Solving the Poisson equation in 1D

Carl Martin Fevang, Håkon Kvernmoen

September 13, 2020

Abstract

We solved the 1D Poisson equation using Dirichlet boundary conditions on a charge density following $f(x) = 100e^{-10x}$. We used a matrix representation of the double derivative and solved the set of linear equations with 3 different algorithms. First using an LU-decomposition with $2/3n^3$ FLOPs where $h = 10^{-1}, 10^{-2}, 10^{-3}$. Then with the Thomas algorithm using $8n$ FLOPs and finally a special case of the Thomas algorithm with known values on the diagonals reducing the FLOPs to $4n$ (both with $h = 10^{-1}$ to $h = 10^{-7}$). Using an analytical solution we analyzed the error over different step lengths and found that the optimal step length should be between 10^{-6} and 10^{-5}.

1 Introduction

In this report we are going to look at some simple numerical methods for solving the Poisson equation in one dimension. This equation is omnipresent in physics, and will often yield differential equations without any analytical solutions. The accuracy of our numerical procedures of solving are therefore obvious targets for examination. We will solve a rather simple version of the equation, with Dirichlet boundary conditions, using approximations for the second derivative and linear algebra. This allows us to examine how the granularity of our numerical procedures affects the precision of our solution, when compared to the analytical solution. Our goal is to quantitatively point to what kind of errors occur when implementing numerical methods on a computer, and to what extent.

The time complexity of an algorithm is also crucial and we will run the algorithms many times to approximate the mean time for each algorithm at different step lengths. Comparing the results we can quantitative compare the different algorithms.

2 Theory

2.1 Poisson's equation

Generally Poisson's equation[1] is written as

$$\nabla^2\phi = f$$

for a given function f and an unknown function ϕ. In electrostatics, Gauss's law[2] takes the form of Poisson's equation with Φ as the electric potential and $\rho(r)$ being the charge density

[1] Sept. 3, 2020. URL: https://en.wikipedia.org/wiki/Poisson%5C%27s_equation.

[2] Sept. 3, 2020. URL: https://en.wikipedia.org/wiki/Gauss%5C%27s_law#Differential_form.

as a function of r in a spherically symmetric system:

$$\begin{aligned} -\nabla^2\Phi &= \frac{\rho(r)}{\epsilon_0} \\ -\frac{1}{r^2}\frac{\mathrm{d}}{\mathrm{d}r}\left(r^2\frac{\mathrm{d}\Phi}{\mathrm{d}r}\right) &= \frac{\rho(r)}{\epsilon_0} \end{aligned}$$

Using the substitution $\Phi(r) = \phi(r)/r \Rightarrow \frac{\mathrm{d}\Phi}{\mathrm{d}r} = \frac{1}{r}\frac{\mathrm{d}\phi}{\mathrm{d}r} - \phi/r^2$ we get

$$\begin{aligned} -\frac{1}{r^2}\frac{\mathrm{d}}{\mathrm{d}r}\left(r\frac{\mathrm{d}\phi}{\mathrm{d}r} - \phi\right) &= \frac{\rho(r)}{\epsilon_0} \\ -\frac{1}{r}\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2} &= \frac{\rho(r)}{\epsilon_0} \\ -\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2} &= \frac{r\rho(r)}{\epsilon_0} \end{aligned}$$

With a quick change of variables we get the following

$$-\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = f(x) \tag{1}$$

This way the three-dimensional system can be solved as a Poisson equation in a single dimension.

For this specific report,we will examin a Poisson equation of this format, with a right hand side $f(x) = 100e^{-10x}$ and with Dirichlet boundary conditions $u(0) = u(1) = 0$. Solving equation (1), we get:

$$\begin{aligned} -\frac{\mathrm{d}^2u}{\mathrm{d}x^2} &= 100e^{-10x} \\ -\mathrm{d}^2u &= 100e^{-10x}\,\mathrm{d}x^2 \\ -u &= e^{-10x} + c_1x + c_2 \\ u &= -e^{-10x} - c_1x - c_2 \end{aligned}$$

