalizing by dividing by $\tilde{b}_2$. From these steps we can see the general algorithm for the backward substitution

$$u_i = \frac{\tilde{f}_i - c_i u_{i+1}}{\tilde{b}_i}, \tag{28}$$

for $i = n-1, n-2, ..., 1$.

Assuming pre initialized vectors `a, b, c, f` and defining as shown, $\tilde{f}_1 = f_1$ and $\tilde{b}_1 = b_1$, our general algorithm reads in pseudo code

for i = 2, 3, ..., n **do**
  quotient = $a_{i-1}/\tilde{b}_{i-1}$;
  $\tilde{b}_i = b_i - c_{i-1}*$ quotient;
  $\tilde{f}_i = f_i^* - \tilde{f}_{i-1}*$ quotient;
end for

compute $u_n = \tilde{f}_n/\tilde{b}_n$;

for i = n-1, n-2, ..., 1 **do**
  $u_i = (\tilde{f}_i - c_i * u_{i+1})/\tilde{b}_i$;
end for

We are also interested in finding the number of floating point operations (FLOPS) in our algorithm. We see that in the first loop, we do 5 FLOPS for every i and in the

second loop we do 3 FLOPS per i. Each loop is run $n-1$ times, thus this algorithm has $(8(n-1)+1)$FLOPS $=$ $(8n-7)$FLOPS. For large n, we can ignore the constant term. Our algorithm therefore uses $8n$ FLOPS. The reason for calculating the quantity `quotient` is thus to reduce the number of FLOPS from $9n$ to $8n$ as this quantity is used twice in the first loop.

B. Specialized algorithm

By using the fact that our matrix is a symmetric Toeplitz matrix with the number 2 along the main diagonal and the number -1 along the two other diagonals, we're able to modify our general algorithm with the purpose of making it more efficient for our special case.

When using the general algorithm, we have in the special case $a_i = c_i = -1$ and $b_i = 2$ for all i. When inserting these values into the Eqs.(22), (23) and (28) we get

$$\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}}, \tag{29}$$

$$\tilde{f}_i = f_i^* + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}}, \tag{30}$$

$$u_i = \frac{\tilde{f}_i + u_{i+1}}{\tilde{b}_i}. \tag{31}$$

In appendix A, we show that the recurrence relation for $\tilde{b}_i$, Eq. (29) has a solution

$$\tilde{b}_i \equiv \tilde{d}_i = \frac{i+1}{i}, \tag{32}$$

where we rename the diagonal elements from b_i to d_i to distinguish this result from the general case. This is a general result for our special matrix, and can thus be calculated outside of the algorithm for solving Eq. (11). The resulting operations to be done in the algorithm (the parts dependent on the right hand-side of the system) are

$$\tilde{f}_i = f_i^* + \frac{\tilde{f}_{i-1}}{\tilde{d}_{i-1}}, \tag{33}$$

$$u_i = \frac{\tilde{f}_i + u_{i+1}}{\tilde{d}_i}. \tag{34}$$

The specialized algorithm thus reads

for i = 2, 3, ..., n **do**
  $\tilde{f}_i = f_i^* + \tilde{f}_{i-1}/\tilde{d}_{i-1}$;
end for

compute $u_n = \tilde{f}_n/\tilde{b}_n$;

for i = n-1, n-2, ..., 1 **do**
  $u_i = (\tilde{f}_i + u_{i+1})/\tilde{d}_i$;
end for

Both the loops now use 2 FLOPS each time they are run, and each loop runs $n-1$ times. Thus this specialized algorithm uses $(4(n-1)+1)$FLOPS $= (4n-3)$FLOPS. For large n, the algorithms thus uses $4n$ FLOPS. As we see, by using the special case for our matrix, we have reduced the algorithm from $8n$ FLOPS to $4n$ FLOPS.

C. LU decomposition

LU decomposition is a method where a non-singular matrix $\hat{A}$ is rewritten as a product of a lower triangular matrix $\hat{L}$ and an upper triangular matrix $\hat{U}$

$$\hat{A} = \hat{L}\hat{U},$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} \tilde{u}_{11} & \tilde{u}_{12} & \tilde{u}_{13} & \tilde{u}_{14} \\ 0 & \tilde{u}_{22} & \tilde{u}_{23} & \tilde{u}_{24} \\ 0 & 0 & \tilde{u}_{33} & \tilde{u}_{34} \\ 0 & 0 & 0 & \tilde{u}_{44} \end{bmatrix}.$$

By writing the matrix $\hat{A}$ as its LU decomposition, we can solve Eq. (11) in two steps

$$\hat{L}\hat{U}\vec{u} = \vec{f}^* \leftrightarrow \hat{L}\vec{v} = \vec{f}^*, \quad \hat{U}\vec{u} = \vec{v}. \tag{35}$$

As $\hat{L}$ is lower triangular, the equation $\hat{L}\vec{v} = \vec{f}^*$ can be solved by forward substitution, and the remaining equation $\hat{U}\vec{u} = \vec{v}$ can be solved by backward substitution.

