Numerical simulation of Poisson's equation in 1D.

Steinn Hauser Magnusson, steinnhauser@mac.com

***Abstract*—A numerical solution to a second-order differential equation with Dirichlet boundary is presented using the Thomas algorithm and LU-decomposition. Several degrees of numerical precision were compared to the analytic solution, and the Thomas algorithm was found to be the most effective for both low and high numerical precision. The optimal number of mesh points for this algorithm was found to be $n = 10^5$, as the numerical precision errors increased drastically for a larger amount.**

I. Introduction

THERE are several cases in science where a single mathematical equation can be utilized to describe several physical phenomena. An example of such a mathematical equation is Poisson's equation, which reveals itself in several different fields of physics such as electrostatics, semiconductors, and meteorology. However, the analytic solution to an equation can often times get increasingly difficult or sometimes impossible to solve. That is why numerical approximations are often utilized, and as computational power increases, the incentive to develop algorithms which produce the most accurate results also increases. This paper compares two such algorithms used to solve the Poisson equation in an electromagnetic context; characteristics of the algorithms such as computational speed and memory efficiency are just some of the points of comparison. The layout is designed to go through the theory and algorithm explanation before discussion and comparison of results.

II. Theory

The theoretical model for electrostatic potential Φ in a field with an electric charge density $\rho(\mathbf{r})$ is written:

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

Assuming a spherically symmetric field, and utilizing the substitution $\Phi = \phi(r)/r$ returns:

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

This equation can be further generalized by functions $u(x)$ and $f(x)$, such that the problem is simplified to:

$$-u''(x) = f(x) \tag{3}$$

This is the equation which the algorithms will attempt to solve numerically. However, an analytical solution to the equation is extremely useful when it comes to analyses of the accuracy of the algorithms. That is why the following solution to the differential equation is presented:

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{4}$$

This is the function which will be compared to the analytic calculation. This function returns (from the double derivative) the analytic function $f(x)$ to be:

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

This function will be useful when calculating the second derivative numerically, as knowing what the function looks like once derived twice is essential to the approximation of $u(x)$.

III. Algorithms

Approximating a solution to Poisson's equation involves reconsidering the solution $u(x)$ to be a vector of discrete points $\hat{u} = [u_1, u_2, \ldots, u_{n-1}]$, where n is the number of points of the mesh grid with step size $h = \frac{x_n - x_0}{n+1}$. This allows us to approximate the second order derivative (using Taylor expansion) to be equal to (see [2] for derivation):

$$-u_i'' = -\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = f_i \tag{6}$$

The indexing of the solution u is such that $u(x_0 + ih)$, where i is the current step number $i = 1, 2, 3, \ldots, n-1$.

$$i = 1 \Rightarrow u_0 - 2u_1 + u_2 = -h^2 f_1 \tag{7}$$

$$i = 2 \Rightarrow u_1 - 2u_2 + u_3 = -h^2 f_2 \tag{8}$$

$$i = n-1 \Rightarrow u_{n-2} - 2u_{n-1} + u_n = -h^2 f_{n-1} \tag{9}$$

This system of equations can be further generalized by utilizing the boundary conditions $u(x_0) = 0$ and $u(x_n) = 0$ and generating a matrix A:

$$\hat{A} = \begin{bmatrix} 2 & -1 & 0 & \ldots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & -1 & \ddots & & 0 \\ \vdots & & & 2 & -1 \\ 0 & \ldots & 0 & -1 & 2 \end{bmatrix} \tag{10}$$

Such that the second derivative can be written in terms of:

$$\hat{A}\hat{u} = \hat{b},\ \hat{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \end{bmatrix},\ \hat{b} = h^2 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \end{bmatrix} \tag{11}$$