Using the boundary conditions, we get:

$$\begin{aligned} u(0) = -1 - c_2 &= 0 \\ c_2 &= -1 \end{aligned}$$

$$\begin{aligned} \Rightarrow u(1) = -e^{10} - c_1 + 1 &= 0 \\ c_2 &= 1 - e^{-10} \end{aligned}$$

So our analytical solution is then

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{2}$$

2.2 Numerical discretization

When we solve problems numerically we can't use continuous variables as we normally do in mathematics. For the variable x we can only operate with discrete values such $x_0, x_1, x_2...x_n$ where $x_i \in [x_0, x_n]$. For the function $f(x)$ we can project values $f_1, f_2, ...f_n$ such that $f_i = f(x_i)$. Similar projections can be done for all variables.

Because the need for discretization in a computer, infinitesimal differences, such as in differentiation, cannot be computed accurately. From the definition of the derivative, however, we can get an approximation. The first derivative of an arbitrary function $f(x)$ is defined accordingly

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h}$$
$$\Downarrow$$
$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} \approx \frac{f(x+h)-f(x)}{h}$$

Using our discretized variable x_i and letting it be linearly space in the interval $[x_0, x_n]$ such that $x_i = x_0 + ih$, we get a discretized approximation for the derivative

$$\frac{\mathrm{d}f_i}{\mathrm{d}x} \approx \frac{f_{i+1}-f_i}{h} \tag{3}$$

This introduces a mathematical error, that grows with the step size h.

Defining a function $g(x) = \frac{\mathrm{d}f(x)}{\mathrm{d}x}$, we can find the second derivative of $f(x)$ as $\frac{\mathrm{d}g(x)}{\mathrm{d}x}$. Using equation (3), we can thereby get an approximation for the second derivative:

$$\frac{\mathrm{d}^2 f_i}{\mathrm{d}x^2} \approx \frac{f_{i+1}+f_{i-1}-2f_i}{h^2} \tag{4}$$

This leaves a numerical error that follows h^2.

Combining the approximation (4) and equation (1), we get a discretized variant

$$\frac{u_{i+1}+2u_i-u_{i-1}}{h^2} = f_i \tag{5}$$

2.3 Some linear algebra

The above discretization of our differential equation can be written as a matrix equation, allowing us to use some manipulations from linear algebra. The equation reads

$$\mathbf{A}\mathbf{u} = \mathbf{b}$$

where $\mathbf{A}$ is an $n \times n$ matrix where the diagonal elements equal 2 and the elements above and below the diagonal are -1. $\mathbf{u}$ is a vector of length n and contains the unknown solution to our problem, and b is a vector of length n that contains the known right hand side function values multiplied by the step size h as $h^2 f(x_0), ..., h^2 f(x_n)$. The solution can be optained with straight Gaussian elimination, but we will look at some better implementations to reduce computation time.

As long as the equation is solvable, we know that $\mathbf{A}$ must be an invertible matrix. Furthermore, the matrix can be broken up into a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ such that $\mathbf{A} = \mathbf{L}\mathbf{U}$. This is called a LU-decomposition. Using this allows us to break up the equation into two parts that are easier to solve due to the properties of $\mathbf{L}$ and $\mathbf{U}$:

$$\mathbf{L}\mathbf{y} = \mathbf{b}$$
$$\mathbf{U}\mathbf{u} = \mathbf{y}$$

3 Implementation

All the programs (algorithms, plotting and benchmarking) are available at https://github.com/haakonkvernmoen/FYS3150

We will use three algorithms, each more specific for our problem. The first algorithm we will use to solve our Poisson equation for n equally space integration points is an LU decomposition. This algorithm requires the use of $n \times n$-matrices, and as such is both computationally heavy and cumbersome on memory. Further, we will show that our discretized Poisson equation leads to a tridiagonal matrix that we can solve using the Thomas algorithm. This algorithm will only use vectors of length n and should be must quicker and more wieldy than the LU decomposition. Lastly, we will make some further simplifications to this algorithm, using some properties of the matrix specific to our problem.