The approach for numerical simulations is to solve Eq. (11) with the three different algorithms. We also studied the relative error of the algorithms as function of n by using Eq. (15). Further we compared the runtimes of the different algorithms.

By reducing the number of FLOPS our code should be able to run faster. By going from $8n$ FLOPS for the general algorithm to $4n$ FLOPS for the specialized algorithm, we expect the code to run twice as fast. This is by assuming that all calculations take the same amount of time. To study this relationship we run both algorithms 1000 times for different values of n increasing logarithmic linearly. For each value of n we calculate the average and the standard deviation for both algorithms.

III. RESULTS

Using our numerical methods we find a graph which matches the analytic solution in Eq. (14). This is shown in Fig. 1 on the following page, where we have used our special algorithm in particular, even though all methods seems to give the same results.

From this graph it is hard to tell the deviation between the numerical and analytical solution. To get a better understanding of this, we plot the logarithm of the maximum relative deviation in Eq. (15) for each value of n. We plot this value versus the logarithm of n to find the maximum error as a function of n. This graph is

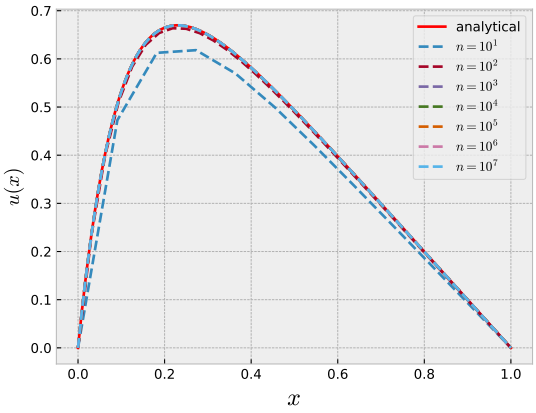

Figure 1. The solution of Poisson's equation with boundary conditions $u(0) = u(1) = 0$. The plot shows both the analytical solution as well as the numerical solution found by Gaussian elimination for various values of n using our specialized algorithm.

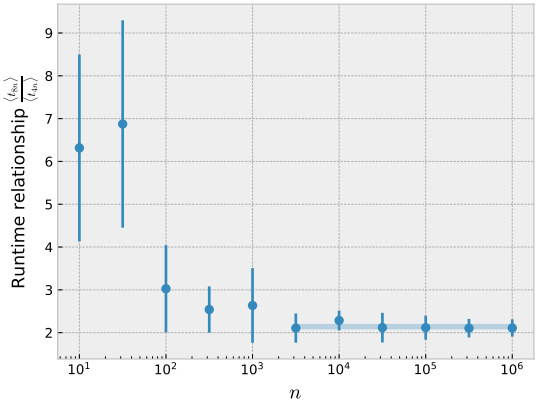

Figure 3. The relationship between the average time for each algorithm calculated by $\langle t_{8n}\rangle/\langle t_{4n}\rangle$. For each n we ran the algorithm 1000 times and calculated the mean time and standard deviation for these runs. The relationship between the runtimes stabilizes at 2.10 ± 0.02 for $\log n > 3.5$.

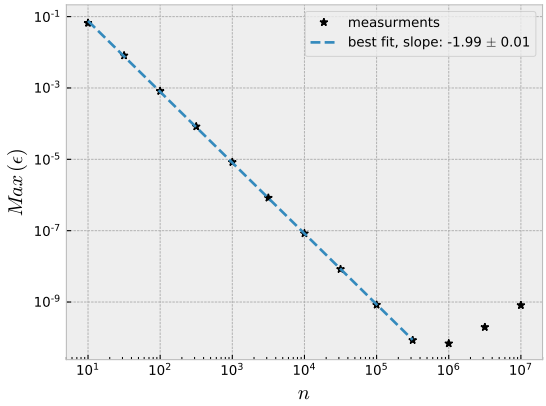

Figure 2. The maximum deviation for each n between the numerical and analytical solution of Poisson's equation. This is shown in a log-log plot to visualize the large differences, and to be able to calculate the slope. By using the least squares method we find that the slope is -1.99 ± 0.01.