A. The Thomas Algorithm

The Thomas algorithm involves solving this system of equations through Gaussian elimination, which utilizes linear algebra row matrix operations to solve for the vector $\hat{u}$. In general, it consists of two components: Forward substitution to eliminate the lower diagonal, and then backward substitution to eliminate the upper diagonal. Both components involve multiplying the previous row with a specific value which results in there only being one diagonal left in the end (see appendix [1] for derivation), resulting in a trivial solution. Implemented into a (C++) programming loop, the forward and backwards substitution for this specific tridiagonal matrix can be written in the following fashion:

```
for(int i=2; i<=n; i++){
  dt_vec[i]=2 - 1/dt_vec[i-1];
  ft_vec[i]=f_vec[i] + ft_vec[i-1]/dt_vec[i-1];
}
```

Where the arrays dt_vec and ft_vec are updated for each iteration. The arrays in this case have size $n+2$, where the first and last elements are set to the boundary conditions presented.
The backwards substitution is similarly programmed in the following fashion:

```
for(int i=n; i>0; i--){
  u_vec[i]=(ft_vec[i]+u_vec[i+1])/dt_vec[i];
}
```

This produces the solution to the differential equation $\hat{u}$. The vectors in the program have length $n+2$, where the indexes $i=0$ and $i=n+1$ are described by the boundary conditions. These loops are for the special case matrix $\hat{A}$ presented. In the general case for a tridiagonal matrix of the form

$$\hat{A} = \begin{bmatrix} d_1 & c_1 & 0 & \dots & 0 \\ a_1 & d_2 & c_2 & & \vdots \\ 0 & a_2 & \ddots & & 0 \\ \vdots & & & d_{n-2} & c_{n-2} \\ 0 & \dots & 0 & a_{n-2} & d_{n-1} \end{bmatrix} \tag{12}$$

then the forwards and backwards substitutions can be solved in the following fashion:

```
for(int i=2; i<=n; i++){
  dt_vec[i]=
  d_vec[i]-c_vec[i-1]*a_vec[i-1]/dt_vec[i-1];
  ft_vec[i]=
  f_vec[i]-ft_vec[i-1]*a_vec[i-1]/dt_vec[i-1];
}
for(int i=n; i>0; i--){
  u_vec[i]=
  (ft_vec[i]-c_vec[i]*u_vec[i+1])/dt_vec[i];
}
```

The $\hat{d}$, $\hat{a}$ and $\hat{c}$ arrays can be customized for use in cases different from this one.

B. LU Decomposition

This algorithm involves decomposing the matrix $\hat{A}$ presented previously (in equation 10) into two matrices $\hat{L}$ and $\hat{U}$, such that $\hat{A} = \hat{L}\hat{U}$. $\hat{L}$ is a lower-triangular matrix and $\hat{U}$ is an upper triangular matrix which are written:

$$\hat{L} = \begin{bmatrix} l_{1,1} & 0 & 0 & \dots & 0 \\ l_{2,1} & l_{2,2} & 0 & & \vdots \\ l_{3,1} & l_{3,2} & \ddots & & 0 \\ \vdots & & & l_{n-2,n-2} & 0 \\ l_{n-1,1} & \dots & l_{n-1,n-3} & l_{n-1,n-2} & l_{n-1,n-1} \end{bmatrix} \tag{13}$$

$$\hat{U} = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \dots & u_{1,n-1} \\ 0 & u_{2,2} & u_{2,3} & & \vdots \\ 0 & 0 & \ddots & & u_{n-3,n-1} \\ \vdots & & & u_{n-2,n-2} & u_{n-2,n-1} \\ 0 & \dots & 0 & 0 & u_{n-1,n-1} \end{bmatrix} \tag{14}$$

This allows the problem from equation 11 to be written in terms of $\hat{L}$ and $\hat{U}$:

$$\hat{A}\hat{u} = \hat{b} \Rightarrow (\hat{L}\hat{U})\hat{u} = \hat{b} \Rightarrow \hat{L}(\hat{U}\hat{u}) = \hat{b} \Rightarrow \hat{L}\hat{z} = \hat{b}, \tag{15}$$

where $\hat{U}\hat{u} = \hat{z}$. Solving the equation $\hat{L}\hat{z} = \hat{b}$ with regards to $\hat{z}$ is trivial with a lower triangular matrix, and then solving the equation $\hat{U}\hat{u} = \hat{z}$ is trivial with an upper triangular matrix. This is the essence of the algorithm.
For simplicities sake, this was not implemented into the final program, as it was far easier to solve with the *armadillo* package implemented for C++.

