FINAL VERSION SEPTEMBER 10, 2018
Preprint typeset using LaTeX style emulateapj v. 12/16/11

POISSON'S EQUATION SOLVED THROUGH OPTIMAL ALGORITHMIC COMPUTATION USING GAUSSIAN ELIMINATION PROJECT 1 - FYS4150 / FYS3150

JAKOB BORG, JULIE THINGWALL & WANJA PAULSEN

Final version September 10, 2018

ABSTRACT

In this project we explore numerical algorithms with our computer as an experimental setup. We solve the well known Poisson equation for a one dimensional electrostatic potential through the Gaussian elimination method of a set of linear equations. We do this with two different algorithms developed to solve the tridiagonal matrix problem, one general case and one specialized to solve for the second derivative. Our goals is to compare the accuracy and CPU time of our numerical solvers. We find that the numerical solution does not converge ever closer to the exact one with ever decreasing step size as the theory would suggest, instead we find an optimal step size of $h = 10^{-6}$. We also compare our results against the solution found from the LU-decomposition of the matrix problem. Here we quickly find a limit to the max capacity of this method, as it will not be able to operate on matrices with bigger dimensions than an order of $n = 10^4$ as our computer run out of memory to store the matrix. Performing the full LU-decomposition is also extremely CPU heavy, as the number of floating point operations (FLOPS) scales as $\mathcal{O}\left(\frac{2}{3}n^3\right)$, while we find the number of flops in our general and specialized algorithms to scale as $\mathcal{O}(8n)$ and $\mathcal{O}(4n)$ respectively.

Subject headings: ODE — Poisson equation — Gaussian Elimination — LU-decomposition — Numerical errors

1. INTRODUCTION

In the research field of physics we always strive to describe and explain the real physical world, as well as predicting what we cannot observe. We do this by the use of mathematical tools, experimental setups and computers, where the latter continues to increase it's significance in the research world. One important mathematical field is widely used both to explain and predict the behaviour of the physical world, the field of differential equations. Differential equations are fantastic tools to study the rates of change in a system, as well as the quantity of change. Unfortunately there are few realistic systems that can be solved through analytic solutions, which is where the field of computational physics has tremendous power. By using computational force one can solve these problems in the matter of seconds through numerical methods, which would take hours by the use of pen and paper.

The aim of this project is to explore one such set of differential equations, namely the one-dimensional Poisson equation with the use of Dirichlet boundary conditions. This will be done by the use of two different, but similar, methods from linear algebra which will be compared in numerical computing efficiency and checked against known methods. The programming language used for this project is C++ as well as Armadillo, a C++ library for linear algebra and scientific computing REF. The methods used are Gaussian Elimination as well as LU-decomposition, both solve sets of linear differential equations.

julie.thingwall@astro.uio.no
wanja.paulsen@fys.uio.no
jakobbor@student.matnat.uio.no
[1] Department of Physics, University of Oslo, P.O. Box 1048 Blindern, N-0316 Oslo, Norway

Another important aspect of this project is to inquire on the errors in our experimental setup, a computer. We discuss numerical error risks in computers in general as well as numerical errors in our specific algorithms.

2. THEORETICAL BACKGROUND

2.1. *Poisson Equation*

Poisson's equation is a well-known differential equation often used to describe the potential field from a charge in a electrostatic field or the mass distribution in a gravitational field. Generally the Poisson equation can be written as

$$\nabla^2 \phi = f \tag{1}$$

in Euclidean space, but in this project we will look at the Poisson equation used to describe electrostatic potential by a localized charge distribution $\rho(\mathbf{r})$. Thus our Poisson equation reads

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

In this project we will simplify to a one-dimensional problem by assuming a spherically symmetric $\mathbf{\Phi}$ and $\rho(\mathbf{r})$. Thus eq. (2) can be rewritten as

$$\frac{1}{r^2}\frac{\mathrm{d}}{\mathrm{d}r}\left(r^2\frac{\mathrm{d}\mathbf{\Phi}}{\mathrm{d}r}\right) = -4\pi\rho(r). \tag{3}$$