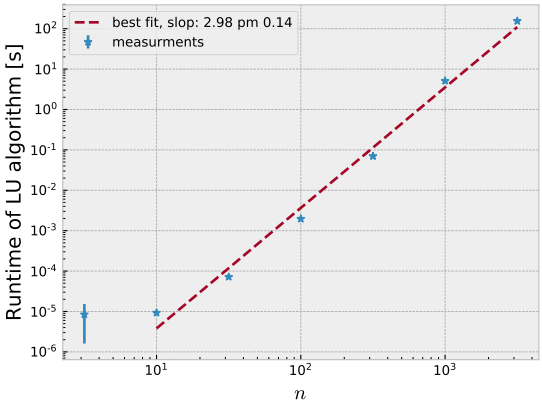

Figure 4. Runtime for LU-decomposition for different values of n for an $n \times n$ matrix. Due to the LU decomposition algorithm being slow we took the average and standard deviation of 7 runs for each value of n.

shown in Fig. 2. In this graph we have used the least squares method to find the best linear fit to the data points. Theory predicts a slope of -2, due to the truncation term going as h^2. When calculating the slope we find that the slope is -1.99 ± 0.01, where the uncertainty is calculated from the least squares method [2].

We see that the error follows this slope until $\log n \approx 5.5$. For n bigger than this, the error increases.

To study the relationship between the runtimes of the general and specialized algorithms we ran both algorithms 1000 times for different values of n increasing logarithmic linearly. How the relationship between two average runtimes depend on n is shown in Fig. 3. In this graph the relationship is calculated by $\langle t_{8N}\rangle/\langle t_{4N}\rangle$. We see in Fig. 3 that we have a large uncertainty for lower n, but as n increases the time relationship stabilizes, and the uncertainty decreases. The value it stabilizes as is calculated to be 2.10 ± 0.02.

When we study the time needed for solving the same problem numerically, but using LU decomposition, we find a logarithmic-linearly dependence on n. This is shown in Fig. 4. We calculate the slope in the log-log-plot to be 2.99 ± 0.1, this corresponds to a runtime proportional to n^3 [1].

When comparing the results produced by the specialized algorithm and the LU decomposition, we find that the results are exactly the same.

IV. DISCUSSION

LU is a general algorithm, which works for all non-singular matrices. Our algorithm, using Gaussian elimination, exploits the fact that we are working on a tridiagonal matrix. Instead of storing the whole $n \times n$ matrix we store only the three diagonal arrays. This makes us able to get a highly efficient algorithm compared to the standard LU decomposition which has to work with the whole matrix.

As we see in Fig. 4 on the preceding page the time used to calculate the LU decomposition goes as n cubed. This means that for larger values of n, the run time becomes intolerable. If we assume that the linear relationship continued we would find that the runtime using the LU decomposition would be around 1.26 months for $n = 10^5$.

By using the Gaussian algorithms the runtimes goes linearly with n. The three algorithms gives exactly the same answers, but the specialized algorithm is tailor-made for our problem, which means it will run faster. This means that we can still run the algorithm when n gets large.

Another problem by using the LU decomposition algorithm, is that we need to store more information on our computer. If we save $\hat{A}$ as a double $n \times n$ matrix, we need to save 8 bytes times n^2, only to store the data of that single matrix. For example if we want to run the LU decomposition for $n = 10^5$, we will need to store $8 \times (10^5)^2 = 8 \times 10^{10}$ bytes on our computer. This is 80 giga bytes. And if we want to store $\hat{L}$ and $\hat{U}$ as well, we need to multiply this by three. To store that much information, when most of the values are zero, is completely unnecessary. LU decomposition would need a lot of memory, and a lot of time, when n gets large. For the general and specialized algorithms, the amount of data we need to store goes linearly with n, and makes life easy.