The matrices and vectors we use have dimensions of $n \times n$ and n respectively. Since we are using Dirichlet boundary conditions $u(0) = 0$ and $u(1) = 0$ are not included in the matrices/vectors (they are known). This leaves us with a total of $n + 2$ points and a step length of $h = 1/(n + 1)$.

We will examine the relative error of this last algorithm compared with the analytic solution as a function of the step-size h. Lastly, the algorithms will be timed and comparisons will be made to examine the efficiency of the algorithms in practice.

3.1 LU decomposition

To find the LU decomposition of our matrix **A**, we will us the Doolittle algorithm. Our matrix **A** can be broken down as follows

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

Writing out the multiplication we obtain an expression for the a_{ij} element of the matrix **A**.

$$a_{ij} = \sum_{s=1}^{n} l_{is} u_{sj} \tag{6}$$

We know that for $s > i \Rightarrow l_{is} = 0$ and for $s > j \Rightarrow u_{sj} = 0$. This implies that we can construct **U** row-by-row and **L** column-by-column. From this we can get the entries for the first row of u's and first column of l's with the use of the expression 6.

$$\begin{aligned} a_{1j} &= \sum_{s=1}^{1} l_{1s} u_{sj} = l_{11} u_{1j} \Rightarrow u_{1j} = \frac{a_{1j}}{l_{11}} \qquad j = 1, 2, ..., n \\ a_{i1} &= \sum_{s=1}^{1} l_{is} u_{s1} = l_{i1} u_{11} \Rightarrow l_{i1} = \frac{a_{i1}}{u_{11}} \qquad i = 1, 2, ..., n \end{aligned}$$

Next we will find the k-th row for **U** and the k-th column for **L** assuming that we correctly found the $1, 2, ..., k - 1$ row/column. Again using the expression 6 we find that

$$a_{kk} = \sum_{s=1}^{k} l_{ks} u_{sk} = \sum_{s=1}^{k-1} l_{ks} u_{sk} + l_{kk} u_{kk}$$

As we can see one of the diagonals u_{kk} and l_{kk} can be chosen rather arbitrary. From convention we will set $l_{kk} = 1$ and solve for u_{kk}. Finally we again use the expression 6 and find the k-th row of $\mathbf{U}$ and the k-th column of $\mathbf{L}$

$$a_{kj} = \sum_{s=1}^{k} l_{ks} u_{sj} = \sum_{s=1}^{k-1} l_{ks} u_{sj} + l_{kk} u_{kj} \Rightarrow u_{kj} = a_{kj} - \sum_{s=1}^{k-1} l_{ks} u_{sj} \quad j = k+1, ..., n \tag{7}$$

$$a_{ik} = \sum_{s=1}^{k} l_{is} u_{sk} = \sum_{s=1}^{k-1} l_{is} u_{sk} + l_{ik} u_{kk} \Rightarrow l_{ik} = \frac{a_{ik} - \sum_{s=1}^{k-1} l_{is} u_{sk}}{u_{kk}} \quad j = k+1, ..., n \tag{8}$$

We can now iterate over the rows and columns and obtain the elements for the $\mathbf{L}$ and $\mathbf{U}$ matrices using expression 8 and 7 respectively. Note that since we forced $l_{kk} = 1$ there is no denominator in expression 8 and we have to find the elements for $\mathbf{U}$ before we find the elements for $\mathbf{L}$. We also assume that $u_{kk} \neq 0$ to avoid division by zero errors.

Although not straight forward to analyze, the number of FLOPs this algorithm requires is $\frac{2}{3}n^3$, ignoring the lower-order terms.[3]

3.2 Thomas algorithm

The Thomas algorithm seeks to solve the same matrix equation $\mathbf{Au} = \mathbf{b}$ as with the LU decomposition. However, when $\mathbf{A}$ is a tridiagonal matrix, some tricks can be implemented. A general tridiagonal $n \times n$ matrix can be written as

$$\mathbf{A} = \begin{pmatrix} d_1 & e_1 & 0 & \cdots & 0 \\ c_1 & d_2 & e_2 & \cdots & 0 \\ 0 & c_2 & d_3 & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & c_{n-1} & d_n \end{pmatrix}$$