By substituting $\mathbf{\Phi}(r) = \frac{\phi(r)}{r}$ we have

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

By letting $\phi \to u$, $r \to x$ and $f(r \to x) = -4\pi x\rho(x)$ be our source function, our one-dimensional Poisson equation can be written as

$$-u''(x) = f(x). \tag{5}$$

2.2. *Gaussian Elimination*

Most of the algorithms used in solving sets of linear differential equations are based on the method of Gaussian elimination (2). It can be described as a sequence of operations which are performed on a matrix $\mathbf{A}$, with the intent of reducing it to a upper-triangular matrix in row echelon form (2). The method starts out with a set of linear equations on the form

$$\mathbf{A}\mathbf{x} = \mathbf{w} \tag{6}$$

where $\mathbf{A}$ is a non-singular matrix with the diagonal satisfying the condition $a_{ii} \neq 0$. The process of reducing this to an upper triangular matrix starts with an algorithm to find the new coefficients of the matrix. The first new coefficients are given as

$$b_{1k} = a_{1k} \quad k = 1, 2, ...n. \tag{7}$$

With this the rest of the coefficients are given as

$$b_{jk} = a_{jk} - \frac{a_{j1}a_{1k}}{a_{11}} \quad j,k = 2,3,...,n. \tag{8}$$

These steps are also called the forward substitution, whilst the process of finding the vector $\mathbf{x}$ is called the backward substitution. This can be written as a sum so that

$$x_m = \frac{1}{b_{mm}}\Big(y_m - \sum_{k=m+1}^{n} b_{mk}x_k\Big) \quad m = n-1, n-2, ..., 1. \tag{9}$$

Thus the solution to the set of linear equations is given by the vector $\mathbf{x}$ (2).

2.3. *LU-Decomposition*

LU-decomposition is a method for factorizing a matrix into a upper and lower triangular matrix, the method is much like a matrix form of Gaussian elimination(section **??**) (2). A matrix equation on the form

$$\mathbf{A}\mathbf{x} = \mathbf{w} \tag{10}$$

can be rewritten by LU-decomposition so that

$$\mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{w}. \tag{11}$$

The method can only be used on matrices $\mathbf{A} \in \mathbb{R}^{n\times n}$ where the determinant of $\det(\mathbf{A}) \neq 0$ (2). This calculation is done in two steps, by solving the following matrix equations

$$\mathbf{L}\mathbf{y} = \mathbf{w} \quad , \quad \mathbf{U}\mathbf{x} = \mathbf{y}. \tag{12}$$

Since $\det(\mathbf{L} = 1)$ by construction it has an inverse and we can thus rewrite eq. (12) to

$$\mathbf{U}\mathbf{x} = \mathbf{L}^{-1}\mathbf{w} = \mathbf{y}. \tag{13}$$

This means that we can obtain both $\mathbf{L}$ and $\mathbf{U}$ by standard linear algebra operations and achieve our goal of LU-decomposing the matrix $\mathbf{A}$ (2).

3. METHOD

3.1. *Discretizing the equations*

To solve the Poisson equation numerically, we need to discretize it. In general, one can discretize a double derivative by the use of Taylor expansion. When doing this for a function $u(x)$ one get

$$u(x \pm h) = u(x) \pm h\frac{\mathrm{d}u}{\mathrm{d}x} + \frac{h^2}{2!}\frac{\mathrm{d}^2u}{\mathrm{d}x^2} \pm \frac{h^3}{3!}\frac{\mathrm{d}^3u}{\mathrm{d}x^3} + \mathcal{O}(h^4), \tag{14}$$

where h is the step size. This expression can then be rearranged to solve for the double derivative, which yields

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + \mathcal{O}(h^2). \tag{15}$$

By introducing the notation

$$x \to x_i \quad i = 0, 1, ..., (n+1)$$
$$u(x) \to u(x_i) \to v_i$$

eq. (15) can be rewritten as

$$\frac{\mathrm{d}^2v_i}{\mathrm{d}x^2} = \frac{v_{i+1} + v_{i-1} - 2v_i}{h^2}, \tag{16}$$

where x_i are discrete values in the range $x \in [0, 1]$, with stepsize $h = x_{i+1} - x_i = \frac{1}{n+1}$.