IV. Results

The results presented are all from two programs written in C++ and python. These programs and examples of the results they produce can be found on my github FYS3150 repository: https://github.com/steinnhauser/FYS3150

Following are the results produced by the Thomas algorithm for $n = 10$ and $n = 100$:

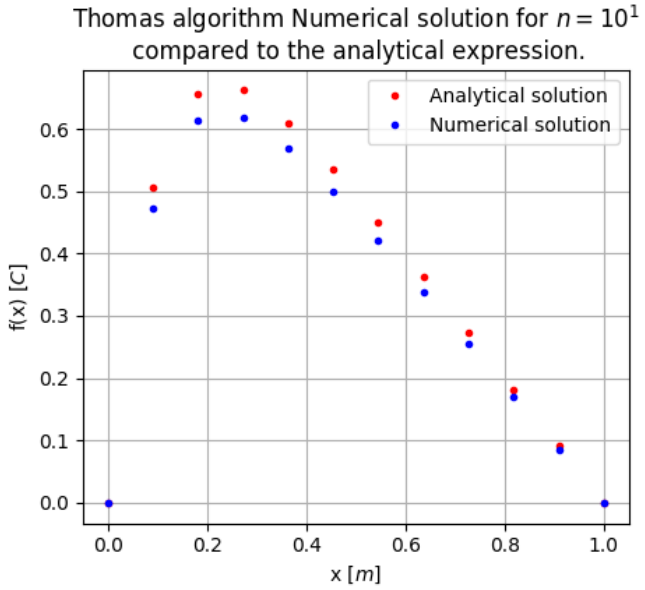

Fig. 1. Thomas algorithm calculation results for $n = 10$ mesh points.

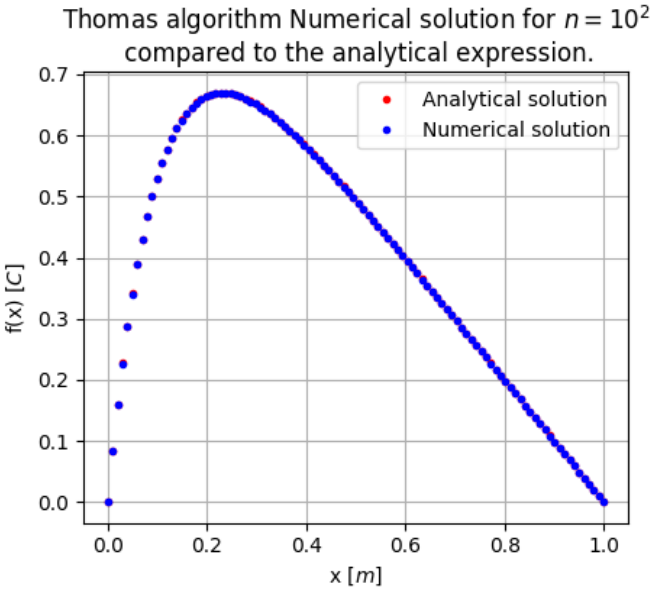

Fig. 2. Thomas algorithm calculation results for $n = 100$ mesh points.

Figures 1 and 2 are plots which illustrate the accuracy of the Thomas algorithm. Following is a logarithmic error plot for

the Thomas algorithm which describes the maximal error ϵ as a function of the step length h:

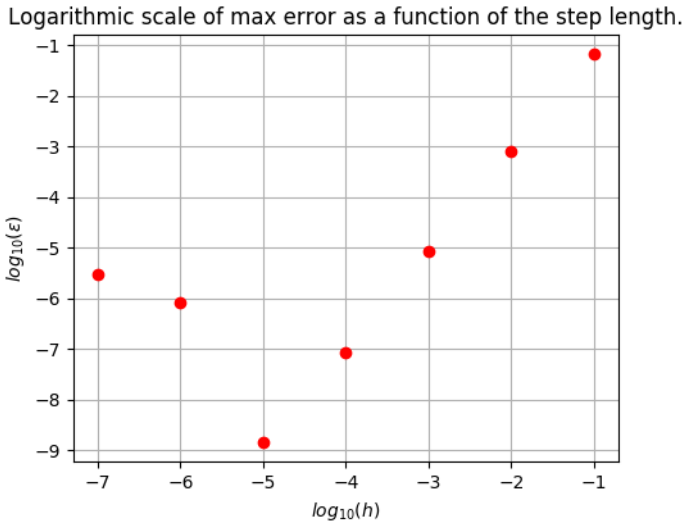

Fig. 3. Maximal relative Thomas algorithm calculation errors.