As we see here, the entire matrix can be defined through the vector $\mathbf{c}$, $\mathbf{d}$ and $\mathbf{e}$. We will use Gaussian elimination to get rid of the lower diagonal elements, which will introduce a new diagonal vector $\tilde{\mathbf{d}}$ and an altered right-hand-side vector $\tilde{\mathbf{b}}$, where the elements are defined as:

$$\tilde{d}_i = d_i - \frac{c_{i-1} e_{i-1}}{\tilde{d}_{i-1}}$$

$$\tilde{b}_i = b_i - \frac{c_{i-1} b_{i-1}}{\tilde{d}_{i-1}}$$

for $i = 2, ..., n$. Defining $q = c_{i-1}/\tilde{d}_{i-1}$ we can skip one FLOP. The startpoints are unaltered as $\tilde{d}_1 = d_1$ and $\tilde{b}_1 = b_1$. This is the forwards substitution, and it leaves us with the matrix equation

$$\begin{pmatrix} \tilde{d}_1 & e_1 & 0 & \cdots & 0 \\ 0 & \tilde{d}_2 & e_2 & \cdots & 0 \\ 0 & 0 & \tilde{d}_3 & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \tilde{d}_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \vdots \\ \tilde{b}_n \end{pmatrix}$$

[3]Sept. 7, 2020. URL: https://en.wikipedia.org/wiki/LU_decomposition#Using_Gaussian_elimination.

Again using Gaussian elimination, this time to rid ourselves of the upper diagonal elements of our altered matrix A, we obtain the solution for $\mathbf{u}$ as

$$u_i = \frac{\tilde{b}_i - e_i u_{i+1}}{\tilde{d}_i}$$

for $i = n-1, n-2, ..., 1$. This part of the algorithm is the backwards substitution.

Counting the number of floating point operations required using this algorithm, we can see that there are $5(n-1)$ operations for the forward substitution and $3(n-1)$ for the backwards one, for a total of $8n$, as n grows large.

3.3 Thomas algorithm for singularly valued diagonals

For a tridiagonal matrix where all the elements of the three diagonal vectors $\mathbf{c}$, $\mathbf{d}$, $\mathbf{e}$ are respectively the same, we can make some simplifications. In our case $\mathbf{c} = \mathbf{e} = (-1, -1, ..., -1)$, and $\mathbf{d} = (2, 2, ..., 2)$. Plugging these values into the forward substitution of the Thomas algorithm gives us a recursive formula for $\tilde{d}_i$

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

which is a difference equation, the solution of which is

$$\tilde{d}_i = \frac{i+1}{i}$$

This result can just be plugged in, together with the value for $\mathbf{c}$, into the equation for $\tilde{b}_i$:

$$\tilde{b}_i = b_i + \frac{b_{i-1}}{\tilde{d}_{i-1}}$$

This means we get analytical solutions for the coefficients of the backwards substitution of the algorithm:

$$u_i = \frac{\tilde{b}_i + u_{i+1}}{\tilde{d}_i}$$

These simplifications allows us to vectorize the computation of the coefficients and improve the time it takes to execute the substitutions on a computer.

Counting the number of floating point operations now, we can ignore the vectorizable coefficients, and we have simply $2(n-1)$ operations for the forward substitution and $2(n-1)$ for the backwards one, for a total of $4n$ operation for large values of n.

3.4 Benchmarking

How fast a computer can run an algorithm depends on many factors, and simply doing one run for each algorithm at different step lengths h is not very precise. Therefore we wrote a program that iterates over each algorithm and each step length $N = 200$ times. Using the mean of all these values and the standard deviation as an uncertainty we can calculate the experimental time complexities. For the algorithm using LU decomposition we will time the actual LU decomposition, the forward substitution and the backward substitution for $h = 10^{-1}$ to 10^{-3}. For the Thomas algorithm we will time the backward and forward substitution for $h = 10^{-1}$ to $h = 10^{-7}$. When using the Thomas algorithm with singular valued diagonals the calculations for the reduced diagonal elements $\tilde{d}_i$ can be vectorized and we will only time the calculations for $\tilde{b}_i$ and the backward substitution.