Using eq. (16) and eq. (5), the discretized Poisson Equation can be written as

$$-(v_{i+1} + v_{i-1} - 2v_i) = h^2 f(x_i) = f_i. \tag{17}$$

As mentioned in the introduction, we look at a case with Dirichlet boundary conditions, meaning the end points v_0 and v_{n+1} are both 0. Taking this into consideration, we only need to solve for v_1 to v_n in our program. To see how we can rewrite eq. (17) as a matrix equation, we look at a special case where $n = 4$.

$$i = 1 : -v_2 - v_0 + 2v_1 = f_1$$
$$i = 2 : -v_3 - v_1 + 2v_2 = f_2$$
$$i = 3 : -v_4 - v_2 + 2v_3 = f_3$$
$$i = 4 : -v_5 - v_3 + 2v_4 = f_4$$

This can immediately be recognized as a matrix equation on the form $\mathbf{A}\mathbf{v} = \mathbf{f}$, where the matrix $\mathbf{A}$ is on the form

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} \tag{18}$$

This can be generalized to an $n \times n$ matrix, and the equation we end up wanting to solve numerically looks like

$$\begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ & & & & & \vdots \\ 0 & \dots & \dots & \dots & \dots & \vdots \\ \vdots & \dots & 0 & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ \\ \vdots \\ f_n \end{pmatrix} \tag{19}$$

3.2. *Numerical algorithms*

With the problem discretized we can develop algorithms to perform the reduction of matrix $\mathbf{A}$ through the Gaussian elimination method. As we solve for the second derivative, the matrix $\mathbf{A}$ is tridiagonal. This is indeed helpful as we don't have to operate on the entire matrix, but only the main diagonal and the first diagonal above and under it. We store these diagonals as vectors **a**, **b** and **c**, and operate only on the vectors and not the entire matrix itself which includes a lot of zeros. Let **b** be the main diagonal, **a** the one under it and **c** the one above. Also, let **f** be the righthand side vector containing the source function and we'll use $\tilde{\mathbf{b}}$ as notation for a vector after it is manipulated through the Gaussian elimination.

In this project we explore two different numerical methods for manipulating these vectors and solving the tridiagonal problem. One generalized algorithm for any vectors on the form we have discussed, and one specialized to solve specifically for the second derivative. As we have seen this can be used to solve the Poisson equation eq. (5), which has many known closed form solutions, one of which we have used to test our numerical methods. With an assumed source function $f(x)$

$$f(x) = 100\exp(-10x) \tag{20}$$

one can find the solution $u(x)$

$$u(x) = 1 - (1-\exp(-10))x - \exp(-10x). \tag{21}$$

We have also tested our results against a solution found using LU-decomposition from the standard library Armadillo (1). This method is known to work, but has several draw backs as well. Performing the full LU-decomposition of a big matrix is CPU consuming. The smaller step size one uses in the discretization, the bigger the matrix has to be. Therefore we can obtain more precise accuracy using large matrices and better algorithms than we can using LU-decomposition. With this in mind we have in addition to comparing our two algorithms compared their computing time against the LU-decomposition method. Here we quickly find the limits of how big matrices Armadillos LU-decomposition can handle.

In both algorithms we have pre-calculated as much as possible, to only include the necessary steps for the forward and backward substitutions needed in each case. As we have seen we only need to find the new coefficients for the main diagonal and the right-hand side of the matrix equation. The calculation of the source function and the exact solution are pre-calculated for both algorithms.