Figure 3 illustrates the relative errors of the method. These results were produced by the generalized algorithm; the specialized Thomas algorithm program produced the same results. Following are the results produced by the LU-decomposition algorithm for $n = 10$ and $n = 100$:

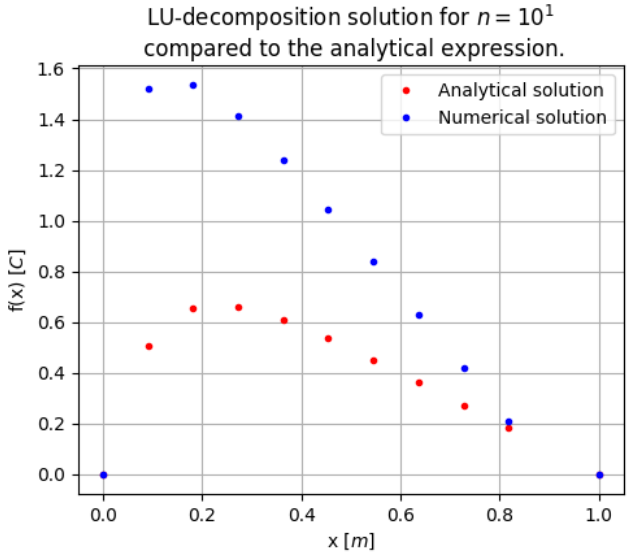

Fig. 4. LU-decomposition calculation results for $n = 100$ mesh points.

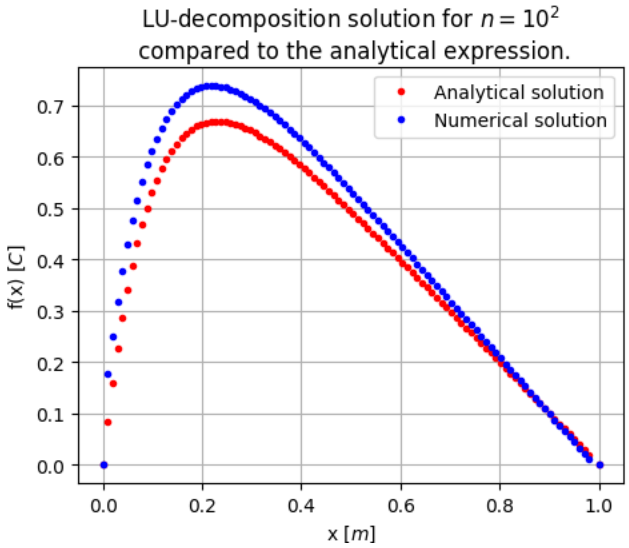

Fig. 5. LU-decomposition calculation results for $n = 100$ mesh points.

Figures 4 and 5 are plots which illustrate the accuracy of the LU-decomposition calculation. Following is a logarithmic error plot for the LU-decomposition method which describes the maximal error ϵ as a function of the step length h:

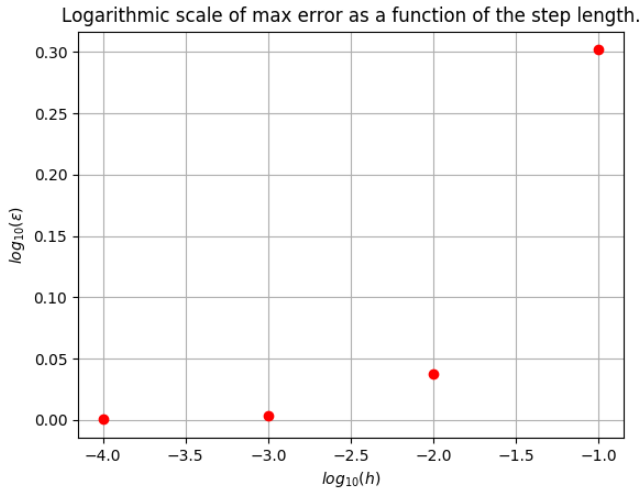

Fig. 6. Maximal relative LU-decomposition calculation errors.