4 Results

The results of the numerical solution follow the analytical solution closely. For our simplified Thomas algorithm the numerical solutions for various step sizes are superimposed on the analytical solution in figure 1.

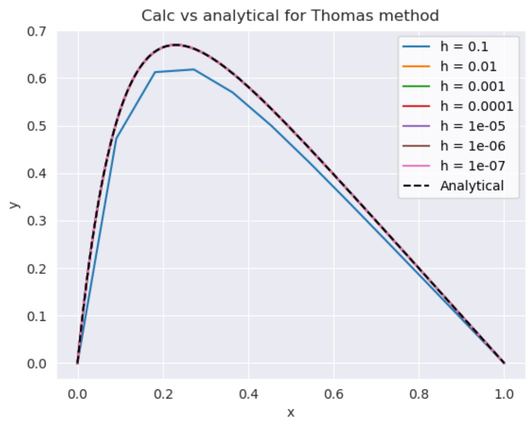

Figure 1: The numerical solution of $\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = 100e^{-10x}$ for step sizes $h = 10^{-1}, ..., 10^{-7}$.

The maximum relative error in any integration point is plotted in figure 2. A linear regression was also fitted over the points that seemed to follow the mathematical error. This returned a slope of 1.98, with a standard deviation 0.01.

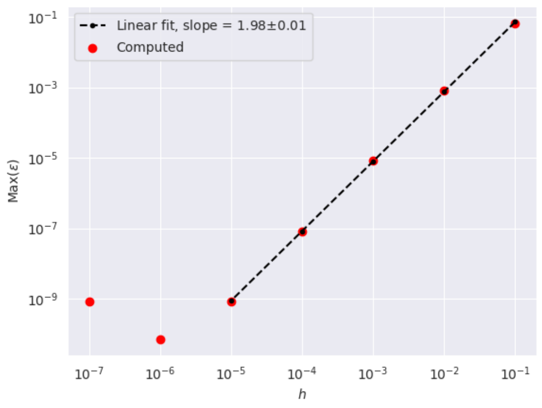

Figure 2: The maximum relative error of any point of the numerical solutions as a function of the step size h. Using logarithmic scale on both axes.

Timing of the various algorithms 200 times each with for $n = 1, 2, ..., 7$ provided the results in figure 3.

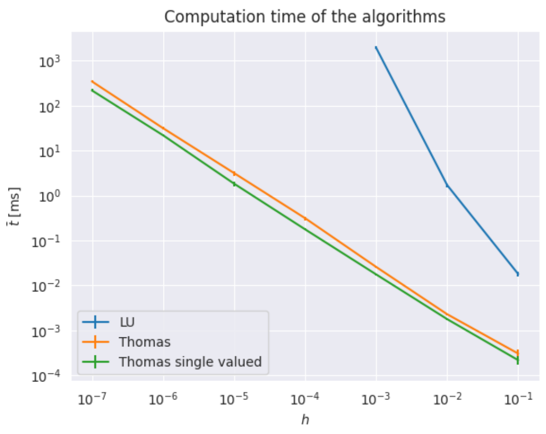

Figure 3: The time it took to compute the numerical solution of the Poisson equation using LU decomposition, the Thomas algorithm, and our simplified Thomas algorithm with error bars.

When timing the different algorithms we ran the LU decomposition with $n = 10, 100$ and 1000 points. Both the Thomas algorithm and Thomas algorithm for singularly valued diagonals was computed for $n = 10$ to $n = 10^7$ integration points. The results are presented in figure 3. The LU decomposition is significantly slower than the two others and unsuitable for small step lengths. The two Thomas algorithms are closer in computational time, but our singular valued algorithm preforms consistently better than the other.