We have run multiple calculations, each with decreasing step size which results in larger matrices. This way we can see how our numerical solutions converge to the exact solution with ever decreasing step size, until the point where the calculations start losing accuracy. Here we will time how our algorithms perform and explore the relative error for up to $n = 10^8$.

3.2.1. *General algorithm*

In the general solver we have to take into account that the three vectors may not have constant coefficients. Therefore we use the full expressions used for the new coefficients in the Gaussian elimination process.

For the forward substitution of vector **b** we use eq. (8) which looks like

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

using our vectors as matrix elements. During the row operations in the Gaussian elimination method one also has to update the righthand side of the equation represented with our vector **f**. This is done with the expression

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

As these steps are results of the same row operation both use a factor $\frac{a_i}{\tilde{b}_{i-1}}$. Knowing this we can calculate it only once and use this value in both steps, then the forward substitution is done with a total of $5n$ FLOPS for a $n \times n$ matrix.

In the backward substitution of **v**, having done the reduction of the matrix we can find the solution using the sum of coefficients in the Gaussian elimination process presented in eq. (9). In our vector notation the expression becomes

$$\tilde{v}_{i-1} = \frac{\tilde{f}_{i-1} - c_{i-1}v_i}{\tilde{b}_{i-1}} \tag{24}$$

where the endpoint is found by $\tilde{v}_n = \frac{\tilde{f}_n}{\tilde{b}_n}$. This process require $3n$ FLOPS, such that in total the hole general algorithm is performed with $8n$ FLOPS. This is implemented in our program as

```
void gauss_general(uword n,vec &a,vec &b,vec
    &c,vec &f,vec &sol){
    double factor;
    for (uword i = 1; i < n; ++i ){
        factor = a[i-1]/b[i-1];
        b[i] = b[i] - c[i-1]*factor;
        f[i] = f[i] - factor*f[i-1];
    }
    sol[n-1] = f[n-1]/b[n-1];
    for (int i = n-2; i >= 0;--i){
        sol[i] = (f[i]-c[i]*sol[i+1])/b[i];
    }

}
```

3.2.2. *Specialized algorithm*

In our specific solver we use the fact that we know the value of each element along the tridiagonal matrix as it is the matrix representation of the set of equations for the second derivative. All elements along the main diagonal represented by vector **b** are set to 2 and the first upper (**c**) and lower (**a**) diagonals are set to -1. We can use this fact to precalculate some of the steps in the general elimination algorithm, and also speed up the algorithm by reducing the number of FLOPS needed in the forward and backward substitution.

We can precalculate all the new main diagonal elements $\tilde{b}_i$ by using that all a_i and c_i equals -1, and b_i equals 2.

$$\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}} \tag{25}$$

By starting with boundary condition and $\tilde{b}_1 = b_1 = 2$ we find an optimized analytical expression for the main diagonals that may be fully precalculated

$$\tilde{b}_1 = \frac{i+1}{i}. \tag{26}$$

We can optimize the forward substitution further by reducing the number of FLOPS when updating the source function $f(x)$ on the righthand side

$$\tilde{f}_i = f_i + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}} \quad (27)$$

where we know $\tilde{b}_{i-1}$ from the precalculation, suchs that the forward substitution can be done with only 2 FLOPS.

The backward substitution can also be optimized in a similar way to

$$v_{i-1} = \frac{1}{\tilde{b}_{i-1}}\left(\tilde{f}_{i-1} + v_i\right) \quad (28)$$

where the endpoint is found by $\tilde{v}_n = \frac{\tilde{f}_n}{\tilde{b}_n}$ again. This step also requiring 2 FLOPS such that the hole specialized algorithm requires $4n$ FLOPS, half of that of the generalized. The specialized algorithm is implemented as

```
void gauss_specific(int n, vec &b, vec &f,
    vec &u){
    for (int i = 1; i < n; i++){
        f[i] = f[i] + f[i-1]/b[i-1];
    }
    u(n-1) = f(n-1)/b(n-1);
    for (int i = n-2; i >= 0; --i) {
        u[i] = (f[i]+u[i+1])/b[i];
    }
}
```

with the main diagonals in vector **b** is precalculated.