Figure 6 illustrates the relative errors of the method. Following is a table which lists the time it took for the three methods (including the special case Thomas algorithm):

TABLE I
SIMULATION RUN TIMES

$n = 10\wedge$	General Thomas $[s]$	LU $[s]$	Special Thomas $[s]$
1	$4e-6$	$1.17e-4$	$3e-6$
2	$4e-6$	$8.11e-4$	$4e-6$
3	$3.1e-5$	0.29	$2.7e-5$
4	$3.27e-4$	307.45	$3.16e-4$
5	$3.47e-03$	x	$3.32e-3$
6	$3.89e-02$	x	$3.54e-2$
7	0.35	x	0.33

Table I lists the runtime of the simulations and their corresponding mesh-point count. The LU-decomposition method could

not be simulated for more than $n = 1000$ mesh points without running out of memory.

V. Discussion

The results of the methods reveal that the accuracy of the Thomas algorithm is far better than that of the LU-decomposition. Even with a grid of only 10 mesh points produced a reasonable approximation in the case of the Thomas algorithm, which was outstanding in the case of 100 mesh points both in runtime and error analyses. Increasing the number of points further increased the precision of this algorithm, whereas it took up to 1000 mesh points for the LU-decomposition method to produce a somewhat accurate approximation; both $n = 10$ and $n = 100$ did not meet the standards set by the Thomas algorithm in the analytic comparison. Additionally, the runtime and memory requirements of the LU-decomposition with an increasing number of mesh points often caused the calculation to be aborted for $n = 10^4$ points. As a footnote, it's interesting that the LU-decomposition method approaches the analytical solution from above, while the Thomas algorithm approaches it from below.

The relative errors of the Thomas algorithm decrease systematically until a point where $n = 10^6$, where loss of numerical precision causes the errors to increase randomly. This is a useful analyses as the optimal number of mesh points n is found in the balance between minimal error and reasonable runtime, both of which were excellent in the case of the Thomas algorithm with $n = 10^6$ points.

The specialized Thomas algorithm turned out to be slightly quicker than the general one. This is because the number of floating point operations was decreased, resulting in less calculations to be made for each iteration. This specialized version can without a doubt be further optimized to take far less time than the general case; it was surprising that the time difference did not vary more than it did.

VI. Conclusion

In summary, the solution to Poisson's equation was solved to a great degree of accuracy using the Thomas algorithm with only 100 mesh points. Increasing the number of mesh points produced a more accurate result up to a certain number of mesh points n, where the loss of numerical precision caused the error's to begin to increase again. The optimal step size h was found to be $h_{opt} = 10^{-5}$. The numerical method of LU-decomposition was extremely inefficient for increased step sizes, as a step length of only $h = 10^{-4}$ caused the program to spend $307.45s$ to calculate. Afterwards, the calculations turned out to not be nearly as accurate as the results of the Thomas algorithm. Furthermore, although the specialized case of the Thomas algorithm took a shorter time, the time saved wasn't nearly what was expected. This is likely due to other factors than the number of floating point operations for each iteration, as it was certainly reduced in the special case.

REFERENCES

[1] Hjorth-Jensen, M. (2015). *Computational Physics - Lecture Notes 2015*. Univerity of Oslo. http://compphysics.github.io/ComputationalPhysics/doc/web/course

[2] Hjorth-Jensen, M. (2015). *Computational Physics - Lecture Notes 2015*. Univerity of Oslo. https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf

[3] Hjorth-Jensen, M. (2018). *Computational Physics - Project 1, 2018*. Univerity of Oslo. https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Projects/2018/Project1/pdf/Project1.pdf