5 Discussion

It is clear from figure 1 that the numerical solutions follow the analytical one closely for step sizes of 0.01 and smaller. Figure 2 gives us a deeper analysis of the validity of our solutions. Already at $h = 0.01$ the maximum relative error is of the order 10^{-3}. The error evolves as we would have predicted mathematically for the until the step sizes grow smaller than 10^{-5}. For this first part, our linear regression showed a slope of 1.98 ± 0.01, which goes to show that the numerical error seems to be $O(h^2)$. Later however, as h gets smaller, we recognize a distinct "hockey stick" shape to the graph, as the error grows larger as the step size is decreased. This is due to the numbers that are used to compute the solution become sufficiently small that the truncation of the decimals becomes significant enough to have an effect on the precision of the result. Thus, it is important to see whether your numerical results could be affected by such errors. Running the algorithm with more step lengths between $h = 10^{-5}$ and $h = 10^{-6}$ could help us determine the optimal step length, but in this report we found that the optimal step length should be somewhere between these two values.

When counting FLOPs we only included the arithmetic operations +, -, *, / and assumed that all of these required the same amount of computation power (on the CPU).[4] This is not true since the cheapest operations are addition and subtraction. Multiplication takes more than these two and division is even worse. In our counting we assumed that all of these are equal which does not reflect reality. Operations like fetching an element from an array and

[4]Sept. 8, 2020. URL: https://streamhpc.com/blog/2012-07-16/how-expensive-is-an-operation-on-a-cpu/.

inserting a value to an array also takes some computation power, but we neglect these as they are very small and the arithmetic operations overshadows them.

In figure 3 we see that the two Thomas algorithms slope much less steeply than the LU-decomposition, which is consistent with their respective number of FLOPs. Furthermore, the refined Thomas algorithm is faster than the general one, as expected. We could only compute the result from the LU decomposition for $n = 1, 2, 3$, since it requires $n \times n$ matrices to be stored in memory. Every entry of this matrix is a double float, requiring 8 bytes each, which results in a memory usage of $10^5 \cdot 10^5 \cdot 8 \approx 8$ GB of memory. Furthermore, if it takes T_p time to compute the $3/2\,10^{3p}$ FLOPs required of the LU decomposition, it should take $T_{p+1} = 10^3 \cdot T_p$ time to compute for the next power of 10. From figure 3 we can see that for $p = 3$ it takes more than a second to compute the LU decomposition, as such it will take at least more than 15 minutes to compute for $p = 4$ and nearly two weeks to compute for $p = 5$. We are on a deadline and do not have time for such tomfoolery.

6 Concluding remarks

We have now seen some of the considerations that need to be taken into account as we solve the one dimensional Poisson equation. The equation is an important one to solve in physics, and generally nice analytical solutions, such as the one we chose, are not available, and we need to resort to numerical approaches to find solutions.

The practical physicist requires two things from these numerical solutions; accuracy and efficiency. Using a simple approximation of the double derivative, we have achieved good accuracy from our solutions, as can be seen in figures 1 and 2. However, as we mitigate the mathematical error, a computational error creeps in, and we cannot achieve arbitrary accuracy. Our approximation of the second derivative was shown to be proportional to $h^{1.98\pm0.01}$, until the truncation error took over.

The algorithms performed differently in terms of efficiency. The LU decomposition was taxing both in therms of computation time and memory usage, although it still produced correct results. The Thomas algorithm, however, was astounding, as it reduced to problem to only require a number of operations proportional to n. This allows us to decrease the step size without sacrificing much in terms of efficiency. We could even simplify the general Thomas algorithm to fit our specific one, to create a slightly faster algorithm, as seen in figure 3.

Thus it is quite possible to achieve both fast and accurate solution to the one-dimensional Poisson equation, using only simple approximations and some linear algebra. However, taking the specific problem into account, and being aware of the limitations of a computer, must be done in order to produce usable results.

References

Sept. 3, 2020. URL: https://en.wikipedia.org/wiki/Poisson%5C%27s_equation.

Sept. 3, 2020. URL: https://en.wikipedia.org/wiki/Gauss%5C%27s_law#Differential_form.

Sept. 7, 2020. URL: https://en.wikipedia.org/wiki/LU_decomposition#Using_Gaussian_elimination.

Sept. 8, 2020. URL: https://streamhpc.com/blog/2012-07-16/how-expensive-is-an-operation-on-a-cpu/.