3.2.3. *LU-Decomposition algorithm*

As mentioned in section 3.2, we use the library Armadillo (1) to perform the LU-decomposition. Unlike the generalized and specialized algorithm, the Armadillo functions LU and solve requires a matrix as an input, not vectors, which means we now have to work with the full matrix **A**. This will naturally affect the number of FLOPS needed to do the calculations. In section 2.3 we describe how LU-decomposition works in general, and from this we calculate that the total number of FLOPS needed to calculate the whole algorithm scales with n as $\frac{2}{3}n^3$. In our program we end up solving the equations using LU-decomposition for $n = 10^4$, as any matrices bigger than this leads to memory errors.

3.2.4. *Numerical uncertainty*

All experimental setups have sources of uncertainties and errors, this is also true for computational experiments. In this project we expect that our main source of error is the relative error accumulated from problems with arithmetic underflow (2) as the step size h becomes smaller and smaller as n increases. This is because a computer cannot represent an infinitesimal number (2). Normally this error can be neglected, but we have experimented as to how small we can make the step size before this error gives significant loss of precision.

An important inhibitory step to avoid computational errors is to write your algorithms such that they are computer-friendly, i.e. use as simple mathematical operations as possible. As mentioned will writing simple algorithms save memory usage, minimize numerical uncertainty and save time.

The relative error was computed for our general/specialized algorithm given in section 3.2 for increasing number of step lengths n, where the number of step lengths increase as 10^n. The calculation was done using

$$\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \quad (29)$$

where v_i denotes the numerical solution of eq. (17) and u_i denotes the exact solution given in eq. (21). From the Taylor expansion (eq. (16) we expect to get a mathematical error scaling with h^2. This means we should expect to see a graph with slope $|2|$ when plotting the logarithmic error against the step size. We expect to be able to see the effect of arithmetic underflow in this graph as we push the step size smaller and smaller.

We also measured how time-consuming the used algorithms were. We did this by initiating a function for measuring time at each algorithm executing the algorithms 5 times and taking the mean value. Doing this gives us information about how efficient our algorithms can be, which is important if they will be used in problems where the number of steps increase to a very large number.

4. RESULTS

A comparison of how well the algorithm for solving the discretized Poisson equation in eq. (17) versus the exact solution in eq. (21) for several values for the length n by Gaussian elimination is presented in fig. 1. In this figure we clearly see that for increasing n, the numerical solution converges towards the exact solution, as we expected.

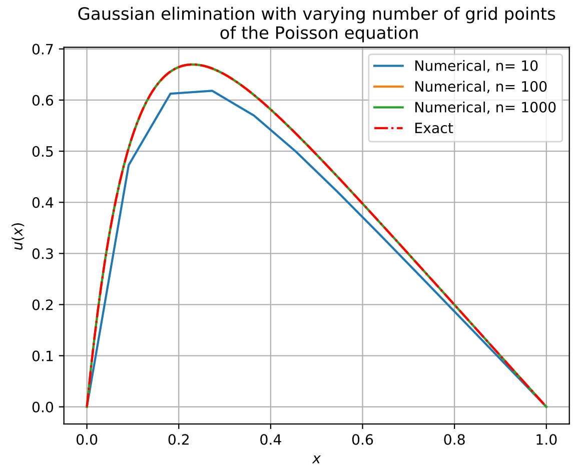

FIG. 1.— Figure comparing our numerical solution of eq. (17) with three different number of grid points against the exact solution on the scaled interval $x \in [0, 1]$. Here one can see how the numerical solution converge to the exact one for increasing n from $n = 10$ to $n = 100$ and $n = 1000$.