Even though our specialized algorithm gives us the opportunity to run the algorithm for large n, we have another restriction on n, if we want precise results. When doing differentiation and integration analytically, we let the small change in the function, $\mathrm{d}x$, go towards zero. We can not do this numerically due to how computers store numbers. As we can see in Fig. 2 on the previous page the logarithm of the maximum deviation decreases linearly with the logarithm of n. But at $n = 10^{5.5}$ this trend stops, and the error gets larger. This error comes from how computers store numbers. Not all numbers can be exactly represented on computers. Since we are storing numbers using 64 bits in our algorithm, all information about the number, which would need more than those 64 bits to represent, gets rounded off. This round-off error is very small, but when we do FLOPS, for example subtraction between two very small numbers, the information we lose by round-off can be a large portion of the resulting number. This error will then play an important part in the resulting answer, and might propagate onwards in our calculation. If we increase n further than what we did in Fig. 2 on the preceding page we will see that the deviation gets even larger.

By using the fact that all the elements in each array were 1's and 2's we were able to optimize our algorithm further, by reducing the number of FLOPS from $8n$ to $4n$. The relationship between the run times for these two algorithms is shown in Fig. 3 on the previous page. As one would expect, the relationship stabilizes around 2 when n increases. For lower values of n the relationship has a higher standard deviation, and is quite far away from 2, so what is going on here?

There is more than just the FLOPS that is happening in our algorithm. For every operation the code also needs to read and write data, and index the arrays.

All FLOPS are not equivalent in computational time. For example is division more numerically challenging than addition for the computer, and will therefore take longer time.

This is the effect we see for low n. When there are few total operations there are other effects and operations which are more significant, which dictates the time-relationship between the two algorithms. We should therefore not assume to half the runtime for small n. As n increases the number of FLOPS increases, and we see that the relationship stabilizes at around 2. The other operations don't effect the total runtime as much when the number of FLOPS is large. From the graph we can also tell that the standard deviation of the mean runtime is much larger for small n, there is more at play here than meets the eye. For example is the resolution of the clock used at μ seconds, and for small n the runtime of the code is the same order of magnitude as the resolution of the clock.

The fact that the LU decomposition gives the exact same result as our own, spezialized algorithm, indicates that the algorithm, and it's implimentation, is correct.

V. CONCLUSION

We have solved the one dimensional Poisson equation, with Dirichlet boundary conditions, numerically. By discretizing the equation we are able to represent the problem as a set of equations, and solve it using Gaussian elimination. We developed an algorithm for solving this problem for a general tridiagonal matrix. This algorithm had $8n$ FLOPS, and was much more efficient, and used less memory, than a LU algorithm for an $n \times n$ matrix, which made it possible to use large values of n.

We were able to optimize the algorithm even further by creating an algorithm for the matrix used in our system. This algorithm used $4n$ FLOPS, and resulted in running twice as fast as the $8n$ FLOPS algorithm.

Even though LU decomposition can solve the set of equations for any matrix, it uses $3n^3/2$ flops, and requires more memory. It is therefore important to specialize algorithms for the specific problem you want to solve, even though it is less general, to reduce the computation time and memory usage.

VI. COMMENTS

All of the code used is available on our github [1]. Here the `README.MD` files will describe the structure of our github, and what each program was used for. The code itself will also include notes explaining what we do.

Appendix A: Analytical expression for $\tilde{d}_i$

$\tilde{d}_i$ is given by the recurrence relation

$$\tilde{d}_i = 2 - \frac{1}{\tilde{d}_{i-1}}, \tag{A1}$$

with the initial condition $\tilde{d}_1 = 2$. We can thus calculate the following values for $\tilde{d}_i$:

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

By inspection we put up the hypothesis that

$$P_n : \tilde{d}_n = \frac{n+1}{n}, \tag{A2}$$

which we have already checked for $n = 2$. We can show by induction that this must be the solution of Eq. (A1). Let us assume that P_{k-1} holds. Then,

$$\begin{aligned}
\tilde{d}_k &= 2 - \frac{1}{\tilde{d}_{k-1}} \\
&= 2 - \frac{1}{\frac{(k-1)+1}{k-1}} \\
&= 2 - \frac{k-1}{k} \\
&= \frac{2k-(k-1)}{k} \\
&= \frac{k+1}{k}.
\end{aligned}$$

Since the hypothesis P_n holds for $n = 2$, and we have shown that it holds for $n = k$ if it holds for $n = k - 1$ it must hold for all $n \geq 2$.

[1] Morten Hjorth-Jensen. Computational physics, lecture notes fall 2015. *Department of Physics, University of Oslo*, page 173, 2015.

[2] G. L. Squires. Practical physics. *Cambride Univserity Press*, 4(5):29–52, 2001.

[1] `http://www.github.uio.no/cecilgl/FYS4150`, and `http://www.github.uio.no/ivarsh/FYS4150`