A comparison between LU-decomposition and Gaussian elemination for $n \in [10, 1000]$ can be seen in fig. 2. Here we see that the LU-decomposition and Gaussian elimination method yields the same results. But, as discussed in the method-section, we know that LU-decomposition requires more FLOPS than both our general and specialized Gaussian-based algorithm. The consequences of this can be seen in table 2, where there is a clear difference in run time between the generalized- and specialized method and the LU-decomposition. As expected, the LU-decomposition is much slower than the other two methods. Meanwhile, as can be seen in in fig. 2, all methods have the same numerical precision. LU-decomposition is neither more nor less accurate than Gaussian elimination for equal value for n. It is worth mentioning that when using LU-decomposition one is limited on the size of the matrix **A**, which in turns leads to a limitation in accuracy of the solution. While the Gaussian elimination reaches maximum accuracy at matrices of the size 10^6, the LU-decomposition is limited at 10^4. With matrices larger than this, we experience problems with computer memory.

Gazing at table 2 we can also see the result of timing both the general and specialized methods. There is a slight difference and the specialized is indeed faster than the generalized method. We observe that for $n = 10$, the specialized method is considerably faster, but for larger n the difference decreases. These values were obtained by running our algorithms 5 times for several values for the number of grid points n and calculating the mean of this. The reason for this is to ensure that our results of time consumption is of acceptable accuracy.

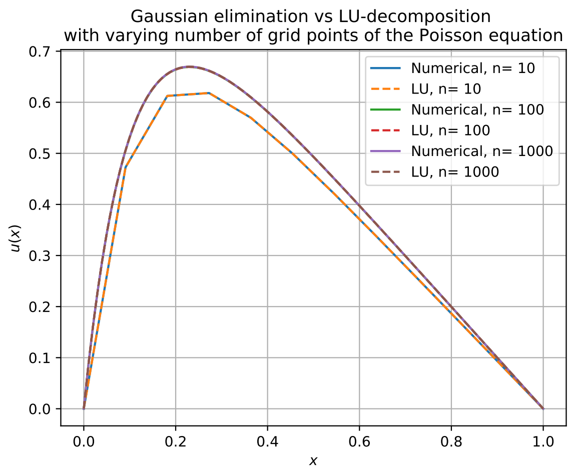

FIG. 2.— Figure comparing how well our numerical algorithms for Gaussian elimination can replicate the results from LU-decomposition for several different number of grid points n. Both the general and specialized algorithm for Gaussian Elimination is included.

We calculated the relative error from eq. (29) for our specialized algorithm in section 3 and tabulated the resulting values in table 1. The table shows the maximum value of the relative error as the number of steps n increase. We see how the relative error decreases as the number of grid points n increase until $n > 6$, where the relative error starts increasing. In fig. 3 we have plotted the relative error with respect to the step size h. Here we see that it indeed follows a slope $|2|$ as to be expected. At $n > 6$, we see that the relative error shows an increasing trend, which we also expected would happen for smaller step sizes. But as we can see in fig. 1, we reach acceptable levels of accuracy long before the step size becomes so small that numerical errors kicks in.

TABLE 1

Value of n in 10^n	Max value of relative error
10	6.61153372855384E-02
10^2	8.16513121740949E-04
10^3	8.31665015291901E-06
10^4	8.33167389814800E-08
10^5	8.33487253694738E-10
10^6	6.87336368474605E-11
10^7	8.12744443760592E-10
10^8	7.42313473332000E-09

NOTE. — Table showing the relation between increasing exponent n and the following maximum relative error value. Relative error calculated using eq. (29).

TABLE 2

n	Generalized	Specialized	LU-decomposition
10	4.80E-06	1.20E-06	7.70E-4
10^3	1.78E-05	1.38E-05	0.10
10^4	1.61E-4	1.34E-4	
10^7	0.15	0.13	

NOTE. — Table showing the mean time in seconds for each algorithm for selected step lengths n.

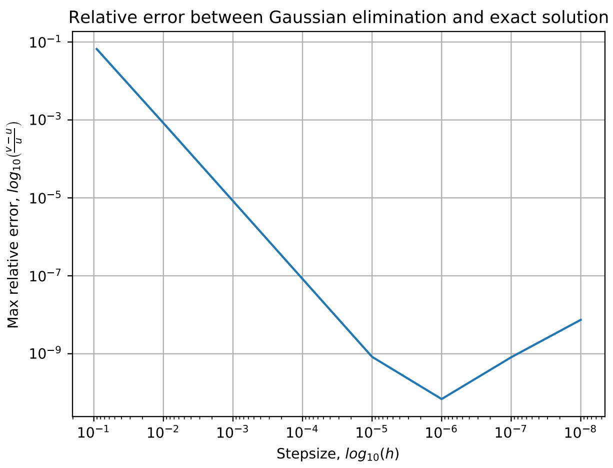

FIG. 3.— Relative error calculated from eq. (29) in logarithmic scale with the step size h for each number of grid points n in logarithmic scale.

5. PERSPECTIVES FOR FUTURE IMPROVEMENTS

As this is our first proper project where we have used C++ as the main programming language, we note that the structure of the program and several small details could be set up differently. By structuring the main function more efficiently we could increase the readability of the program, and make it easier to follow the work flow. This could be done by increasing the number of functions used, to reduce the number of lines and operations performed by the main function itself. Most of the memory allocations for the big vectors are done with Armadillo, which streamlines the process, but this as well could be done outside the main function body. Further improvements could be done by making the program object oriented so that it is easier to use in the future for different problems. This could also be done without object orientation if we included more control and checks of how the user want to run the program, as well as incorporate the different programs used for plotting and analyzing the results into the main function. We also feel like our programming style from Python shines through the code, so one additional source for improvements would be to follow more standard C++ conventions.

6. CONCLUSIONS

Our intent of exploring how well we could solve the Poisson equation for describing an electrostatic potential numerically has been achieved with beneficial results. We have discretized the function to be able to solve this non-linear differential equation through methods from linear algebra which solves sets of linear differential equations. We programmed algorithms by using the method Gaussian Elimination and backwards substitution for both a generel tridiagonal matrix as well as our special case of a tridiagonal matrix. By checking our results with an algorithm implemented in C++ which uses LU-decomposition as method we could confirm that our algorithms did not just solve the Poisson equation successfully, they solved it in a shorter time span than the implemented function could. The general Gaussian Elimination method solved the problem in $1.78E-05$ seconds whilst LU-decomposition used 0.1 seconds for $n = 10^3$ points. Our specialized method was even faster at $1.38E-05$ seconds. We could not measure the time span for grid points larger than $n = 10^4$ because of limited computer memory for LU-decomposition, which again shows how efficient our algorithms were for this problem. How well our specialized algorithm could reproduce the results from the LU-decomposition were also positively successful in fig. 2.

We also compared our numerical results with the exact solution in fig. 1 with our specialized algorithm of the problem with definite success. We saw that we reached a sufficient precision at $n = 100$ grid points, which is also far beyond the region where our relative error had significant impact. How arithmetic underflow could influence our results was also investigated further, which resulted in a interesting result of increasing relative error when grid points exceeded $n > 10^7$ in table 1 and fig. 3.

7. APPENDIX: LINK TO ALL PROGRAMS

Link to all programs in Github

REFERENCES

[1]C. Sanderson and R. Curtin. Armadillo: a template based c++ library for linear algebra. Journal of Open Source Software, Vol 1., pp.26, 2016. [Online; accessed 10-September-2018].

[2]H.J. Morten. Computational physics - lecture notes fall 2015, 2015.